

Programação de Computadores

Capítulo 1: Introdução



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a Professora Teresa Martinez.

SEBENTA - Material de Apoio

Programa

1. Computadores e Programas
2. Conceitos sobre tipo e representação de dados em computadores
3. Noções básicas sobre programas e programação estruturada
4. Operações básicas sobre dados
5. Controlo de fluxo
6. Funções
7. Endereços e ponteiros
8. Entrada e Saída (leitura e escrita de ficheiros)
9. Estruturas de dados
10. Algoritmos e exemplos práticos

Bibliografia

Bibliografia Base

- [1] Luís Damas, Linguagem C, FCA – Editora de Infomática, 1999, 24^a Edição em 2015
- [2] Alexandre Pereira, C e Algoritmos, Edições Sílabo, 2013.
- [3] The C Programming Language, 2nd edition Brian Kernighan e Dennis Ritchie, Prentice Hall, 1988.

Carga Horária

- ▶ Esta unidade curricular possui **6 ECTS** o que corresponde a **162 horas de trabalho neste semestre**
- ▶ Logo, devem trabalhar cerca de **8 horas por semana** nesta unidade curricular.
- ▶ Existem 2h (PL c/ componente teórica) + 2h (PL) por semana.
- ▶ As 4h/semana são do tipo PL (ensino Prático e Laboratorial) sendo que uma parte das aulas PLs serão dedicadas ao ensino dos conceitos e fundamentos de programação em C.
- ▶ **Chave para o sucesso em Programação de Computadores: estudo e trabalho regulares.**

Aulas Laboratoriais

- ▶ Em geral, haverá uma folha/Ficha com problemas por cada aula laboratorial (com algumas exceções, para sincronização das turmas devido aos feriados).
Os estudantes deverão ocupar as restantes, aproximadamente, 8 horas semanais com estudo individual/independente, nomeadamente na resolução dos problemas das Fichas que não tenham conseguido terminar em aula.

Aulas Laboratoriais

- ▶ Em geral, haverá uma folha/Ficha com problemas por cada aula laboratorial (com algumas exceções, para sincronização das turmas devido aos feriados).

Os estudantes deverão ocupar as restantes, aproximadamente, 8 horas semanais com estudo individual/independente, nomeadamente na resolução dos problemas das Fichas que não tenham conseguido terminar em aula.

- ▶ O compilador de linguagem C que iremos utilizar é o gcc do GCC (*the GNU Compiler Collection*) em linha de comando, num terminal com sistema operativo Windows. Iremos utilizar o editor de texto [Geany](#).

Aulas Laboratoriais

- ▶ Em geral, haverá uma folha/Ficha com problemas por cada aula laboratorial (com algumas exceções, para sincronização das turmas devido aos feriados).
Os estudantes deverão ocupar as restantes, aproximadamente, 8 horas semanais com estudo individual/independente, nomeadamente na resolução dos problemas das Fichas que não tenham conseguido terminar em aula.
- ▶ O compilador de linguagem C que iremos utilizar é o gcc do GCC (*the GNU Compiler Collection*) em linha de comando, num terminal com sistema operativo Windows. Iremos utilizar o editor de texto [Geany](#).
- ▶ No UC-Student estão instruções para instalação em Windows do Geany e compilador em C.

Computadores, programas e computação

Elementos Básicos de um Computador: visão de alto nível

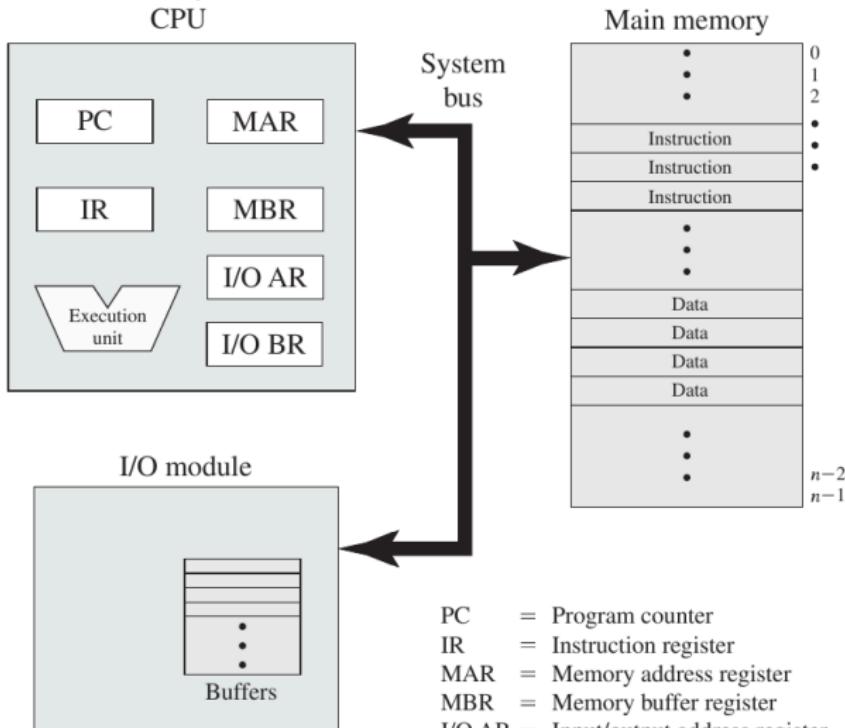


Figura: Elementos de um computador: visão de alto nível (Em Fig. 1.1 de William Stallings, *Operating systems : internals and design principles*, Boston, Pearson, 2015)

Computadores, programas e computação

Ciclo básico de Instruções (sem interrupções)

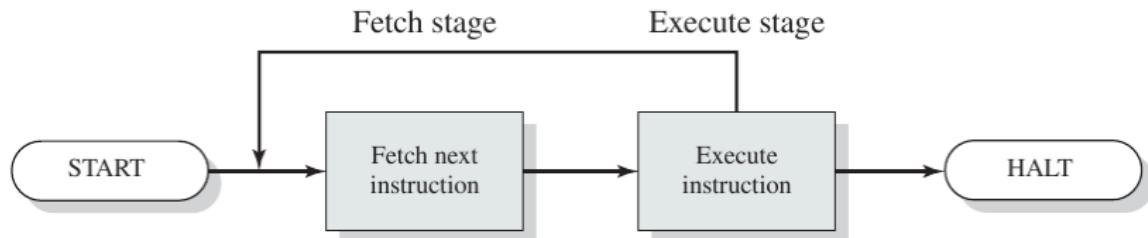


Figura: Ciclo básico de Instruções (Em Fig. 1.2 de William Stallings,
Operating systems : internals and design principles, Boston, Pearson, 2015)

Computadores, programas e computação

Computadores

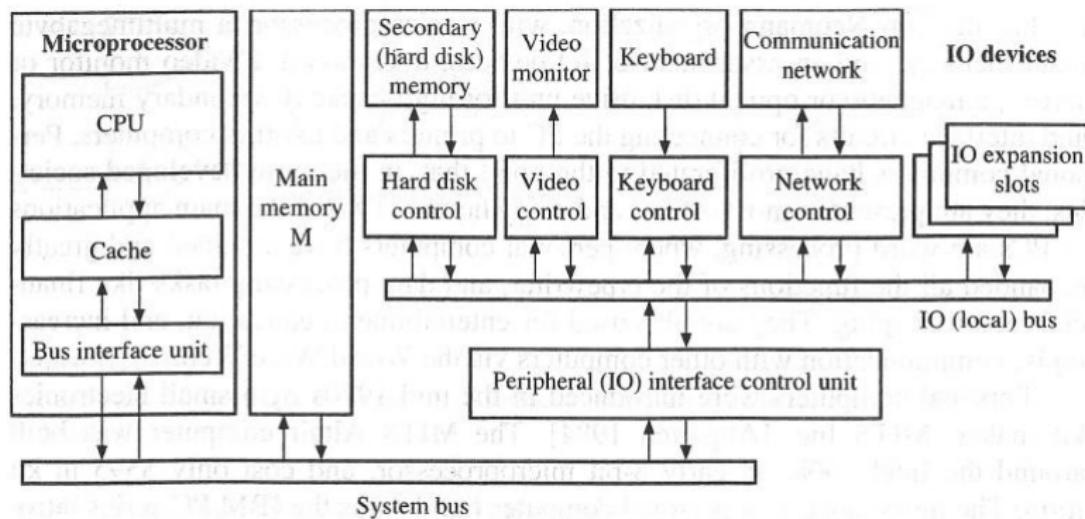


Figura: Um Computador Pessoal Típico (em J. P. Hayes. "Computer Architecture and Organization", Mc-Graw Hill, 1998).

Computadores, programas e computação

Programas

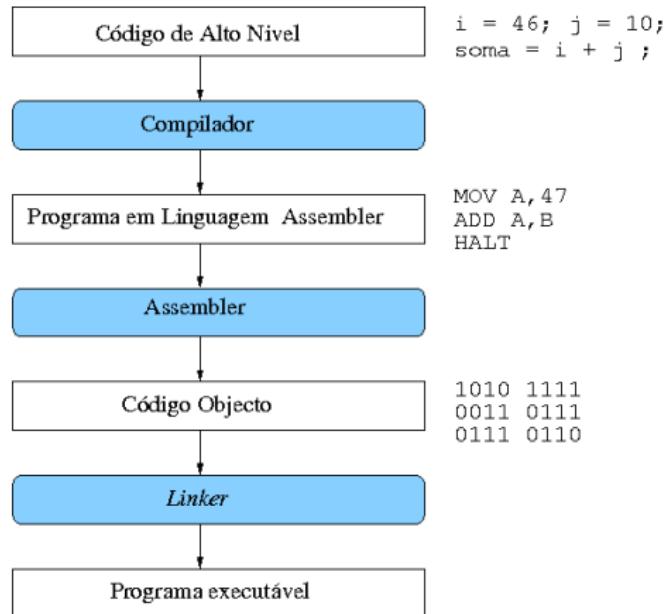


Figura: Transformação dum programa de alto nível num programa executável

Computadores, programas e computação

Computação

- ▶ A computação pode ser definida como a busca de uma solução para um problema, a partir de entradas (*inputs*), e através de um algoritmo.
- ▶ Durante milhares de anos, a computação foi executada com caneta e papel, ou com giz e ardósia, ou mentalmente, por vezes com o auxílio de tabelas, ou utensílios artesanais.

Origem: Wikipédia, a encyclopédia livre.

Programação de Computadores

Capítulo 2: Representação de informação em computadores



(PdC)

Plano: Representação de informação em computadores

1. Notação binária
2. Números em vírgula flutuante
3. Gamas em notação binária
4. Representação Octal e Hexadecimal
5. A tabela ASCII

Notação binária

- A notação decimal possui dez símbolos diferentes:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$$256 = 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

$$12.07 = 1 \times 10^1 + 2 \times 10^0 + 0 \times 10^{-1} + 7 \times 10^{-2}$$

Notação binária

- A notação decimal possui dez símbolos diferentes:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$$256 = 2 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

$$12.07 = 1 \times 10^1 + 2 \times 10^0 + 0 \times 10^{-1} + 7 \times 10^{-2}$$

- Na notação binária dispomos de apenas 2 símbolos: 0 e 1. Se utilizarmos 3 desses símbolos (ou seja 3 bits):

número decimal	número em binário	verificando
0	000	$0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 0 + 0 + 0 = 0$
1	001	$0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 0 + 0 + 1 = 1$
2	010	$0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 0 + 2 + 0 = 2$
3	011	$0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 0 + 2 + 1 = 3$
4	100	$1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 4 + 0 + 0 = 4$
5	101	$1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 0 + 1 = 5$
6	110	$1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 4 + 2 + 0 = 6$
7	111	$1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 4 + 2 + 1 = 7$

e a gama é $[0, 2^3 - 1]$ ($8 = 2^3$ valores diferentes).

Notação binária

Gamas: exemplificando com a notação decimal

- ▶ Suponha que apenas dispõe de 4 posições para armazenar números inteiros sem sinal, em notação decimal:

0	0	0	0
...			
9	9	9	9

ou seja $10^4 = 10000$ valores diferentes, e
a gama é $[0, 10^4 - 1]$.

Facilmente se deduz que a gama possível para um número inteiro sem sinal, representado por n dígitos decimais é $[0, 10^n - 1]$.

Notação binária

Gamas: exemplificando com a notação decimal

- ▶ Suponha que apenas dispõe de 4 posições para armazenar números inteiros sem sinal, em notação decimal:

0	0	0	0
...			
9	9	9	9

ou seja $10^4 = 10000$ valores diferentes, e a gama é $[0, 10^4 - 1]$.

Facilmente se deduz que a gama possível para um número inteiro sem sinal, representado por n dígitos decimais é $[0, 10^n - 1]$.

- ▶ Se utilizar a primeira posição para guardar o sinal...

-	9	9	9
...			
-	0	0	0
+	0	0	0
...			
+	9	9	9

Logo a gama vai de -999 a +999, ou seja é $[-10^3 + 1, 10^3 - 1]$ (ou seja $2 \times 10^3 - 1$ valores diferentes, com duas representações para zero).

Notação binária

Gamas em notação binária: números inteiros sem sinal

- ▶ A maior parte dos computadores armazenam números utilizando 16, 32 ou 64 bits.
- ▶ O conjunto de 8 bits tem a designação de octeto ou *byte*.

Notação binária

Gamas em notação binária: números inteiros sem sinal

- ▶ A maior parte dos computadores armazenam números utilizando 16, 32 ou 64 bits.
- ▶ O conjunto de 8 bits tem a designação de octeto ou *byte*.
- ▶ Se desejarmos representar apenas número inteiros positivos, qual a gama que é possível representar utilizando um octeto?

0000 0000 = 0

0000 0001 = 1

0000 0010 = 2

0000 0011 = 3

...

1111 1110 = 254

1111 1111 = 255

Notação binária

Gamas em notação binária: números inteiros sem sinal

- ▶ A maior parte dos computadores armazenam números utilizando 16, 32 ou 64 bits.
- ▶ O conjunto de 8 bits tem a designação de octeto ou *byte*.
- ▶ Se desejarmos representar apenas número inteiros positivos, qual a gama que é possível representar utilizando um octeto?

$$0000\ 0000 = 0$$

$$0000\ 0001 = 1$$

$$0000\ 0010 = 2$$

$$0000\ 0011 = 3$$

...

$$1111\ 1110 = 254$$

$$1111\ 1111 = 255$$

- ▶ A gama é $[0, 255]$ (ou seja $2^8 = 256$ valores diferentes).
A gama para números *inteiros sem sinal*, se armazenados em *n* bits é $[0, 2^n - 1]$.

Notação binária

Transbordo em notação binária: números inteiros sem sinal

- Se somássemos 1 a 9999 (no exemplo anterior com gama de 4 dígitos), obteríamos

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline & 9 & 9 & 9 & 9 \\ \hline + & 0 & 0 & 0 & 1 \\ \hline \end{array} \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

o valor obtido seria 0 e ocorreria um transbordo de uma unidade!

Notação binária

Transbordo em notação binária: números inteiros sem sinal

- ▶ Se somássemos 1 a 9999 (no exemplo anterior com gama de 4 dígitos), obteríamos

	9	9	9	9
+	0	0	0	1
	1	0	0	0

o valor obtido seria 0 e ocorreria um transbordo de uma unidade!

- ▶ Nos computadores, em que é utilizado um número fixo de bits para representar números, este é um problema que pode ocorrer:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ + 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

ou seja não é possível representar (desta forma) números inteiros iguais ou superiores a 255.

Notação binária

Transbordo em notação binária: números inteiros sem sinal

- ▶ Se somássemos 1 a 9999 (no exemplo anterior com gama de 4 dígitos), obteríamos

$$\begin{array}{r} \begin{array}{|c|c|c|c|} \hline & 9 & 9 & 9 & 9 \\ \hline + & 0 & 0 & 0 & 1 \\ \hline \end{array} \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

o valor obtido seria 0 e ocorreria um transbordo de uma unidade!

- ▶ Nos computadores, em que é utilizado um número fixo de bits para representar números, este é um problema que pode ocorrer:

$$\begin{array}{r} \begin{array}{r} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \\ \hline \end{array}$$

ou seja não é possível representar (desta forma) números inteiros iguais ou superiores a 255.

- ▶ Obviamente que se utilizarmos 16 bits em vez de 8 passamos a poder representar números sem sinal desde 0 até $2^{16} - 1 = 65535$.
Mas, dado o número de bits disponível, n , existe sempre um limite dado por $2^n - 1$.

Notação binária

Gamas em notação binária: números inteiros com sinal

- Na representação de **sinal e módulo**, o bit mais significativo (o mais à esquerda) define o sinal e restando 7 bits para representar o módulo do numero – assumindo que $n = 8$.

A gama é $[-127, +127]$ e o **zero tem duas representações**:

$$+0_{10} = 0000 \ 0000 \quad -0_{10} = 1000 \ 0000$$

Notação binária

Gamas em notação binária: números inteiros com sinal

- ▶ Na representação de **sinal e módulo**, o bit mais significativo (o mais à esquerda) define o sinal e restando 7 bits para representar o módulo do numero – assumindo que $n = 8$.

A gama é $[-127, +127]$ e o **zero tem duas representações**:

$$+0_{10} = 0000 \ 0000 \quad -0_{10} = 1000 \ 0000$$

- ▶ A **forma mais utilizada** para representar números inteiros com sinal é **complementos para dois** (*two's complement*):

0_{10}	=	0000 0000
3_{10}	=	0000 0011
-3_{10}	=	1111 1101
127_{10}	=	0111 1111
-127_{10}	=	1000 0001
-128_{10}	=	1000 0000

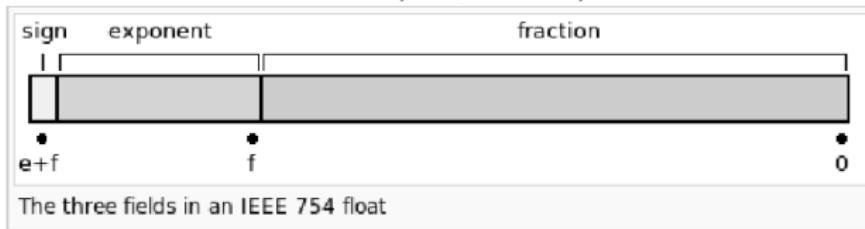
O estudo dessa representação sai fora do âmbito da disciplina mas, além de simplificar aos cálculos, é responsável pelo facto da gama ser $[-128, +127]$, pois só usa **uma única representação para o 0**.

Números em vírgula flutuante

- Não vamos estudar a representação de números reais num computador.

Pretende-se apenas esclarecer porque razão se diz que estes números são armazenados em *vírgula flutuante*.

Se forem utilizados 32 bits, $e + f = 31$, $e = 8$ e $f = 23$:

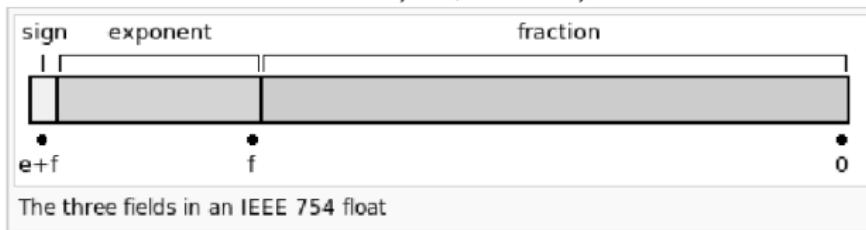


Números em virgula flutuante

- Não vamos estudar a representação de números reais num computador.

Pretende-se apenas esclarecer porque razão se diz que estes números são armazenados em *virgula flutuante*.

Se forem utilizados 32 bits, $e + f = 31$, $e = 8$ e $f = 23$:



- Analogia (muito simplificada) com a representação decimal:

+

+3	1234
----	------

representaria $+0.1234 \times 10^{+3} = 123.4$

E

-2	1234
----	------

representaria $+0.1234 \times 10^{-2} = 0.001234$

Gamas em notação binária

- ▶ Números Inteiros com sinal:

Número de bits	Inteiro com sinal	
	menor	maior
16 (<i>2 bytes</i>)	$-2^{15} = -32768$	$2^{15} - 1 = +32767$
32 (<i>4 bytes</i>)	$-2^{31} = -2147483648$	$2^{31} - 1 = +2147483647$
64 (<i>8 bytes</i>)	-2^{63}	$2^{63} - 1$

- ▶ Números em vírgula flutuante (números reais):

Número de bits	Real		Precisão
	menor	maior	
32 (<i>4 bytes</i>)	-3.4E+38	3.4E+38	6 dígitos
64 (<i>8 bytes</i>)	-1.7E+308	1.7E+308	15 dígitos

<http://www.h-schmidt.net/FloatConverter/IEEE754.html>

Representação Octal e Hexadecimal

Representação octal

- ▶ Escrever números, fazer cálculos com “0s” e “1s” não é natural! Assim foram definidas duas bases, **octal** e **hexadecimal**, que permitem representar números binários de forma mais compacta: em grupos de 3 e 4 bits, respetivamente.

Representação Octal e Hexadecimal

Representação octal

- ▶ Escrever números, fazer cálculos com “0s” e “1s” não é natural! Assim foram definidas duas bases, **octal** e **hexadecimal**, que permitem representar números binários de forma mais compacta: em grupos de 3 e 4 bits, respectivamente.
- ▶ $\underbrace{000}_{0} - \underbrace{111}_{7}$ são oito valores diferentes (símbolos de 0 a 7).

Dec.	Binário	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Exemplo:

$$\overbrace{101}^5 \quad \overbrace{010}^2 \quad \overbrace{011}^3 = 523_8$$

$$\begin{aligned} 523_8 &= 5 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 \\ &= 320_{10} + 16_{10} + 3_{10} \\ &= 339_{10} \end{aligned}$$

Representação Octal e Hexadecimal

Representação Hexadecimal

- ▶ $\underbrace{0000}_{0}$ – $\underbrace{1111}_{15}$: são dezasseis valores diferentes.

Dec.	Binário	Hex.	Dec.	Binário	Hex.
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Nota: A, B, C, D, E, F ou a, b, c, d, e, f

Representação Octal e Hexadecimal

Representação Hexadecimal

- ▶ $\underbrace{0000}_{0} - \underbrace{1111}_{15}$: são dezasseis valores diferentes.

Dec.	Binário	Hex.	Dec.	Binário	Hex.
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Nota: A, B, C, D, E, F ou a, b, c, d, e, f

- ▶ Dado um número binário, para o representar em notação hexadecimal, basta agrupar os bits quatro a quatro.

Exemplificando com um número formado por 16 bits:

$$\underbrace{0010}_2 \underbrace{1101}_D \underbrace{1100}_C \underbrace{0111}_7 = 2DC7_{16} = 2dc7_{16}$$

Representação Octal e Hexadecimal

Representação Hexadecimal: Exemplos

- Um octeto:

$$\begin{array}{rcl} 3_{10} = 3_{16} & 15_{10} = F_{16} \\ \overbrace{0011} \quad \overbrace{1111} & = & 3F_{16} \\ 3F_{16} & = & 3_{10} \times 16^1_{10} + 15_{10} \times 16^0_{10} = 63_{10} \end{array}$$

Representação Octal e Hexadecimal

Representação Hexadecimal: Exemplos

- ▶ Um octeto:

$$\begin{array}{rcl} 3_{10} = 3_{16} & 15_{10} = F_{16} \\ \overbrace{0011} \quad \overbrace{1111} & = & 3F_{16} \\ 3F_{16} & = & 3_{10} \times 16^1_{10} + 15_{10} \times 16^0_{10} = 63_{10} \end{array}$$

- ▶ Os cálculos anteriores são mais compactos do que:

$$0011\ 1111 = (2^5 + 2^4) + (2^3 + 2^2 + 2^1 + 2^0) = 63_{10}$$

Representação Octal e Hexadecimal

Representação Hexadecimal: Exemplos

- ▶ Um octeto:

$$\begin{array}{rcl} 3_{10} = 3_{16} & 15_{10} = F_{16} \\ \overbrace{0011} & \overbrace{1111} & = 3F_{16} \\ 3F_{16} & = & 3_{10} \times 16^1_{10} + 15_{10} \times 16^0_{10} = 63_{10} \end{array}$$

- ▶ Os cálculos anteriores são mais compactos do que:

$$0011\ 1111 = (2^5 + 2^4) + (2^3 + 2^2 + 2^1 + 2^0) = 63_{10}$$

- ▶ Dois octetos:

$$\begin{array}{cccc} \overbrace{1000}^{8_{16}} & \overbrace{1010}^{A_{16}} & \overbrace{1100}^{C_{16}} & \overbrace{0000}^{0_{16}} \\ 8AC0_{16} \end{array}$$

$$8AC0_{16} = 8 \times 16^3_{10} + 10_{10} \times 16^2_{10} + 12_{10} \times 16^1_{10} + 0_{10} \times 16^0_{10} = 35520_{10}.$$

A tabela ASCII – American Standard Code for Information Interchange

Alguns símbolos

A representação de caracteres num computador passa pela associação de cada símbolo do alfabeto a um código numérico.

A tabela ASCII utiliza 7 bits, logo permite representar 128 símbolos (ver no último slide deste capítulo).

- ▶ Os dígitos **0** a **9** têm o código 48_{10} a 57_{10} .
Em ling. C esses caracteres escrevem-se de '**0**' a '**9**'.

A tabela ASCII – American Standard Code for Information Interchange

Alguns símbolos

A representação de caracteres num computador passa pela associação de cada símbolo do alfabeto a um código numérico.

A tabela ASCII utiliza 7 bits, logo permite representar 128 símbolos (ver no último slide deste capítulo).

- ▶ Os dígitos 0 a 9 têm o código 48_{10} a 57_{10} .
Em ling. C esses caracteres escrevem-se de '**0**' a '**9**'.
- ▶ As letras maiúsculas A a Z têm código 65_{10} a 90_{10} . Em ling. C esses caracteres escrevem-se de '**A**' a '**Z**'.

A tabela ASCII – American Standard Code for Information Interchange

Alguns símbolos

A representação de caracteres num computador passa pela associação de cada símbolo do alfabeto a um código numérico.

A tabela ASCII utiliza 7 bits, logo permite representar 128 símbolos (ver no último slide deste capítulo).

- ▶ Os dígitos $[0]$ a $[9]$ têm o código 48_{10} a 57_{10} .
Em ling. C esses caracteres escrevem-se de '**0**' a '**9**'.
- ▶ As letras maiúsculas $[A]$ a $[Z]$ têm código 65_{10} a 90_{10} . Em ling. C esses caracteres escrevem-se de '**A**' a '**Z**'.
- ▶ As letras minúsculas $[a]$ a $[z]$ têm código 97_{10} a 122_{10} . Em ling. C esses caracteres escrevem-se de '**a**' a '**z**'.

A tabela ASCII – American Standard Code for Information Interchange

Alguns símbolos

A representação de caracteres num computador passa pela associação de cada símbolo do alfabeto a um código numérico.

A tabela ASCII utiliza 7 bits, logo permite representar 128 símbolos (ver no último slide deste capítulo).

- ▶ Os dígitos 0 a 9 têm o código 48_{10} a 57_{10} .
Em ling. C esses caracteres escrevem-se de '**0**' a '**9**'.
- ▶ As letras maiúsculas A a Z têm código 65_{10} a 90_{10} . Em ling. C esses caracteres escrevem-se de '**A**' a '**Z**'.
- ▶ As letras minúsculas a a z têm código 97_{10} a 122_{10} . Em ling. C esses caracteres escrevem-se de '**a**' a '**z**'.
- ▶ **Atenção:** os caracteres, quando utilizados em operações aritméticas são convertidos em números com sinal (por exemplo $0+0$ ' vale $+48_{10}$!)

A tabela ASCII <http://www.ascitable.com/>

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	@	Ø	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

Source: www.LookupTables.com

Programação de Computadores

Capítulo 3: Declarações Básicas e Expressões



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a Professora Teresa Martinez do DEEC-UC.

(PdC)

Plano: Declarações básicas e expressões

1. Evolução da linguagem C
2. A estrutura básica de um programa
3. Expressões Simples
4. Variáveis e armazenamento
5. Definição e declaração de variáveis
6. Atribuição; a função printf
7. Inteiros, Reais
8. Leitura formatada (scanf)
9. Divisão
10. Caracteres, getchar versus scanf
11. Leitura em buffer e whitespace
12. Caracteres e inteiros
13. Conversão (cast) de tipos: conversão implícita, conversão explícita
14. Exemplos

Bibliografia:
Capítulos 1 e 2 de [1]

Evolução da linguagem C

- ▶ A linguagem C foi criada em 1972 por Dennis Ritchie na Bell Labs.
- ▶ Em 1978 é publicado por Kernighan & Ritchie o livro “*The C Programming Language*” (K&R C).
- ▶ O *American National Standards Institute* elabora uma norma para a linguagem (finalizada em fins de 1988): ANSI C (também designado por C89).
- ▶ A *International Organization for Standardization* (ISO) - C90 (aceitou o ANSI C, com pequenas modificações).
- ▶ A linguagem C é relativamente estável, surgindo novas normas de 10 em 10 anos aproximadamente:
C99 e C11.

A estrutura básica de um programa

Um programa que não faz nada...

- ▶ Na linguagem C um programa começa sempre pela execução das instruções na função **main**
- ▶ Todo o código a executar é colocado entre chavetas: **{ }**
- ▶ As linhas de código entre um par de chavetas – **{ }** – designa-se por **Bloco**
- ▶ Comentários auxiliares, ignorados pelo compilador têm início em **/*** e terminam em ***/**, não pode haver blocos de comentários encadeados (ou seja um dentro de outro).

```
/* main-vazio.c
   Um programa que não faz nada na norma C99 e C11
 */
int main() /* a função devolve um inteiro ao SO */
{ /* chaveta de inicio de bloco (vazio)           */
    /* faltando a intrução de devolução           */
    /* por omissão retorna 0                      */
} /* chaveta de fim de bloco (vazio)             */
```

Introdução de compilação

Dependo do sistema operativo e do compilador

Para compilar o programa do slide anterior (escrito num editor de texto) usando o compilador da GNU (gcc):

- ▶ `gcc main-vazio.c`
cria o ficheiro executável `a.out`
- ▶ `gcc main-vazio.c -o main-vazio`
cria o ficheiro executável `main-vazio`

Introdução de compilação

Dependo do sistema operativo e do compilador

Para compilar o programa do slide anterior (escrito num editor de texto) usando o compilador da GNU (gcc):

- ▶ `gcc main-vazio.c`
cria o ficheiro executável `a.out`
- ▶ `gcc main-vazio.c -o main-vazio`
cria o ficheiro executável `main-vazio`
- ▶ Em dois passos, permite fazer a verificação sintática antes de criar o executável – particularmente útil quando os programas se estendem por vários ficheiros:
 - ▶ `gcc -c main-vazio.c`
cria ficheiro objecto `main-vazio.o`
 - ▶ `gcc main-vazio.o -o main-vazio`
cria o ficheiro executável (ligando com livrarias / bibliotecas)
- ▶ O `gcc` também poderá ter o nome alternativo `cc`.

A estrutura básica de um programa

Um programa que não faz nada (conclusão)

- ▶ Na referência [1] (as chaves da bibliografia estão nos slides do capítulo 2) encontra um exemplo semelhante a:

```
/* main-vazio-C90.c
   Um programa que não faz nada na norma C90
*/
/* int */ main()
{}
```

Na versão C90 uma função retornando um inteiro poderia omitir o valor de retorno: o tipo *int* estava implícito!

- ▶ A referência [1] (livro de Luís Damas, publicado em janeiro de 1999) segue a norma ANSI C ou C90 e a referência [3], a 2^aEd. do livro de K&R (de 1988) segue o que foi designado por K&R C (muito próximo do C ANSI).
- ▶ As normas, procuraram sempre manter a compatibilidade para trás, gerando avisos quando necessário.
- ▶ Compilando main-vazio-C90.c com a opção **-std=c11** origina o aviso (mas compila com sucesso):

```
main-vazio-C90.c:4:12: warning: return type defaults to 'int' [-Wimplicit-int]
/* int */ main()
^~~~
```

A estrutura básica de um programa

Um programa que não faz nada

- ▶ Compilando o programa anterior `main(){}` com:

`gcc -ansi -Wpedantic main-vazio-C90.c -o main-vazio-C90`
não gera qualquer aviso (-ansi corresponde ao C90 da ISO)

- ▶ Embora a referência [1] diga que os avisos ao compilar o programa que se segue:

```
/* main-void.c
   Um programa que não faz nada
*/
void main()
{}
```

se devem à utilização de um compilador de c++, isso não é estritamente verdade. De facto `void main(){}` não faz parte da norma ANSI, apesar do gcc o compilar com sucesso, mesmo quando lhe é solicitado "strict ANSI mode". Compilando com:

`gcc -ansi -Wpedantic main-void.c -o main-void`
obtemos o aviso:

```
main-void.c:4:7: warning: return type of 'main' is not 'int' [-Wmain]
void main()
^
```

Meu primeiro programa não vazio

Apresenta mensagem

```
/* Um primeiro programa em C
 * Autor: Teresa Gomes           Data: 2019/09/16
 * Objectivo: Demonstracao de um programa simples
 * Utilizacao: Execute que a mensagem aparece
 * */
/* Directiva de inclusão da biblioteca */
#include <stdio.h> /* do standard input/output */

int main()      /* inicio do programa */
{
    /* inicio do bloco de intruções */
    /* Apresenta no ecrã "Bom dia!" e muda de linha */
    printf ("Bom dia!\n");
    return (0); /* Termina, devolve 0 ao Sist. Operat. (SO) */
}                /* fim do bloco de intruções */
```

O caractere `\n` indica que o cursor deve deslocar-se para o início da linha seguinte
(caracteres especiais são representados por `\n`)

Cada instrução dentro de um bloco de {} é terminada por um `;`.

A linguagem C é **Case Sensitive**: `main()` difere de `Main()`

Expressões simples

- Operadores simples:

Operador	Significado
*	multiplicação
/	divisão
+	adição
-	subtração
%	resto da divisão inteira

Expressões simples

- ▶ Operadores simples:

Operador	Significado
*	multiplicação
/	divisão
+	adição
-	subtração
%	resto da divisão inteira

- ▶ Os operadores “*”, “/”, “%” têm precedência sobre a adição e a subtração. Os parêntesis curvos “()” – e apenas estes – podem ser usados para agrupar termos.

Expressões simples

Um exemplo

Um programa inútil, mas que compila sem erro:

```
// main-inutil.c
int main()
{
    (14 % 3) * 5; // Instrução inútil!
    return (0);
}
```

O programa calcula o valor 10 e termina...

Compilando, solicitando o envio de avisos ([-Wall](#)) :

```
gcc -Wall -o main-inutil main-inutil.c
main-inutil.c: In function 'main':
main-inutil.c:4:4: warning: statement with no effect [-Wunused-value]
    (14 % 3) * 5; // Instrução inútil!
    ^
```

De notar a utilização de [comentários no estilo C++](#):

 inicio de comentário que apenas pode estar numa linha

Obs: comentários no estilo C++ não são permitidos no ANSI C, mas são permitidos em versões posteriores do C.

Variáveis e armazenamento

- ▶ A linguagem C permite **armazenar** valores em **variáveis**. Cada variável é identificada por um *nome de variável*.

Variáveis e armazenamento

- ▶ A linguagem C permite **armazenar** valores em **variáveis**. Cada variável é identificada por um *nome de variável*.
- ▶ **O nome de uma variável:** Sequência de letras ou dígitos. O primeiro caractere deve ser uma letra. O caractere `_` é considerado uma letra.

Variáveis e armazenamento

- ▶ A linguagem C permite **armazenar** valores em **variáveis**. Cada variável é identificada por um *nome de variável*.
- ▶ **O nome de uma variável:** Sequência de letras ou dígitos. O primeiro caractere deve ser uma letra. O caractere `_` é considerado uma letra.
- ▶ **A letras minúsculas são diferentes das maiúsculas.** Logo a variável `azul` é diferente de `Azul`.

Estilo: utilize sempre uma letra minúscula para primeira letra de uma variável; não diferencie variáveis com base na capitalização ou não de uma letra.

Variáveis e armazenamento

- ▶ A linguagem C permite **armazenar** valores em **variáveis**. Cada variável é identificada por um *nome de variável*.
- ▶ **O nome de uma variável:** Sequência de letras ou dígitos. O primeiro caractere deve ser uma letra. O caractere `_` é considerado uma letra.
- ▶ **A letras minúsculas são diferentes das maiúsculas.** Logo a variável `azul` é diferente de `Azul`.

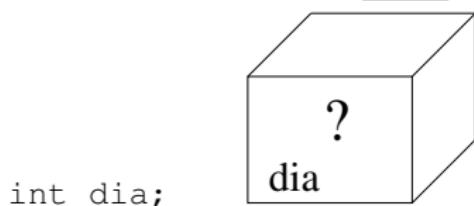
Estilo: utilize sempre uma letra minúscula para primeira letra de uma variável; não diferencie variáveis com base na capitalização ou não de uma letra.

- ▶ Algumas palavras têm um significado especial na linguagem C pelo que **palavras reservadas** ou **palavras chave – keywords** – **não podem ser utilizadas como nome de variáveis**.

Exemplos: **int, float, while, for**.

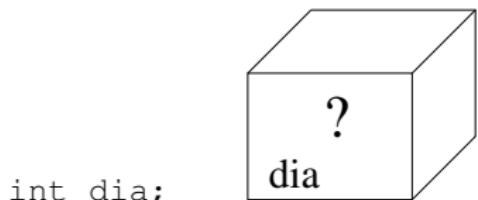
Variáveis e armazenamento

- ▶ A definição de uma variável cria um contentor capaz de conter um valor de um dado tipo (o tipo `int` é utilizado para guardar valores inteiros):

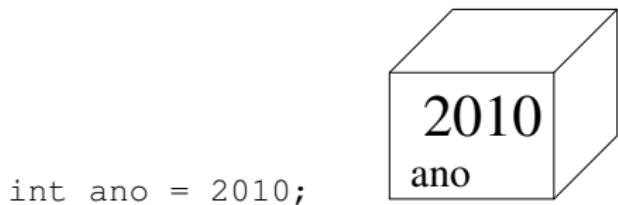


Variáveis e armazenamento

- ▶ A definição de uma variável cria um contentor capaz de conter um valor de um dado tipo (o tipo `int` é utilizado para guardar valores inteiros):



- ▶ A definição de uma variável com inicialização cria um contentor e coloca esse valor no contentor:



Variáveis e armazenamento

- ▶ Uma **variável** tem associado uma **posição de memória** (o contentor) capaz de armazenar um valor de um dado tipo.
- ▶ O **valor por omissão** de uma variável (definida num bloco), antes de lhe ser atribuído um valor, é indefinido (**é lixo**).

Variáveis e armazenamento

- ▶ Uma **variável** tem associado uma **posição de memória** (o contentor) capaz de armazenar um valor de um dado tipo.
- ▶ O **valor por omissão** de uma variável (definida num bloco), antes de lhe ser atribuído um valor, é indefinido (**é lixo**).
- ▶ A **informação contida** nessa posição de memória pode **variar** ao longo da execução do programa

Variáveis e armazenamento

- ▶ Uma **variável** tem associado uma **posição de memória** (o contentor) capaz de armazenar um valor de um dado tipo.
- ▶ O **valor por omissão** de uma variável (definida num bloco), antes de lhe ser atribuído um valor, é indefinido (**é lixo**).
- ▶ A **informação contida** nessa posição de memória pode **variar** ao longo da execução do programa
- ▶ A definição uma ou mais variáveis de um mesmo tipo – (sem qualificadores) – pode fazer-se utilizando a seguinte **sintaxe**:
`// Comentario sobre vari, i=1,2,... n
tipo var1[, var2, ..., varn];`

Em que os [] indicam que os elementos dentro dos [] são opcionais.

Variáveis e armazenamento

Nomes possíveis para uma variável

- ▶ Nomes possíveis para uma variável

```
temperaturaMax // temperatura maxima  
pi             // o numero pi  
area_quadrado // area de um quadrado  
ano1           // primeiro ano
```

Variáveis e armazenamento

Nomes possíveis para uma variável

- ▶ Nomes possíveis para uma variável

```
temperaturaMax // temperatura maxima  
pi             // o numero pi  
area_quadrado // area de um quadrado  
ano1           // primeiro ano
```

- ▶ Uma variável **não pode ter um dos seguintes nomes:**

```
3dias          // comeca por um numero  
$valor         // contem um $  
area circulo   // contem um espaco  
char           // palavra reservada
```

Variáveis e armazenamento

Nomes possíveis para uma variável

- ▶ Alguns ambientes podem considerar necessário estender ou restringir o conjunto de caracteres aceitos no identificador de uma variável.

Variáveis e armazenamento

Nomes possíveis para uma variável

- ▶ Alguns ambientes podem considerar necessário estender ou restringir o conjunto de caracteres aceites no identificador de uma variável.
- ▶ As extensões (por exemplo a possibilidade de utilizar um \$ no nome de uma variável) resultam em **programas não portáveis**.

Definição e declaração de variáveis

Exemplo de alguns tipos

- ▶ Tipos simples com sinal:

```
int k, num;          /* numeros inteiros */
float x, zeta;       /* números reais */
double gama;         /* números reais */
char letra;          /* um caracter (tabela ASCII) */
```

Mais tipos com sinal:

```
short numCurto;      /* numeros inteiros */
long numLongo;        /* números inteiros */
// Ou em alternativa
short int iCurto;    /* numeros inteiros */
long int iLongo;      /* números inteiros */
/* long float não existe */ 
long double alfa, beta; /* números reais */
```

Definição e declaração de variáveis

Exemplo de alguns tipos

- ▶ Tipos simples com sinal:

```
int k, num;          /* numeros inteiros */
float x, zeta;       /* números reais */
double gama;         /* números reais */
char letra;          /* um caracter (tabela ASCII) */
```

Mais tipos com sinal:

```
short numCurto;      /* numeros inteiros */
long numLongo;        /* números inteiros */
// Ou em alternativa
short int iCurto;    /* numeros inteiros */
long int iLongo;      /* números inteiros */
/* long float não existe */
long double alfa, beta; /* números reais */
```

- ▶ Tipos sem sinal:

```
unsigned int ui;      /* 0 e números naturais */
unsigned short int ct; /* 0 e números naturais */
unsigned long int positivo; /* 0 e números naturais */
unsigned char letraX; /* um caracter (ASCII) */
```

- ▶ Mais à frente iremos voltar a referir estes tipos.

Definição e declaração de variáveis

- Uma variável só pode ser utilizada depois de declarada.

A declaração de uma variável tem três objetivos:

- Define o nome da variável
- Define o tipo da variável (inteiro, real, caractere, etc), que indica ao compilador o espaço que esta ocupa e de que forma deve interpretar o seu valor
- Dá ao programador uma descrição da variável (comentários!)

Definição e declaração de variáveis

- ▶ Uma variável só pode ser utilizada depois de declarada.

A declaração de uma variável tem três objetivos:

- ▶ Define o nome da variável
- ▶ Define o tipo da variável (inteiro, real, caractere, etc), que indica ao compilador o espaço que esta ocupa e de que forma deve interpretar o seu valor
- ▶ Dá ao programador uma descrição da variável (comentários!)

- ▶ Exemplo de uma definição, que também é uma declaração:

```
int custo;      // custo total
```

Definição e declaração de variáveis

- ▶ Uma variável só pode ser utilizada depois de declarada.
- ▶ A declaração de uma variável tem três objetivos:
 - ▶ Define o nome da variável
 - ▶ Define o tipo da variável (inteiro, real, caractere, etc), que indica ao compilador o espaço que esta ocupa e de que forma deve interpretar o seu valor
 - ▶ Dá ao programador uma descrição da variável (comentários!)
- ▶ Exemplo de uma definição, que também é uma declaração:

```
int custo;      // custo total
```
- ▶ O exemplo anterior além de declarar custo é uma definição porque define uma entidade associada ao nome custo.

Neste caso essa entidade é um espaço de memória (num dado endereço) capaz de armazenar um valor inteiro. Quando dermos funções será mais clara a diferença entre declaração e definição.

Atribuição

- ▶ Valores são atribuídos a variáveis através de instruções de atribuição.

Em primeiro lugar a variável tem de estar declarada:

```
int custo;      // custo total
```

em seguida a variável `custo` já pode ser utilizada numa instrução de atribuição:

```
custo = 100 * 20; // custo <- 2000
```

A variável do lado esquerdo do sinal de igual (`=`) toma o valor da expressão `100 * 20` que se encontra do lado direito.

Atribuição

- ▶ Valores são atribuídos a variáveis através de **instruções de atribuição**.

Em primeiro lugar a variável tem de estar declarada:

```
int custo; // custo total
```

em seguida a variável `custo` já pode ser utilizada numa instrução de atribuição:

```
custo = 100 * 20; // custo <- 2000
```

A variável do lado esquerdo do sinal de igual (`=`) toma o valor da expressão `100 * 20` que se encontra do lado direito.

- ▶ Quando uma variável é declarada e definida, o C reserva espaço de armazenamento para essa variável e coloca nesse espaço um valor indefinido!

O símbolo `=` é usado para atribuição, não para teste de igualdade!

Atribuição

Sintaxe da atribuição

variavel = expressao; /* variavel toma o valor da expressao */

Exemplo (troço de um programa):

```
int k;    // define k
k = 100; // atribui 100 a k (ou seja k <-- 100)
```

Declaração com inicialização

Uma variável pode ser inicializada quando se faz a sua declaração:

Exemplo (troço de um programa):

```
int num = 100;
```

É também possível **atribuir o mesmo valor a várias variáveis** (troço de um programa):

```
int a, b, c, d;
a = b = c = d = 120; // d<--120; c<--d; b<--c; a<--b;
```

A função *printf*

- A função *printf* é utilizada para enviar texto (entre aspas) para a consola.

A função *printf*

- ▶ A função *printf* é utilizada para enviar texto (entre aspas) para a consola.
- ▶ A função *printf* também pode ser utilizada para apresentar um caractere, uma sequência de caracteres, números ou resultados do cálculo de expressões numéricas.

```
#include <cstdio> // bib. C de canais entrada/saída

int main()
{
    // Apresenta no ecrã linhas de texto
    printf("Usando printf\npara apresentar resultados:\n");
    printf("3*5 vale %d \n", 3*5);
    return (0); // termina e devolve ao SO o valor 0 (ok!)
}
```

Surgirá na última linha da consola:

```
3*5 vale 15
```

Imprimir uma expressão do tipo inteiro

O formato de escrita de um inteiro na função *printf* é **%d**

Atribuição

```
1 #include <stdio.h>
2 int main()
3 {
4     int dado;      // variavel usada para exemplificar
5
6     dado = 2 * 4;
7     printf("O triplo de %d vale %d\n", dado, 3*dado);
8     printf("e dado continua a valer %d\n", dado);
9     dado = dado + 1; // dado <- dado + 1
10    printf("Mas agora vale %d\n", dado);
11    return 0; // OU return (0)
12 }
```

- ▶ A instrução na linha 3 declarou a variável `dado` e reservou uma espaço de memória para armazenamento do seu valor;

Linha 3: `dado`

?

Linha 6: `dado`

8

Linha 10: `dado`

9

Variáveis locais a um bloco

Uma variável local a um bloco, quando definida, fica sempre com um valor indeterminado.

Atribuição

O programa anterior produziria a seguinte saída na consola:

```
O triplo de 8 vale 24
e dado continua a valer 8
Mas agora vale 9
```

Inteiros

- ▶ Um dos tipos de variáveis suportados pelo C é o tipo inteiro, ou seja números sem parte fracionária e sem ponto decimal.

A forma geral da declaração de um inteiro é:

int nome_da_variavel; // comentário

Inteiros

- ▶ Um dos tipos de variáveis suportados pelo C é o tipo inteiro, ou seja números sem parte fracionária e sem ponto decimal.

A forma geral da declaração de um inteiro é:

int nome_da_variavel; // comentário

- ▶ Os computadores usam 0s e 1s (como já foi referido) para armazenar e manipular informação (esses números binários, por omissão, são convertidos em números decimais quando são enviados para o ecrã).

Inteiros

- ▶ Um dos tipos de variáveis suportados pelo C é o tipo inteiro, ou seja números sem parte fracionária e sem ponto decimal.

A forma geral da declaração de um inteiro é:

int nome_da_variavel; // comentário

- ▶ Os computadores usam 0s e 1s (como já foi referido) para armazenar e manipular informação (esses números binários, por omissão, são convertidos em números decimais quando são enviados para o ecrã).
- ▶ O número de bits utilizados para armazenar um inteiro depende do computador.

Hoje em dia a maior parte dos computadores utiliza 64 bits, logo a gama de valores poderá ir de -2^{63} a $2^{63} - 1$.

Inteiros e suas variações

Exemplo usando a função sizeof: short, int, long e unsigned

```
//int-short-long.c
#include <stdio.h>
int main()
{
    int i = 713;
    short /* int */ s = 4;
    long /* int */ k = 31232434;

    unsigned int ui = 15;
    unsigned short int us = 12;
    unsigned long int uk = 156;

    printf("i=%d\ts=%hd\tuk=%ld\n", i, s, k );
    printf("ui=%u\tus=%hu\tuk=%lu\n\n", ui, us, uk );

    printf("sizeof(int) == %lu\n", sizeof(int));
    printf("sizeof(short int) == %lu\n", sizeof(short int));
    printf("sizeof(long int) == %lu\n\n", sizeof(long int));
    // Omitindo int
    printf("sizeof(unsigned ) == %lu\n", sizeof(unsigned));
    printf("sizeof(unsigned short) == %lu\n", sizeof(unsigned short));
    printf("sizeof(unsigned long) == %lu\n", sizeof(unsigned long));

    return 0;
}
```

Inteiros e suas variações

Exemplo usando a função `sizeof`: short, int, long e unsigned

A função `sizeof` devolve o número de *bytes* que cada tipo ocupa:

```
i=713      s=4      uk=31232434
ui=15     us=12     uk=156

sizeof(int) == 4
sizeof(short int) == 2
sizeof(long int) == 8

sizeof(unsigned ) == 4
sizeof(unsigned short) == 2
sizeof(unsigned long) == 8
```

Inteiros e suas variações

Exemplo usando a função `sizeof`: `short`, `int`, `long` e `unsigned`

A função `sizeof` devolve o número de *bytes* que cada tipo ocupa:

```
i=713      s=4      uk=31232434
ui=15     us=12     uk=156

sizeof(int) == 4
sizeof(short int) == 2
sizeof(long int) == 8

sizeof(unsigned ) == 4
sizeof(unsigned short) == 2
sizeof(unsigned long) == 8
```

Formato de leitura e escrita de `short int` ou `long int`

O formato de leitura e escrita de variáveis inteiras `short` ou `long` deve ser precedido por `h` e `l`, respectivamente.

Inteiros e suas variações

Exemplo usando a função `sizeof`: `short`, `int`, `long` e `unsigned`

A função `sizeof` devolve o número de *bytes* que cada tipo ocupa:

```
i=713      s=4      uk=31232434
ui=15     us=12     uk=156

sizeof(int) == 4
sizeof(short int) == 2
sizeof(long int) == 8

sizeof(unsigned ) == 4
sizeof(unsigned short) == 2
sizeof(unsigned long) == 8
```

Formato de leitura e escrita de `short int` ou `long int`

O formato de leitura e escrita de variáveis inteiras `short` ou `long` deve ser precedido por `h` e `l`, respectivamente.

`signed` versus `unsigned`

Por omissão todos os tipos inteiros são `signed`.

Se for desejado utilizar apenas valores positivos (incluindo o 0), é preciso usar o especificador `unsigned`.

Reais: Números em vírgula flutuante

- ▶ O C utiliza o ponto decimal para distinguir entre números inteiros e números em vírgula flutuante: 5 é um inteiro mas 5.0 já não. O zero em vírgula flutuante deve ser escrito 0.0
- ▶ Por omissão **valores constantes em vírgula flutuante** são **double**.

Reais: Números em vírgula flutuante

- ▶ O C utiliza o ponto decimal para distinguir entre números inteiros e números em vírgula flutuante: 5 é um inteiro mas 5.0 já não. O zero em vírgula flutuante deve ser escrito 0.0
- ▶ Por omissão valores constantes em vírgula flutuante são **double**.
- ▶ Exemplos de números em vírgula flutuante: 3.14, -9.95 e 0.5 (pode escrever-se .5 em vez e 0.5, mas...).

Um número em vírgula flutuante pode ser dado indicando o expoente de base 10: 2.5e12 é uma forma abreviada de escrever 2.5×10^{12} .

Reais: Números em vírgula flutuante

- ▶ O C utiliza o ponto decimal para distinguir entre números inteiros e números em vírgula flutuante: 5 é um inteiro mas 5.0 já não. O zero em vírgula flutuante deve ser escrito 0.0
- ▶ Por omissão valores constantes em vírgula flutuante são **double**.
- ▶ Exemplos de números em vírgula flutuante: 3.14, -9.95 e 0.5 (pode escrever-se .5 em vez e 0.5, mas...).

Um número em vírgula flutuante pode ser dado indicando o expoente de base 10: 2.5e12 é uma forma abreviada de escrever 2.5×10^{12} .

- ▶ A forma geral da declaração de um número em vírgula flutuante:

```
float nomef;      // comentário  
double nomed;    // comentário
```

Reais: Números em vírgula flutuante

- ▶ O C utiliza o ponto decimal para distinguir entre números inteiros e números em vírgula flutuante: 5 é um inteiro mas 5.0 já não. O zero em vírgula flutuante deve ser escrito 0.0
- ▶ Por omissão valores constantes em vírgula flutuante são **double**.
- ▶ Exemplos de números em vírgula flutuante: 3.14, -9.95 e 0.5 (pode escrever-se .5 em vez e 0.5, mas...).

Um número em vírgula flutuante pode ser dado indicando o expoente de base 10: 2.5e12 é uma forma abreviada de escrever 2.5×10^{12} .

- ▶ A forma geral da declaração de um número em vírgula flutuante:

```
float nomef;      // comentário  
double nomed;    // comentário
```

- ▶ Os números em vírgula flutuante podem ser mostrados em vários formatos:

```
printf("Vale %f %g %e \n", 2.0/7.0, 1.0/3.0, 4.0/7.0);  
printf("Vale %F %G %E \n", 2.0/7.0, 1.0/3.0, 4.0/7.0);
```

Imprimir uma expressão do tipo real

O formato de escrita de um real (*float*) na função printf é **%f**, **%e** ou **%g**; no C99 o formato de um *double* deve ser começar por **l** (**%lf**, **%le** ou **%lg**).

Reais: Números em vírgula flutuante

Ilustrando problemas numéricos 1/2

```
//float-precision.c
#include <stdio.h>
int main()
{
    // Ilustrando problemas numéricos devido à precisão finita da
    // representação de números em vírgula flutuante num computador
    float x = 0.1; // 0.1 é uma dízima infinita na base 2

    x = x + 0.1; // a adição sucessiva de 0.1 até obter 0.1*10
    x = x + 0.1; x = x + 0.1 ;
    x = x + 0.1; x = x + 0.1 ;
    x = x + 0.1; x = x + 0.1 ;
    x = x + 0.1; x = x + 0.1 ;

    printf("Com seis casas decimais\nx = %f\n\n", x);
    printf("Com doze casas decimais\n");
    printf("soma sucessiva de 10 parcelas 0.1, x = %.12e\n", x);
    printf("        multiplicando x por 10, 10*x = %.12e\n", x*10);
}
```

Fará surgir no ecrã 1.000000 e também 1.000000000000?

Sim para o primeiro (é o valor arredondado), mas não para o segundo!

Precisão de números em *float* é 6 dígitos na base 10.

Reais: Números em vírgula flutuante

Ilustrando problemas numéricos 2/2

O programa anterior fará surgir no ecrã o texto:

```
Com seis casas decimais  
x = 1.000000
```

```
Com doze casas decimais  
soma sucessiva de 10 parcelas 0.1, x = 1.000000119209e+00  
multiplicando x por 10, 10*x = 1.000000095367e+01
```

Pode verificar-se que 0.1 é uma dízima infinita usando:

<http://www.h-schmidt.net/FloatConverter/IEEE754.html>

A precisão do resultado depende da precisão das variáveis

Se tivéssemos utilizado *double* em vez de *float*, o resultado seria o mesmo?

Leitura formatada (*scanf*)

- ▶ A função permite enviar para a consola texto e valores de variáveis sobre a forma de texto.
- ▶ A função correspondente para leitura de valores é *scanf*.

```
// scanf-int-float.c
#include <stdio.h>
int main()
{
    int num; // Define (e declara) num
    float x; // Define (e declara) x
    printf("De-me dois números, um inteiro seguido de um real:");
    scanf("%d%f", &num, &x); // string inicial dá o formato
    printf("Os números dados foram %d e %f\n", num, x);
    return (0);
}
```

Leitura formatada (*scanf*)

- ▶ A função permite enviar para a consola texto e valores de variáveis sobre a forma de texto.
- ▶ A função correspondente para leitura de valores é *scanf*.

```
// scanf-int-float.c
#include <stdio.h>
int main()
{
    int num; // Define (e declara) num
    float x; // Define (e declara) x
    printf("De-me dois números, um inteiro seguido de um real:");
    scanf("%d%f", &num, &x); // string inicial dá o formato
    printf("Os números dados foram %d e %f\n", num, x);
    return (0);
}
```

Função *scanf*

Na função *scanf*, depois de especificado o formato de leitura de todos os argumentos, estes devem ser colocados pela mesma ordem, precedidos de **&**. A string não deve conter outros caracteres além dos necessários para os formatos de leitura.

Resumo de alguns formatos de escrita (*printf*)

Tipo	Formato (especificador)	Descrição
char	%c	Um único caractere
unsigned char	%c	Um único caractere
int	%d ou %i	Um inteiro base 10
int	%o	Um inteiro (base 8)
int	%x ou %X	Um inteiro (base 16)
short int	%hd ou %hi	Um short inteiro (base 10)
long int	%ld ou %li	Um long inteiro (base 10)
unsigned int	%u	Um inteiro positivo (base 10)
unsigned short int	%hu	Um short inteiro positivo (base 10)
unsigned long int	%lu	Um long inteiro positivo (base 10)
float	%f ou %e ou %g %F ou %E ou %G	Um número real precisão simples
double	%lf ou %le ou %lg %lF ou %lE ou %lG	Um número real precisão dupla

Resumo de alguns formatos de leitura (`scanf`)

Tipo	Formato (especificador)	Descrição
char	%c	Um único caractere
unsigned char	%c	Um único caractere
int	%d	Um inteiro (base 10)
int	%i	Um inteiro (base 10, 8 ou 16)
int	%o	Um inteiro (base 8)
int	%x ou %X	Um inteiro (base 16)
short int	%hd	Um short inteiro (base 10)
short int	%hi	Um short inteiro (base 10, 8 ou 16)
long int	%ld	Um long inteiro (base 10)
long int	%li	Um long inteiro (base 10, 8 ou 16)
unsigned int	%u	Um inteiro positivo (base 10)
unsigned short int	%hu	Um short inteiro positivo na base 10
unsigned long int	%lu	Um long inteiro positivo na base 10
float	%f ou %e ou %g %F ou %E ou %G	Um número real precisão simples
double	%lf ou %le ou %lg %lf ou %le ou %lg	Um número real precisão dupla

Os

especificadores de conversão que não consomem (ou seja que não ignoram) nenhum espaço em branco inicial, como %c, podem fazer isso usando um caractere de espaço em branco antes da string de formato:

```
scanf(" %c", &c);
```

stdin, stdout, stderr

- ▶ Ao escrevermos mensagens na consola estamos a escrever no fluxo (*stream*) de saída *stdout*
- ▶ Ao ler do teclado estamos a receber informação no fluxo (*stream*) de entrada *stdin*
- ▶ As mensagens de erro são enviadas para o fluxo (*stream*) de erros *stderr*

Nota: A referência [2] traduz *stream* como fluxo. Nesta unidade curricular o termo *canal* poderá ser como tradução de *stream*.

Divisão inteira versus divisão em vírgula flutuante

- ▶ O operador divisor (/) permite fazer a divisão no conjunto \mathbb{Z} , quando o dividendo e o divisor são inteiros e no conjunto \mathbb{R} quando um deles é real.

Divisão inteira versus divisão em vírgula flutuante

- ▶ O operador divisor (/) permite fazer a divisão no conjunto \mathbb{Z} , quando o dividendo e o divisor são inteiros e no conjunto \mathbb{R} quando um deles é real.
- ▶ Relembrando a divisão em \mathbb{Z} : a divisão inteira de 123 por 10 resulta num quociente de valor 12 e resto 3.

Divisão inteira versus divisão em vírgula flutuante

- ▶ O operador divisor (/) permite fazer a divisão no conjunto \mathbb{Z} , quando o dividendo e o divisor são inteiros e no conjunto \mathbb{R} quando um deles é real.
- ▶ Relembrando a divisão em \mathbb{Z} : a divisão inteira de 123 por 10 resulta num quociente de valor 12 e resto 3.
- ▶ Assim em C:

Expressão	Resultado	Tipo do resultado
$123/10$	12	Inteiro
$123.0/10.0$	12.3	Vírgula flutuante
$123.0/10$	12.3	Vírgula flutuante (A evitar!)
$123/10.0$	12.3	Vírgula flutuante (A evitar!)

Divisão inteira versus divisão em vírgula flutuante

Exemplos

```
#include <stdio.h>

int main()
{
    int i;           // variavel inteira p/ exemplificar
    double x;        // variavel em virg. flutuante p/ exemplificar

    x = 1.0 / 5.0;   // x passa a valer 0.2
    i = 1 / 5;       // i passa a valer 0
    printf("\nx = %lf  i = %d\n", x, i);

    x = 10.0 / 4.0;  // x passa a valer 2.5
    printf("\nx = %lf",x);

    // Evitar!
    i = x;           // i passa a valer 2
    x = 10 / 3;      // x passa a valer 3.0 (e nao 3.3(3))
    printf("\nx = %lf  i = %d\n", x, i);
    return (0) ;
}
```

Divisão inteira versus divisão em vírgula flutuante

Exemplos

A saída do programa anterior seria:

```
x = 0.200000 i = 0
```

```
x = 2.500000
```

```
x = 3.000000 i = 2
```

Caracteres

- ▶ O tipo *char* representa um único caractere.
- ▶ O espaço necessário para armazenar *char* é um *byte*.
- ▶ A forma geral desta declaração é:

```
char nome; // comentário
```

Caracteres

- ▶ O tipo *char* representa um único caractere.
- ▶ O espaço necessário para armazenar *char* é um *byte*.

- ▶ A forma geral desta declaração é:

```
char nome; // comentário
```

- ▶ Os caracteres são dados entre **aspas simples** ou **plicas** e não entre aspas duplas: 'a', 'A', '#', '?' ou '8'.

As aspas duplas servem para representar *strings*, cadeias de caracteres (como as que surgem em *printf* e *scanf*).

Caracteres

- ▶ O tipo *char* representa um único caractere.
- ▶ O espaço necessário para armazenar *char* é um *byte*.
- ▶ A forma geral desta declaração é:
`char nome; // comentário`
- ▶ Os caracteres são dados entre **aspas simples** ou **plicas** e não entre aspas duplas: 'a', 'A', '#', '?' ou '8'.
As aspas duplas servem para representar *strings*, cadeias de caracteres (como as que surgem em *printf* e *scanf*).
- ▶ Caracteres especiais (de difícil representação) são precedidos do caractere \ (*escape character*), e dados entre aspas simples.
O caractere '\n' permite ir para a linha seguinte e e '\t' permite avançar para a posição seguinte de tabulação.

Caracteres

- ▶ O tipo *char* representa um único caractere.
- ▶ O espaço necessário para armazenar *char* é um *byte*.
- ▶ A forma geral desta declaração é:
`char nome; // comentário`
- ▶ Os caracteres são dados entre **aspas simples** ou **plicas** e não entre aspas duplas: 'a', 'A', '#', '?' ou '8'.
As aspas duplas servem para representar *strings*, cadeias de caracteres (como as que surgem em *printf* e *scanf*).
- ▶ Caracteres especiais (de difícil representação) são precedidos do caractere \ (*escape character*), e dados entre aspas simples.
O caractere '\n' permite ir para a linha seguinte e e '\t' permite avançar para a posição seguinte de tabulação.

Formato de leitura e escrita de char

O formato de escrita e de leitura de um caractere nas funções *printf* e *scanf*, respetivamente, é **%c** em ambos os casos.

Declarações básicas e expressões

Tabela de Caracteres Especiais

\b	<i>backspace</i>	Mover o cursor uma vez para trás
\f	<i>form feed</i>	Ir para o início de uma nova página
\n	<i>new line</i>	Ir para a linha seguinte
\r	<i>return</i>	Ir para o início da linha corrente
\t	<i>tab</i>	Ir para a posição de tabulação seguinte
\'	<i>apóstrofe</i>	O símbolo '
\"	<i>aspas</i>	O símbolo "
\\"	<i>barra à esquerda</i>	O símbolo \
\a	<i>bell</i>	Emite um som (apito)
\nnn	<i>O caractere cujo código em octal é nnn</i>	Apresenta esse caractere
\xNN	<i>O caractere cujo código em hexadecimal é NN</i>	Apresenta esse caractere

Para obter um % deve escrever **%%.**

Caracteres

Exemplificando

O programa

```
#include <stdio.h>

int main()
{
    char letral; // primeiro caractere
    char letra2; // segundo caractere
    char letra3; // terceiro caractere

    letral = 'a';
    letra2 = 'b';
    letra3 = 'c';

    printf("%c%c%c", letral, letra2, letra3);
    printf(" invertendo ");
    printf("%c%c%c\n", letra3, letra2, letral);
    return(0);
}
```

Produciria a saída:

```
abc invertendo cba
```

getchar versus scanf

- A leitura de um char pode ser feita usando *scanf*

```
// scanf-char.c
#include <stdio.h>
int main()
{
    char ch; // Define (e declara) ch
    printf("De-me um caractere: ");
    scanf("%c", &ch); // string inicial dá o formato
    printf("O carácter digitado foi%c\n", ch);
    return (0);
}
```

getchar versus scanf

- A leitura de um char pode ser feita usando *scanf*

```
// scanf-char.c
#include <stdio.h>
int main()
{
    char ch; // Define (e declara) ch
    printf("De-me um caractere: ");
    scanf("%c", &ch); // string inicial dá o formato
    printf("O carácter digitado foi %c\n", ch);
    return (0);
}
```

- Ou usando a função *getchar* específica para ler um caractere:

```
// le_getchar.c
#include <stdio.h>
int main()
{
    char ch; // Define (e declara) ch
    printf("De-me um caractere: ");
    ch = getchar(); //ch <- getchar() sem parâmetros ou formato
    printf("O caractere digitado foi %c\n", ch);
    return (0);
}
```

Leitura em buffer e whitespace

Exemplo: Leitura de dois *char* que não funciona como desejado...

- Se tentar ler 2 char como se indica usando *scanf*:

```
// scanf-charx2.c
#include <stdio.h>
int main()
{
    char ch1, ch2; // Define (e declara) ch1 e ch2
    printf("De-me um caractere: ");
    scanf("%c", &ch1); // string inicial dá o formato
    printf("De-me outro caractere: ");
    scanf("%c", &ch2); // string inicial dá o formato
    printf("Digitou '%c' e '%c'\n", ch1, ch2);
    return (0);
}
```

O resultado será:

```
De-me um caractere: r
De-me outro caractere: Digitou 'r' e '
'
```

Leitura em *buffer* e *whitespace*

Exemplo: Leitura de dois *char* que não funciona como desejado...

- Se tentar ler 2 char como se indica usando *scanf*:

```
// scanf-charx2.c
#include <stdio.h>
int main()
{
    char ch1, ch2; // Define (e declara) ch1 e ch2
    printf("De-me um caractere: ");
    scanf("%c", &ch1); // string inicial dá o formato
    printf("De-me outro caractere: ");
    scanf("%c", &ch2); // string inicial dá o formato
    printf("Digitou '%c' e '%c'\n", ch1, ch2);
    return (0);
}
```

O resultado será:

```
De-me um caractere: r
De-me outro caractere: Digitou 'r' e '
'
```

- A leitura de valores usa o *buffer* do teclado.
- O que foi digitado no exemplo anterior foi:
[r] seguido se [ENTER], ou seja [r] ficou armazenado em ch1 e
[\n] em ch2 e o programa terminou.

Leitura em *buffer* e *whitespace*

Leitura de dois *char* ignorando *whitespace*

- ▶ Quando lemos **números** os *whitespace*, ou seja o espaço, o tab, e o <ENTER>, são separadores e são consumidos/ignorados.
- ▶ Quando lemos um **caractere** isso já não acontece.
- ▶ Para que o programa do slide anterior funcione como desejamos:
 - ▶ No formato do segundo *scanf* escrever "%c" em que _ representa um espaço - ver exemplo que se segue.

```
// scanf-charx2-ok.c
#include <stdio.h>
int main()
{
    char ch1, ch2; // Define (e declara) ch1 e ch2
    printf("De-me um caractere: ");
    scanf("%c", &ch1); // string inicial dá o formato: "%c"
    printf("De-me outro caractere: ");
    scanf(" %c", &ch2); // string inicial dá o formato: " %c"
    printf("Digitou '%c' e '%c'\n", ch1, ch2);
    return (0);
}
```

Caracteres e inteiros

- ▶ Como já foi referido, os caracteres em C são guardados como valores inteiros num único *byte*. Assim as operações sobre *int* podem ser também ser feitas sobre *char*.
- ▶ Pode recordar a tabela ASCII nos slides do Cap. 2 ou em <https://pt.wikipedia.org/wiki/ASCII>

```
1 // ascii-char-A.c
2 #include <stdio.h>
3 int main()
4 { // A letra 'A'
5     char ch1, ch2, ch3, ch4; // para guardar a letra 'A'
6     ch1 = 'A';             // formato tradicional, o mais natural
7     ch2 = 65;              // letra cujo código é 65 na tabela ASCII (ZZZ)
8     ch3 = '\101';          // 101 na base 8 vale 65 na base 10
9     ch4 = '\x41';          // 41 na base 16 vale 65 na base 10
10    printf("A letra '%c', '%c', '%c' e '%c'\n", ch1, ch2, ch3, ch4);
11    return (0);
12 }
```

Produziria a saída:

```
A letra 'A', 'A', 'A' e 'A'
```

Caracteres e inteiros

Apresentando o código numérico de uma letra: conversão implícita

O programa:

```
// ascii-char2int-1.c
#include <stdio.h>
int main()
{ // char para int com conversão implícita
    char ch; // para guardar uma letra
    printf("De-me uma letra:\n");
    scanf("%c", &ch); // não esquecer o símbolo &
    printf("O catácter '%c' tem o código ASCII %d\n", ch, ch);
    return (0);
}
```

Produziria a saída, caso o utilizador digitasse `a<ENTER>`:

```
De-me uma letra:
a
O catácter 'a' tem o código ASCII 97
```

- ▶ Este programa funciona, mas não está escrito da melhor forma!
- ▶ A função esperava encontrar como argumento final uma expressão do tipo *int* e não do tipo *char*. A expressão é **convertida de forma implícita** de *char* para *int* o valor correspondente é apresentado como inteiro.
- ▶ No slide anterior – linha 7 – foi realizada a operação inversa: o **byte menos significativo** do inteiro 65 foi armazenado na variável *ch2*.

Conversão (*cast*) de tipos

Conversão Implicita

Conversão implícita

Pode dizer-se, de forma simplificada, que sempre que numa expressão estão presentes valores ou variáveis de tipo diferentes o compilador de C faz uma conversão (*cast*) **implícita** para o tipo de maior capacidade.

Se uma expressão contém várias expressões que têm de ser calculadas, esta regra aplica-se a cada uma delas, à medida que o cálculo progride.

Conversão (*cast*) de tipos

Conversão Implicita

Conversão implícita

Pode dizer-se, de forma simplificada, que sempre que numa expressão estão presentes valores ou variáveis de tipo diferentes o compilador de C faz uma conversão (*cast*) **implícita** para o tipo de maior capacidade.

Se uma expressão contém várias expressões que têm de ser calculadas, esta regra aplica-se a cada uma delas, à medida que o cálculo progride.

Conversão implícita e atribuição

Se a expressão do lado direito de uma atribuição é de tipo diferente da variável do lado esquerdo, o compilador de C calcula o valor dessa expressão e seguidamente faz uma conversão (*cast*) implícita para o tipo da variável que armazenará o valor resultante.

Conversão (*cast*) de tipos

Conversão Cast Implícita usando *int* e *double*

```
//conversao-implicita-int-double.c
#include <stdio.h>
int main()
{
    printf("\nCalcula 5 / 2 e obtem 2;");
    printf("\nfaz cast implicito para double e calcula 2.0 * 3.0 ");
    printf("\n(5 / 2) * 3.0 = %f", (5 / 2) * 3.0);

    printf("\n\nFaz cast implicito de 7 para 7.0, ");
    printf("calcula 7.0 / 2.0 e obtem 3.5; ");
    printf("\nfaz cast implicito de 4 para 4.0 e adiciona-lhe 3.5");
    printf("\n7 / 2.0 + 4 = %f\n", 7 / 2.0 + 4);
    return(0);
}
```

Produziria a saída:

```
Calcula 5 / 2 e obtem 2;
faz cast implicito para double e calcula 2.0 * 3.0
(5 / 2) * 3.0 = 6.000000

Faz cast implicito de 7 para 7.0, calcula 7.0 / 2.0 e obtem 3.5;
faz cast implicito de 4 para 4.0 e adiciona-lhe 3.5
7 / 2.0 + 4 = 7.500000
```

Conversão (*cast*) de tipos

Conversão Cast Implicita usando *int* e *char*

```
//conversao-implicita-char-int.c
#include <stdio.h>
int main()
{
    char letral, letra2; // primeiro e segundo caracter
    int i;              // codigo ASCII de letral
    int j;              // codigo ASCII de letra2

    letral = 'd'; i = letral; // letral <- 'd' e i <- 100
    letra2 = i+1; j = letra2; // letra2 <- 'e' e j <- 101
    printf("\na) letral = |%c", letral);      // primeira linha
    printf("| e tem o codigo ASCII %d", i);
    printf("\nb) letra2 = |%c", letra2);      // segunda linha
    printf("| e tem o codigo ASCII %d", j);
    letra2 = letral + 2 ; // letra2 <- 'f'
    j = letra2 ;          // j <- 102
    printf("\nc) letra2 = |%c", letra2);      // terceira linha
    printf("| e tem o codigo ASCII %d\n", j);
    return(0);
}
```

Produciria a saída:

- a) letral = |d| e tem o codigo ASCII 100
- b) letra2 = |e| e tem o codigo ASCII 101
- c) letra2 = |f| e tem o codigo ASCII 102

Conversão (*cast*) de tipos

Conversão Cast Implicita – Mais um Exemplo

```
//conversao-implicita-char-int-double.c
#include <stdio.h>

int main()
{
    char letra; // um caracter
    int i; // codigo ASCII de letra
    double x; // Número real

    letra = 'b'; // letra <- 'b', cujo codigo ASCII vale 98
    i = letra; // i <- 98 (de char para int)
    i = i - 1; // i <- 97
    letra = i; // letra <- 'a' (de int para char)

    printf("\nletra = |%c", letra);
    printf("| e tem o codigo ASCII %d", i);

    // Cast implicito de int para double
    x = i + 0.9; // x <- 97.0 + 0.9
    printf("\nx = %f\n", x);

    x = i/2; // calcula 97/2 = 48; x <- 48.0 (de int para double)
    printf("\nx = %f\n", x);
    return(0);
}
```

Produziria a saída:

```
letra = |a| e tem o codigo ASCII 97
x = 97.900000
x = 48.000000
```

Conversão (*cast*) de tipos

Conversão Cast Implicita – Problemas

Conversões a evitar na leitura (compila com aviso!)

A leitura variáveis com a função *scanf* utilizando formatos não adequados aos tipos das variáveis que são passadas como argumentos poderá resultar em erros difíceis de detectar.

Conversão (*cast*) de tipos

Conversão Cast Implicita – Problemas

Conversões a evitar na leitura (compila com aviso!)

A leitura variáveis com a função *scanf* utilizando formatos não adequados aos tipos das variáveis que são passadas como argumentos poderá resultar em erros difíceis de detectar.

- ▶ Exemplos do impacto de conversões indevidas:
 - ▶ ao ler para variável do tipo *int* com o formato "%c" é armazenado o código ASCII do caractere lido no *byte* menos significativo, ficando os restantes *bytes* do inteiro com o valor anterior.
 - ▶ ao ler para variável do tipo *char* com o formato "%d" vai corromper a memória do seu programa, pois vão ser escritos vários bytes quando a variável *char* só tem reservado um único *byte*.

Conversão (*cast*) de tipos

Conversão explícita

Conversão explícita

Sempre que temos uma expressão ou variável de um dado tipo promovendo-o para um tipo de maior capacidade ou despromovendo-o para um tipo de menor capacidade, basta colocar-lhe como prefixo o tipo final desejado entre parêntesis.

Exemplos da [explicitação de conversões](#) que poderiam ter sido implícitas:

```
// Parte de um programa
float x, y ; // para guardar reais
int i; // para guardar um inteiro
x = 3.9;
i = (int) x; // i <- 3
y = (float)i / 2.0 + 0.2; // y <- 3.0/2.0 + 0.2
printf("y=%f, i=%d\n", y, i);
```

Conversão explícita

Exemplo: Conversão usando *(int)*

O programa:

```
1 // ascii-char2int-1.c
2 #include <stdio.h>
3 int main()
4 {   // char para int com conversão EXPLICITA
5     char ch; // para guardar uma letra
6     printf("De-me uma letra:\n");
7     scanf("%c", &ch); // não esquecer o símbolo &
8     printf("O caractere '%c' tem o código ASCII %d\n", ch, (int) ch);
9     return (0);
10 }
```

Produziria a saída, caso o utilizador digitasse a<ENTER>:

```
De-me uma letra:
a
O caractere 'a' tem o codigo ASCII 97
```

- ▶ A função encontra agora como argumento final uma expressão do tipo *int* e não do tipo *char*: com o prefixo *(int)* o programa, na linha 8, **converte explicitamente** o valor de tipo *char* (de *ch*) para o tipo *int* e apresenta o valor correspondente como inteiro.

Exemplo: troca do valor de duas variáveis

Como se pode trocar o valor de duas variáveis, i e j ?

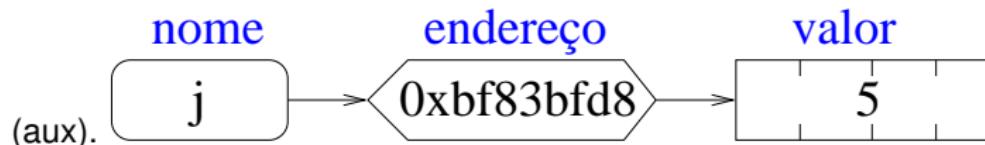
Exemplo: troca do valor de duas variáveis

Como se pode trocar o valor de duas variáveis, i e j ?



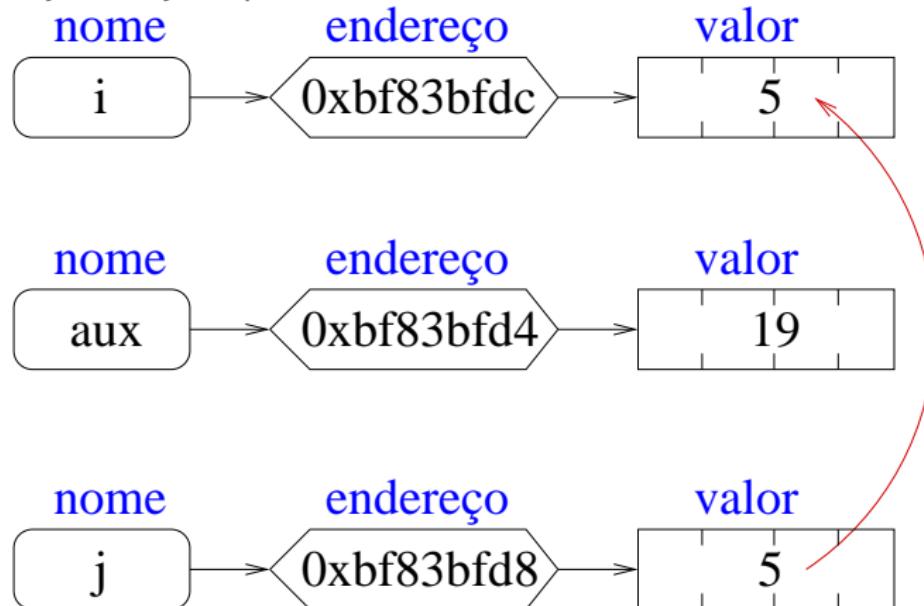
Exemplo: troca do valor de duas variáveis

Em primeiro lugar copia-se o valor de uma delas (*i*) para a variável auxiliar



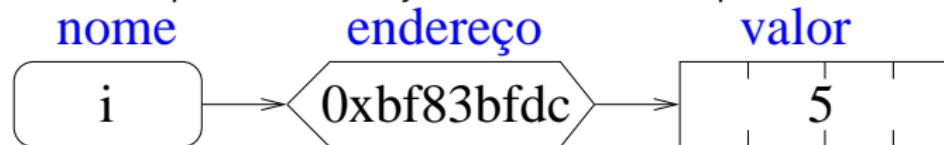
Exemplo: troca do valor de duas variáveis

Em segundo lugar copia-se o valor da segunda variável (*j*) para a variável cujo valor já foi preservado.



Exemplo: troca do valor de duas variáveis

Finalmente, copia-se o valor original de *i*, salvo em aux para o local reservado para o valor de *j*. E a troca está completa!



Exemplo: troca do valor de duas variáveis

Como se pode trocar o valor de duas variáveis, i e j?

```
#include <stdio.h>

int main()
{
    int i = 19; // i e j, as variaveis
    int j = 5; // cujo conteudo vais ser trocado
    int aux; // variavel auxiliar para a troca

    printf("i = %d\tj = %d\n", i, j); //imprimi i e j
    aux = i; // aux <- i (salva o valor de i em aux)
    i = j; // i <- j (i toma o valor de j)
    // imprime o valor de i, j e aux.
    printf("i = %d\tj = %d\taux = %d\n", i, j, aux);
    j = aux; // j <- aux (aux tem o valor de i)
    printf("\nTrocou:\n");
    printf("i = %d\tj = %d\n", i, j); //imprimi i e j
}
```

```
i = 19  j = 5
i = 5   j = 5  aux = 19
```

Trocou:

```
i = 5   j = 19
```

Resumo

Tipos básicos

- ▶ A linguagem C tem 4 tipos básicos: *char*, *int*, *float* e *double*.
- ▶ Uma variável depois de definida (declarada no *main*) fica com valor indefinido.
- ▶ O símbolo permite fazer a atribuição de valores a uma variável:

```
(...)
int a, k, j=4;    /* j <-- 4  a e k indefinidos */
k = j + 3;        /* k <-- 7
a = i = k = 5;    /* atribuições encadeadas */
```

- ▶ Operações entre inteiros (*/*, ***, *+*, *-* *%*) devolvem sempre um inteiro (Exemplo: 17/3 vale 5).
- ▶ As operações entre reais podem usar os símbolos anteriores exceto *%* (resto da divisão entre dois inteiros).

Resumo

Prefixos de Tipos de básicos e Leitura e Escrita

- ▶ Prefixos:

- ▶ ***int*** admite os prefixos *short*, *long* , *signed* e *unsigned*.
- ▶ ***char*** admite os prefixos *signed* e *unsigned*.
- ▶ ***double*** admite o prefixo *long*.

¹Em [1, págs. 69 e 70] é **incorrectamente** referido que %f é válido para ***double***.

Resumo

Prefixos de Tipos de básicos e Leitura e Escrita

- ▶ Prefixos:
 - ▶ **int** admite os prefixos *short*, *long* , *signed* e *unsigned*.
 - ▶ **char** admite os prefixos *signed* e *unsigned*.
 - ▶ **double** admite o prefixo *long*.
- ▶ Os **char** são armazenados sob a forma de inteiros de 8 bits com sinal, logo podemos usar as operações previstas para manipular inteiros sobre valores do tipo **char**, mas o **resultado já não é** do tipo **char**.

¹Em [1, págs. 69 e 70] é **incorrectamente** referido que %f é válido para **double**.

Resumo

Prefixos de Tipos de básicos e Leitura e Escrita

- ▶ Prefixos:
 - ▶ **int** admite os prefixos *short*, *long* , *signed* e *unsigned*.
 - ▶ **char** admite os prefixos *signed* e *unsigned*.
 - ▶ **double** admite o prefixo *long*.
- ▶ Os **char** são armazenados sob a forma de inteiros de 8 bits com sinal, logo podemos usar as operações previstas para manipular inteiros sobre valores do tipo **char**, mas o **resultado já não é** do tipo **char**.
- ▶ Leitura e Escrita
 - ▶ A leitura e a escrita são feitas usando as funções *printf* e *scanf*, utilizando os formatos **requeridos** por cada tipo a ler:
%**d** – int, %**f** – float¹, %**lf** – double, %**Lf** – long double,
%**c** – char (ver slide 31 para mais opções).
 - ▶ No *scanf* o nome de cada variável deve ser precedida de &.
 - ▶ Sendo preciso *promover* uma expressão para determinado tipo, essa promoção é temporária (ver exemplo no slide 47).

¹Em [1, págs. 69 e 70] é **incorrectamente** referido que %f é válido para double.

Programação de Computadores

Capítulo 4: Controlo de Fluxo – Testes e condições



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a Professora Teresa Martinez do DEEC-UC.

(PdC)

Plano: Controlo de Fluxo – Testes e condições

1. Valores lógicos
2. Operadores relacionais
3. A instrução if-else
4. Operadores lógicos
5. O operador ternário ?:
6. A instrução switch

Bibliografia:
Capítulo 3 de [1]

Valores lógicos: Verdade e Falso

Valores lógicos C89

Em C89 não existe nenhum tipo específico de dados para valores lógicos. O valor Falso é representado por 0 (zero) e tudo o que é diferente de 0 é Verdade.

Valores lógicos C99 e C11

Se incluirmos a biblioteca `stdbool.h`, passamos a ter o tipo `bool` e são definidos os símbolos `true` e `false`.

`true` e `false` são expandidos nas constantes inteiras 1 e 0, respectivamente.

Exemplos (sem usar o tipo `bool`):

Falso `0`

Verdade `1, 3, 0.25, -7, 18.5, 6`

Operadores relacionais

Operadores relacionais

Operador	Significado	Exemplo	Interpretação
<code>==</code>	igual	<code>a == b</code>	a é igual a b?
<code>!=</code>	diferente	<code>a != b</code>	a é diferente de b?
<code><</code>	menor que	<code>a < b</code>	a é menor que b?
<code><=</code>	menor ou igual que	<code>a <= b</code>	a é menor ou igual que b?
<code>></code>	maior que	<code>a > b</code>	a é maior que b?
<code>>=</code>	maior ou igual que	<code>a >= b</code>	a é maior ou igual que b?

Operadores relacionais

Operadores relacionais

Operador	Significado	Exemplo	Interpretação
<code>==</code>	igual	<code>a == b</code>	a é igual a b?
<code>!=</code>	diferente	<code>a != b</code>	a é diferente de b?
<code><</code>	menor que	<code>a < b</code>	a é menor que b?
<code><=</code>	menor ou igual que	<code>a <= b</code>	a é menor ou igual que b?
<code>></code>	maior que	<code>a > b</code>	a é maior que b?
<code>>=</code>	maior ou igual que	<code>a >= b</code>	a é maior ou igual que b?

Valores lógicos C99 e C11

Uma expressão que contenha um operador relacional devolve sempre o valor **true** (1) ou **false** (0).

Operadores relacionais

Operadores relacionais

Operador	Significado	Exemplo	Interpretação
<code>==</code>	igual	<code>a == b</code>	a é igual a b?
<code>!=</code>	diferente	<code>a != b</code>	a é diferente de b?
<code><</code>	menor que	<code>a < b</code>	a é menor que b?
<code><=</code>	menor ou igual que	<code>a <= b</code>	a é menor ou igual que b?
<code>></code>	maior que	<code>a > b</code>	a é maior que b?
<code>>=</code>	maior ou igual que	<code>a >= b</code>	a é maior ou igual que b?

Valores lógicos C99 e C11

Uma expressão que contenha um operador relacional devolve sempre o valor **true** (1) ou **false** (0).

NÃO Confundir:

a atribuição (`=`)

e o operador relacional de igualdade (`==`)

O operador `=` atribui a uma variável o valor da expressão à sua direita. O operador `==` verifica se duas expressões têm o mesmo valor.

Operadores relacionais

Exemplo

```
// op-relacionais.c
#include <stdio.h>
int main()
{
    int i, j;
    printf("Digite dois inteiros:"); scanf("%d%d", &i, &j);
    printf("%d == %d vale %d\n", i, j, i == j);
    printf("%d != %d vale %d\n", i, j, i != j);
    printf("%d < %d vale %d\n", i, j, i < j);
    printf("%d <= %d vale %d\n", i, j, i <= j);
    printf("%d > %d vale %d\n", i, j, i > j);
    printf("%d >= %d vale %d\n", i, j, i >= j);
    return 0;
}
```

- ▶ Apresentará no ecrã, se o utilizador digitar **5 7 <ENTER>**:

Operadores relacionais

Exemplo

```
// op-relacionais.c
#include <stdio.h>
int main()
{
    int i, j;
    printf("Digite dois inteiros:"); scanf("%d%d", &i, &j);
    printf("%d == %d vale %d\n", i, j, i == j);
    printf("%d != %d vale %d\n", i, j, i != j);
    printf("%d < %d vale %d\n", i, j, i < j);
    printf("%d <= %d vale %d\n", i, j, i <= j);
    printf("%d > %d vale %d\n", i, j, i > j);
    printf("%d >= %d vale %d\n", i, j, i >= j);
    return 0;
}
```

- Apresentará no ecrã, se o utilizador digitar **5 7 <ENTER>**:

- ```
Digite dois inteiros: 5 7
5 == 7 vale 0
5 != 7 vale 1
5 < 7 vale 1
5 <= 7 vale 1
5 > 7 vale 0
5 >= 7 vale 0
```

# A instrução *if-else*

Sintaxe do *if-else* sem blocos:

```
if (condição)
 instruçãoV ;
[else instruçãoF;]
```

A componente *else* do *if* é facultativa (está entre parêntesis retos)

Ainda a sintaxe:

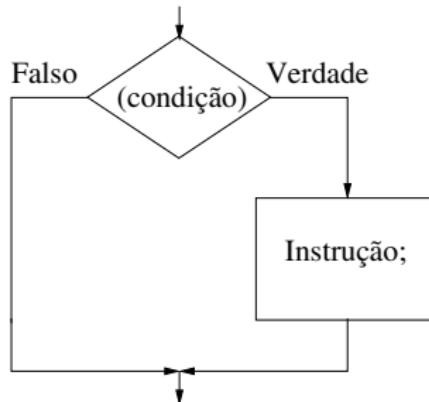
- ▶ A *condição* tem de estar dentro de parêntesis curvos
- ▶ A *instruçãoV* e a *instruçãoF* são seguidas de  [ ; ]

# A instrução *if*

Com uma única instrução

```
if (condição)
 instrução;
```

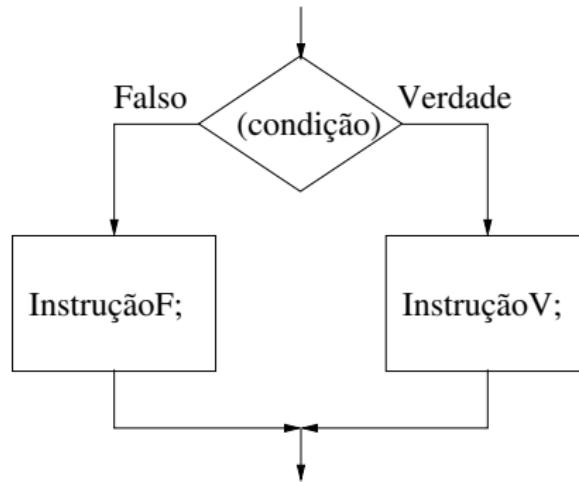
Se a *condição* é verdadeira (true), ou seja diferente de zero, a instrução que se segue é executada; caso contrário essa instrução não é executada.



# A instrução *if* com *else*

Com uma única instrução em ambos os ramos

- ▶ A instrução `else` é sempre precedida de um `if`.
- ▶ `if (condição)`  
*instruçãoV;*  
`else`  
*instruçãoF;*



# Exemplo de *if-else*

E operador relacional

```
//if-else-1.c
#include <stdio.h>
int main()
{
 int k;
 printf("Introduza um número inteiro: ");
 scanf("%d", &k);
 /* Note a indentação! */
 if (k >= 0)
 printf("%d é um número positivo\n", k);
 else
 printf("%d é um número negativo\n", k);

 return (0);
}
```

Duas possíveis execuções do programa anterior:

# Exemplo de *if-else*

E operador relacional

```
//if-else-1.c
#include <stdio.h>
int main()
{
 int k;
 printf("Introduza um número inteiro: ");
 scanf("%d", &k);
 /* Note a indentação! */
 if (k >= 0)
 printf("%d é um número positivo\n", k);
 else
 printf("%d é um número negativo\n", k);

 return (0);
}
```

Duas possíveis execuções do programa anterior:

- ▶ Introduza um número inteiro: 8  
8 é um número positivo

# Exemplo de *if-else*

E operador relacional

```
//if-else-1.c
#include <stdio.h>
int main()
{
 int k;
 printf("Introduza um número inteiro: ");
 scanf("%d", &k);
 /* Note a indentação! */
 if (k >= 0)
 printf("%d é um número positivo\n", k);
 else
 printf("%d é um número negativo\n", k);

 return (0);
}
```

Duas possíveis execuções do programa anterior:

- ▶ Introduza um número inteiro: 8  
8 é um número positivo

- ▶ Introduza um número inteiro: -5  
-5 é um número negativo

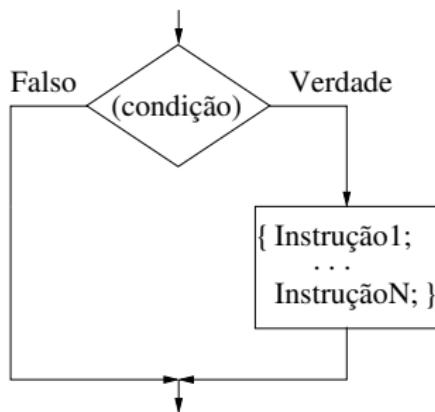
# A instrução *if*

Com um bloco de instruções

```
if (condição)
{
 instrução 1;
 instrução 2;
 ...
 instrução N;
}
```

Se a *condição* é verdadeira (*true*), ou seja diferente de zero, o bloco de instruções delimitado por `{ }` que se segue é executado; caso contrário esse bloco de instruções não é executado.

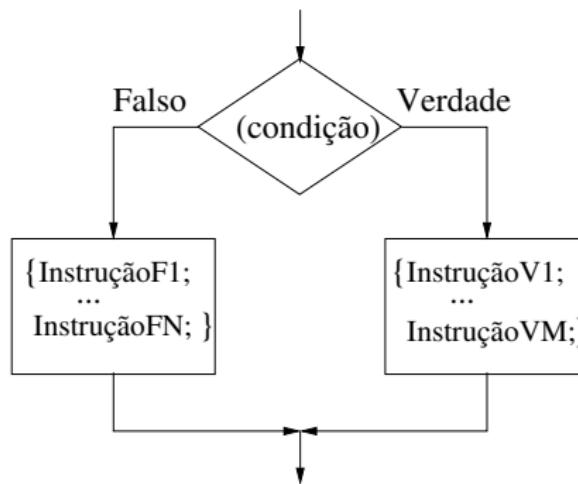
**Nota:** Depois de um bloco não precisa colocar ponto e vírgula (;)



# A instrução *if-else*

Com blocos de instruções em ambos os ramos

- ▶ A instrução `else` é sempre precedida de um `if`.
- ▶ `if (condição)`  
    { *InstruçãoV1; ... InstruçãoVN;* }  
`else`  
    { *InstruçãoF1; ... InstruçãoFM;* }



# *if-else* encadeados

- Quando existem várias opções, cada uma requerendo uma acção específica, podemos utilizar *if-else* encadeados (ou em cascata):

```
//if-else-encadeado-1.c
#include <stdio.h>
int main() {
 int opcao;
 printf("1 - Comprar\n2 - Vender\n3 - Trocar\n4 - Sair");
 printf("\nQual a sua escolha: ");
 scanf("%d", &opcao);
 /* if-else encadeados (alternativas mutuamente exclusivas)*/
 if (1 == opcao)
 printf("*Vai comprar!");
 else
 if (2 == opcao)
 printf("*Vai vender!");
 else
 if (3 == opcao)
 printf("*Vai trocar!");
 else
 if (4 == opcao)
 printf("*Abandona!");
 else
 printf("Opcao invalida!");
 /* fim dos if-else encadeados */
 printf("\n");
 return 0;
}
```

# *if-else* encadeados

- Quando existem várias opções, cada uma requerendo uma acção específica, podemos utilizar *if-else* encadeados (ou em cascata):

```
//if-else-encadeado-1.c
#include <stdio.h>
int main() {
 int opcao;
 printf("\n1 - Comprar\n2 - Vender\n3 - Trocar\n4 - Sair");
 printf("\nQual a sua escolha: ");
 scanf("%d", &opcao);
 /* if-else encadeados (alternativas mutuamente exclusivas)*/
 if (1 == opcao)
 printf("*Vai comprar!");
 else
 if (2 == opcao)
 printf("*Vai vender!");
 else
 if (3 == opcao)
 printf("*Vai trocar!");
 else
 if (4 == opcao)
 printf("*Abandona!");
 else
 printf("Opcao invalida!");
 /* fim dos if-else encadeados */
 printf("\n");
 return 0;
}
```

- A **indentação** em cima é uma possibilidade, mas no caso de *if-else* encadeados, em que cada instrução é uma opção alternativa, também se pode escrever o programa como se vê no slide seguinte.

# *if-else* encadeados

- Cada instrução dos *if-else* encadeados (ou em cascata), corresponde a um opção única:

```
//if-else-encadeado-2.c
#include <stdio.h>
int main() {
 int opcao;
 printf("1 - Comprar\n2 - Vender\n3 - Trocar\n4 - Sair");
 printf("\nQual a sua escolha: ");
 scanf("%d", &opcao);
 if (1 == opcao) printf("*Vai comprar!");
 else if (2 == opcao) printf("*Vai vender!");
 else if (3 == opcao) printf("*Vai trocar!");
 else if (4 == opcao) printf("*Abandona!");
 else printf("Opcao invalida!");
 printf("\n");
 return 0;
}
```

```
1 - Comprar
2 - Vender
3 - Trocar
4 - Sair
Qual a sua escolha: 2
*Vai vender!
```

# Notas sobre *if-else* encadeados

## *if-else* encadeados

Sempre que existam *if-else* encadeados, cada componente *else* pertence sempre ao último *if* que não tenha um *else* associado.

## *if-else* encadeados, incluindo algum *if* sem *else*

Sempre que existam *if-else* encadeados, e um *if* não tenha *else* é necessário usar chavetas para evitar que o *else* de outro *if* mais exterior não seja erradamente associado ao anterior.

```
// prog308.c [1] alterado
#include <stdio.h>
int main()
{ /* Programa com ERRO lógico */
 int a,b;
 printf("Introduza dois inteiros: ");
 scanf("%d%d",&a,&b);
 /* A indentação induz em ERRO... */
 if (a >= 0)
 if (b > 10)
 printf("A>=0 e B é muito grande\n");
 else
 printf("A tem um valor negativo\n");
 return(0);
}
```

```
// prog308b.c [1] alterado
#include <stdio.h>
int main()
{ /* Programa CORRIGIDO */
 int a,b;
 printf("Introduza dois inteiros: ");
 scanf("%d%d",&a,&b);
 /* É preciso usar as chavetas */
 if (a >= 0)
 {
 if (b > 10)
 printf("A>=0 e B é muito grande\n");
 }
 else
 printf("A tem um valor negativo\n");
 return(0);
}
```

# Operadores lógicos

Precedência dos Operadores Lógicos e Relacionais

## Operadores Lógicos

| Operador | Significado           | Utilização       |
|----------|-----------------------|------------------|
|          | OU lógico ( $\vee$ )  | (exp1)    (exp2) |
| &&       | E lógico ( $\wedge$ ) | (exp1) && (exp2) |
| !        | Negação ( $\neg$ )    | ! (exp1)         |

## Precedência dos Operadores Lógicos e Relacionais e do Operador Ternário

| Operadores               |
|--------------------------|
| <      <=      >      >= |
| ==      !=               |
| &&                       |
|                          |
| ? :                      |

E em caso de dúvida utilize os parêntesis curvos ()

Na referência [2], na página 47 Tabela 2.9 as linhas com os operadores & e | estão erradas e devem ser ignoradas; ignore também o exemplo nessa página.

# Ordem de avaliação

Tabela de verdade de operadores lógicos

## Ordem avaliação nas expressões com operadores lógicos

Nas expressões em que surgem `&&` ou `||`, o seu valor é avaliado da esquerda para a direita e *é interrompida* assim que o *valor final* (verdade ou falso) é conhecido – *lazy evaluation*.

Esta é uma característica que algumas vezes é útil explorar!

**Negação**

| A       | $! A$   |
|---------|---------|
| Falso   | Verdade |
| Verdade | Falso   |

**Operadores Lógicos**

| A       | B       | OU lógico<br>$A  B$ | E lógico<br>$A&&B$ |
|---------|---------|---------------------|--------------------|
| Falso   | Falso   | Falso               | Falso              |
| Falso   | Verdade | Verdade             | Falso              |
| Verdade | Falso   | Verdade             | Falso              |
| Verdade | Verdade | Verdade             | Verdade            |

# O operador ternário ?:

O operador ternário é um operador condicional

- ▶ Forma geral do operador ternário `?:` `condição ? Valor_V : Valor_F`
- ▶ Se a `condição` tem o valor *true*, executa `Valor_V`, senão executa `Valor_F`.

# O operador ternário ?:

O operador ternário é um operador condicional

- ▶ Forma geral do operador ternário `?:` `condição ? Valor_V : Valor_F`
- ▶ Se a `condição` tem o valor *true*, executa `Valor_V`, senão executa `Valor_F`.

## :? e if-else

- ▶ O `if-else` determina a sequência de instruções a executar.
- ▶ O operador `?:` devolve sempre um resultado, mas o `if-else` não.  
Por esse motivo o `?:` pode ser utilizado numa expressão mas o `if-else` não!

# O operador ternário ?:

- ▶ Utilizando o operador ternário `?:` numa expressão.

# O operador ternário ?:

- ▶ Utilizando o operador ternário `?:` numa expressão.
- ▶ Calcular e mostrar o maior de dois números, dados pelo utilizador:

```
//op-ternario.c
#include <stdio.h>
int main()
{
 int i, j, maior;
 printf("De-me dois inteiros: ");
 scanf("%d%d", &i, &j);
 maior = i > j ? i : j ;
 printf("max(%d, %d) = %d\n", i, j, maior);
 return 0;
}
```

# O operador ternário ?:

- ▶ Utilizando o operador ternário `?:` numa expressão.
- ▶ Calcular e mostrar o maior de dois números, dados pelo utilizador:

```
//op-ternario.c
#include <stdio.h>
int main()
{
 int i, j, maior;
 printf("De-me dois inteiros: ");
 scanf("%d%d", &i, &j);
 maior = i > j ? i : j ;
 printf("max(%d, %d) = %d\n", i, j, maior);
 return 0;
}
```

- ▶ Exemplificando :

```
De-me dois inteiros: 5 67
max(5, 67) = 67
```

# O operador ternário ?:

- Resolvendo o problema anterior usando um *if else*:

```
//op-ternario-vs-if.c
#include <stdio.h>
int main()
{
 int i, j, maior;
 printf("De-me dois inteiros: ");
 scanf("%d%d", &i, &j);

 if(i > j)
 maior = i;
 else
 maior = j;

 printf("max(%d, %d) = %d\n", i, j, maior);
 return 0;
}
```

# O operador ternário ?:

- Resolvendo o problema anterior usando um *if else*:

```
//op-ternario-vs-if.c
#include <stdio.h>
int main()
{
 int i, j, maior;
 printf("De-me dois inteiros: ");
 scanf("%d%d", &i, &j);

 if(i > j)
 maior = i;
 else
 maior = j;

 printf("max(%d, %d) = %d\n", i, j, maior);
 return 0;
}
```

- É quase sempre preferível utilizar um *if-else* ao operador ternário.

Contudo o operador **?:** conduz em geral a código mais compacto:  
`printf("max(%d, %d) = %d\n", i, j, (i>j ? i : j));`

# Quando os *if-else* encadeados ficam difíceis de interpretar, um *switch* é uma boa alternativa

```
//prog0311-alterado.c (adaptação de prog0311.c [1])
#include <stdio.h>
int main()
{
 char estado;
 printf("Qual o Estado Civil:");
 estado = getchar();
 if (estado == 'S' || estado == 's')
 printf("Solteiro");
 else
 if (estado == 'P' || estado == 'p')
 printf("Separado");
 else
 if (estado == 'C' || estado == 'c')
 printf("Casado");
 else
 if (estado == 'D' || estado == 'd')
 printf("Divorciado");
 else
 if (estado == 'V' || estado == 'v')
 printf("Viúvo");
 else
 printf("Estado Civil Inválido");
 printf("\n");
 return(0);
}
```

# A instrução *switch*

Substituindo os *if-else* por *switch*

- A instrução *switch* é uma alternativa aos *if-else* encadeados, particularmente interessante quando estes têm muitas opções.

## Sintaxe do *switch*

```
switch (expressão)
{
 case constante_1: instruções_1;
 case constante_2: instruções_2;
 ...
 case constante_n: instruções_n;
 [case default: instruções;]
}
```

- A *expressão* tem de ter um valor do tipo inteiro (*char*, *int*, *short* ou *long*).
- Se o valor da expressão for igual a alguma das constantes após um *case*, então são executadas as instruções que se seguem a esse *case*; caso contrário são executadas as instruções após o *default*.
- O *default* é opcional, pelo que se este não existir e nenhuma constante coincidir com *expressão* o *switch* termina sem fazer nada.

# A instrução switch

## A instrução *switch* sem *break*

Usando a instrução *switch* (sem *break*):

```
//prog0312-sem-break.c (prog0312.c [1] alterado)
#include <stdio.h> /* Implementação ERRADA */
int main() /* Implementação ERRADA */
{ /* Implementação ERRADA */
 char estado;
 printf("Qual o estado Civil: ");
 scanf(" %c",&estado); /* ou estado = getchar(); */
 switch(estado)
 {
 case 'C': printf("Casado\n");
 case 'S': printf("Solteiro\n");
 case 'P': printf("SeParado\n");
 case 'D': printf("Divorciado\n");
 case 'V': printf("Viúvo\n");
 default : printf("Estado Civil Incorrecto\n");
 }
 return 0;
}
```

Se o utilizador escrever: P<enter>, a saída é:

```
SeParado
Divorciado
Viúvo
Estado Civil Incorrecto
```

# A instrução *switch*

A instrução *switch* e o *break*:

- Na descrição da sintaxe do *switch* está:

*"Se o valor da expressão for igual a alguma das constantes após um case, então são executadas as instruções que se seguem a esse case,"*

faltou reforçar que são mesmo **todas** as intruções que se seguem (incluindo as dos *case* seguintes), até que o *switch* termine, ou seja encontrado um *break*.

# A instrução *switch*

A instrução *switch* e o *break*:

- ▶ Na descrição da sintaxe do *switch* está:

*"Se o valor da expressão for igual a alguma das constantes após um case, então são executadas as instruções que se seguem a esse case,"*

faltou reforçar que são mesmo **todas** as intruções que se seguem (incluindo as dos *case* seguintes), até que o *switch* termine, ou seja encontrado um *break*.

- ▶ Ou seja, para impedir o efeito de "queda livre" pelas restantes opções do switch deve usar-se a intrução *break*, que termina o *switch* assim que é encontrada.

# A instrução *switch*

A instrução *switch* e o *break*:

- ▶ Na descrição da sintaxe do *switch* está:

*"Se o valor da expressão for igual a alguma das constantes após um case, então são executadas as instruções que se seguem a esse case,"*

faltou reforçar que são mesmo **todas** as intruções que se seguem (incluindo as dos *case* seguintes), até que o *switch* termine, ou seja encontrado um *break*.

- ▶ Ou seja, para impedir o efeito de "queda livre" pelas restantes opções do *switch* deve usar-se a intrução *break*, que termina o *switch* assim que é encontrada.
- ▶ A última opção (o *default* ou o último *case*) não precisa de *break*, porque no final deste o *switch* termina – mas pode ser colocado para evitar problemas, quando se acrescenta uma nova opção...

# A instrução *switch*

A instrução *switch* e o *break*:

- ▶ Na descrição da sintaxe do *switch* está:

*"Se o valor da expressão for igual a alguma das constantes após um case, então são executadas as instruções que se seguem a esse case,"*

faltou reforçar que são mesmo **todas** as intruções que se seguem (incluindo as dos *case* seguintes), até que o *switch* termine, ou seja encontrado um *break*.

- ▶ Ou seja, para impedir o efeito de "queda livre" pelas restantes opções do *switch* deve usar-se a intrução *break*, que termina o *switch* assim que é encontrada.
- ▶ A última opção (o *default* ou o último *case*) não precisa de *break*, porque no final deste o *switch* termina – mas pode ser colocado para evitar problemas, quando se acrescenta uma nova opção...
- ▶ O *default* não precisa de ser a última opção.
- ▶ Não é obrigatório haver a opção *default*, mas é boa prática.

# A instrução switch

## A instrução switch com break

Usando a instrução switch com break:

```
//prog0312-com-break.c (prog0321.c [1] alterado)
#include <stdio.h>

int main()
{
 char estado;
 printf("Qual o estado Civil: ");
 scanf(" %c",&estado); /* ou EstCivil = getchar(); */
 switch(estado)
 {
 case 'C': printf("Casado\n"); break;
 case 'S': printf("Solteiro\n"); break;
 case 'P': printf("SeParado\n"); break;
 case 'D': printf("Divorciado\n"); break;
 case 'V': printf("Viúvo\n"); break;
 default : printf("Estado Civil Incorrecto\n");
 }
 return 0;
}
```

Se o utilizador escrever: D<enter>, a saída é:

Divorciado

# A instrução switch

Reescrevendo com um *switch* o programa fica mais fácil de interpretar:

Reescrevendo o programa no slide 20:

```
// prog013.c [1] alterado
#include <stdio.h>

int main()
{
 char estado;
 printf("Qual o estado Civil: ");
 scanf(" %c",&estado); /* ou estado = getchar(); */
 switch(estado)
 {
 case 'c':
 case 'C': printf("Casado\n"); break;
 case 's':
 case 'S': printf("Solteiro\n"); break;
 case 'p':
 case 'P': printf("Separado\n"); break;
 case 'd':
 case 'D': printf("Divorciado\n"); break;
 case 'v':
 case 'V': printf("Viúvo\n"); break;
 default : printf("Estado Civil Incorrecto\n");
 }
 return 0;
}
```

# A instrução *switch* e o *if-else*

```
switch (expressao)
{
 case constante_1: instrucoes_1; break;
 case constante_2: instrucoes_2; break;
 ...
 case constante_n: instrucoes_n; break;
 [case default: instrucoes;]
}
```

*switch* com *break* equivale aos *ifs* em cascata (exemplo [1, pág. 110], mas adicionando chavetas):

```
if (expressao == constante_1)
 { instrucoes_1; } /*{} porque pode ser mais que uma instrução*/
else
 if (expressao == constante_2)
 { instrucoes_2; }
 ...
 else
 if (expressao == constante_n)
 { instrucoes_n; }
 [else
 { instrucoes; }] /* default é opcional */
```

# Resumo

## Valores lógicos e instrução *if else*

- ▶ Na linguagem C não existe nenhum típico específico de dados para valores lógicos:
  - ▶ O valor **Falso** é representado por 0 (zero) e tudo o que é diferente de 0 é **Verdade**.
  - ▶ Embora qualquer valor não nulo seja **Verdade** é fortemente desaconselhado o uso de variáveis reais para representar valores lógicos.

# Resumo

## Valores lógicos e instrução *if else*

- ▶ Na linguagem C não existe nenhum típico específico de dados para valores lógicos:
  - ▶ O valor **Falso** é representado por 0 (zero) e tudo o que é diferente de 0 é **Verdade**.
  - ▶ Embora qualquer valor não nulo seja **Verdade** é fortemente desaconselhado o uso de variáveis reais para representar valores lógicos.
- ▶ O *if else* e o *switch* permitem condicionar a sequência de instruções com base no teste de condições.
  - ▶ No *if else* a condição é avaliada e se for verdadeira é executada a instrução (ou **bloco** de instruções – conjunto de instruções entre chavetas) associado ao *if*; caso contrário é executada a instrução (ou bloco de instruções) associado ao *else*.
  - ▶ A componente *else* do *if* é opcional.
  - ▶ A **indentação** é importante: serve para representar as dependências que algumas instruções têm de outras que as precedem.

# Resumo

## Operadores lógicos, instruções *switch* e *break*

- ▶ Podemos operar sobre valores lógicos usando os operadores binários `&&` (E lógico) e `||` (OU lógico) ou o operador unário `!`.
  - ▶ O resultado de uma expressão com operadores lógicos será também um valor lógico.
  - ▶ `&&` tem prioridade sobre `||` e o `!` tem prioridade sobre ambos.
  - ▶ Em caso de dúvida sobre a ordem dos operadores use parêntesis!

# Resumo

## Operadores lógicos, instruções *switch* e *break*

- ▶ Podemos operar sobre valores lógicos usando os operadores binários `&&` (E lógico) e `||` (OU lógico) ou o operador unário `!`.
  - ▶ O resultado de uma expressão com operadores lógicos será também um valor lógico.
  - ▶ `&&` tem prioridade sobre `||` e o `!` tem prioridade sobre ambos.
  - ▶ Em caso de dúvida sobre a ordem dos operadores use parêntesis!
- ▶ O **operador ternário** poderá ser preferido (ao *if else*) quando uma mesma variável recebe um de dois valores, dependendo de uma condição.

# Resumo

## Operadores lógicos, instruções *switch* e *break*

- ▶ Podemos operar sobre valores lógicos usando os operadores binários `&&` (E lógico) e `||` (OU lógico) ou o operador unário `!`.
  - ▶ O resultado de uma expressão com operadores lógicos será também um valor lógico.
  - ▶ `&&` tem prioridade sobre `||` e o `!` tem prioridade sobre ambos.
  - ▶ Em caso de dúvida sobre a ordem dos operadores use parêntesis!
- ▶ O **operador ternário** poderá ser preferido (ao *if else*) quando uma mesma variável recebe um de dois valores, dependendo de uma condição.
- ▶ Ao contrário do *if else*, o ***switch*** **não permite** definir gamas de valores, e.g., (`k >= 5 && k < 10`).  
Mas o ***switch*** permite evitar *if else* encadeados, quando existe um conjunto pré-definido de valores constantes (*int, char ou long int*).

# Resumo

## Operadores lógicos, instruções *switch* e *break*

- ▶ Podemos operar sobre valores lógicos usando os operadores binários `&&` (E lógico) e `||` (OU lógico) ou o operador unário `!`.
  - ▶ O resultado de uma expressão com operadores lógicos será também um valor lógico.
  - ▶ `&&` tem prioridade sobre `||` e o `!` tem prioridade sobre ambos.
  - ▶ Em caso de dúvida sobre a ordem dos operadores use parêntesis!
- ▶ O **operador ternário** poderá ser preferido (ao *if else*) quando uma mesma variável recebe um de dois valores, dependendo de uma condição.
- ▶ Ao contrário do *if else*, o ***switch*** **não permite** definir gamas de valores, e.g., (`k >= 5 && k < 10`).  
Mas o ***switch*** permite evitar *if else* encadeados, quando existe um conjunto pré-definido de valores constantes (*int, char ou long int*).
- ▶ A instrução ***break*** permite interromper a execução de uma sequência de instruções dentro de um *switch*. As instruções associadas a um *case* não precisam de chavetas: o *break* diz quando o bloco é interrompido.

# Programação de Computadores

## Capítulo 5: Controlo de Fluxo – Ciclos



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a  
Professora Teresa Martinez do DEEC-UC.

(PdC)

# Plano: Controlo de Fluxo – Ciclos

1. Ciclos: a instrução while
2. Exemplos
3. Ciclos: A instrução for
4. Ciclos: a instrução do-while
5. A instrução break
6. A instrução continue
7. Operadores Unários de atribuição
8. Atribuição composta
9. Exemplos

Bibliografia:  
Capítulo 4 de [1]

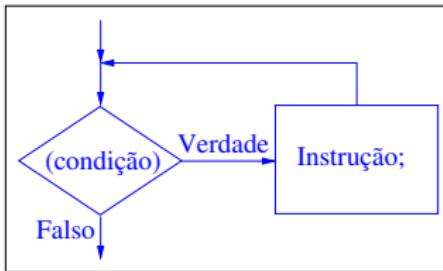
# Ciclos: a instrução *while*

- ▶ Para que um computador **repita** uma tarefa, necessitamos que um programa possa implementar ciclos.

# Ciclos: a instrução *while*

- ▶ Para que um computador **repita** uma tarefa, necessitamos que um programa possa implementar ciclos.
- ▶ Forma geral:

**while** (*condição*)  
    *Instrução;*



# Ciclos: a instrução *while*

- ▶ Para que um computador **repita** uma tarefa, necessitamos que um programa possa implementar ciclos.

- ▶ Forma geral:

```
while (condição)
 Instrução;
```

ou

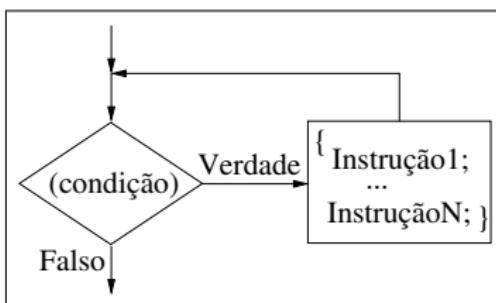
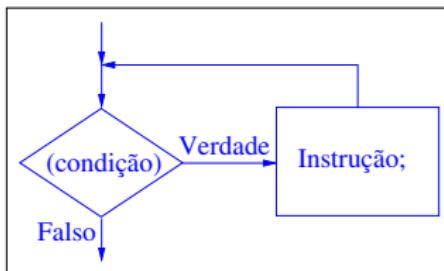
```
while (condição)
{
```

*Instrução1*;

...

*InstruçãoN*;

}



- ▶ Se a **condição** de entrada é falsa, o ciclo não é executado nenhuma vez!

# Ciclos: a instrução *while*

Exemplo: visualizar os códigos numéricos (decimais) ASCII correspondentes a 5 símbolos

► Considere o seguinte problema:

- Escreva um programa que permita visualizar os códigos numéricos (decimais) ASCII correspondentes a 5 símbolos da tabela ASCII.
- A primeira dessas letras deve poder ser introduzida pelo utilizador do programa.
- Declare uma variável letra, do tipo char, e apresente os 5 valores seguintes como se apresenta:

| letra |  | Dec. |
|-------|--|------|
| a     |  | 97   |
| b     |  | 98   |
| ...   |  | ...  |

---

No exemplo "Código fonte 2.42. Instrução iterativa while" da referência [2, pág. 64] falta a instrução de inicialização. Nesse exemplo caso a variável *mult* tivesse o valor 0, o ciclo seria infinito!

# Ciclos: a instrução *while*

## Exemplo - algoritmo

- ▶ Um algoritmo de resolução deste problema tem dois passos principais:
  - ▶ Pedir um carácter ao utilizador, em seguida ler e armazenar esse carácter em letra
  - ▶ Repetir 5 vezes o mesmo tipo de instrução
    - ▶ mostrar o *char* (letra + 0) e o seu código como pedido
    - ▶ mostrar o *char* (letra + 1) e o seu código como pedido
    - ▶ mostrar o *char* (letra + 2) e o seu código como pedido
    - ▶ mostrar o *char* (letra + 3) e o seu código como pedido
    - ▶ mostrar o *char* (letra + 4) e o seu código como pedido

# Ciclos: a instrução *while*

Exemplo sem ciclo *while*

- ▶ Uma resolução do problema **sem utilização de ciclos nem ifs**:

```
// letras-sem-ciclo.c
#include <stdio.h>

int main()
{
 char letra;
 // pede e le caracter
 printf("De-me um caracter: ");
 letra = getchar();

 // Repete 5 vezes
 printf("letra | Dec.\n");
 printf(" %c | %d\n", (char)(letra+0), (letra+0));
 printf(" %c | %d\n", (char)(letra+1), (letra+1));
 printf(" %c | %d\n", (char)(letra+2), (letra+2));
 printf(" %c | %d\n", (char)(letra+3), (letra+3));
 printf(" %c | %d\n", (char)(letra+4), (letra+4));
 return(0);
}
```

# Ciclos: a instrução *while*

Exemplo sem ciclo *while*

- Exemplo, quando o utilizador digita  A:

```
De-me um caracter: A
letra | Dec.
```

|   |  |    |
|---|--|----|
| A |  | 65 |
| B |  | 66 |
| C |  | 67 |
| D |  | 68 |
| E |  | 69 |

# Ciclos: a instrução *while*

## Exemplo com ciclo *while*

- ▶ Uma resolução do problema utilizando o ciclo *while*:

```
// letras-com-ciclo-while.c
#include <stdio.h>

int main()
{
 char letra;
 // pede e le caracter
 printf("De-me um caracter: ");
 letra = getchar();

 printf("letra | Dec.\n");

 int i = 0; // para obter letra+i, com i = 0,1,2,3,4
 while(i < 5) // quando i == 5 o ciclo termina
 {
 printf("%c | %d\n", (char)(letra+i), (letra+i));
 i = i + 1; /* i <-- 1, 2, 3, 4, 5 */
 }
 return(0);
}
```

# Ciclos: a instrução *while*

Exemplo com ciclo *while*

- Exemplo, quando o utilizador digita  e:

```
De-me um caracter: e
letra | Dec.
e | 101
f | 102
g | 103
h | 104
i | 105
```

- Exemplo, quando o utilizador digita  3:

```
De-me um caracter: 3
letra | Dec.
3 | 51
4 | 52
5 | 53
6 | 54
7 | 55
```

# Ciclos: a instrução *while*

Exemplo com ciclo *while*

- ▶ Impondo que o utilizador apenas deve introduzir **letras maiúsculas!**

```
// letras-M-ciclo-while.c
#include <stdio.h>
int main()
{
 char letra = '?'; // valor inicial que não pertence a [A,Z]
 // repete enquanto letra fora de [A,Z]
 while(letra < 'A' || letra > 'Z')
 {
 printf("De-me um caracter [A..Z]: ");
 scanf(" %c", &letra); // Note o espaço antes do formato %c
 if(letra < 'A' || letra > 'Z')
 printf("A letra dada '%c' não é maiuscula!\n", letra);
 }

 printf("letra | Dec.\n"); // Mostra como desejado
 int i = 0; // para obter letra+i, com i = 0,1,2,3,4
 while(i < 5) // quando i == 5 o ciclo termina
 {
 printf(" %c | %d\n", (char)(letra+i), (letra+i));
 i = i + 1; // i <-- 1, 2, 3, 4, 5
 }
 return(0);
}
```

# Ciclos: a instrução *while*

Exemplo com ciclo *while*

- Exemplo quando o utilizador digita d<enter>4<enter>F<enter>I:

```
De-me um caracter [A..Z]: d
A letra dada 'd' nao e maiuscula!
De-me um caracter [A..Z]: 4
A letra dada '4' nao e maiuscula!
De-me um caracter [A..Z]: I
letra | Dec.
 I | 73
 J | 74
 K | 75
 L | 76
 M | 77
```

# Exemplo: Mostra os números de 1 a n

```
1 // mostra-num-while.c
2 #include <stdio.h>
3 int main() // Programa que mostra os números 1..n
4 {
5 int n = -1; // => entrar no ciclo
6 while(n <= 0)
7 {
8 printf("De-me um inteiro maior que zero: ");
9 scanf("%d", &n);
10 } // end while
11
12 i = 1; // Conta de 1 ate n
13 while(i <= n)
14 {
15 printf("%d\n", i);
16 i = i +1; // i <- i +1
17 }
18 return 0;
19 }
```

```
De-me um inteiro maior que zero: -4
```

```
De-me um inteiro maior que zero: 4
```

```
1
2
3
4
```

# Exemplo: Cálculo de um somatório

Introduz *putchar*

```
// somatorio-while.c
#include <stdio.h>
int main() // Programa que calcula \sum_{i=1}^n i, i > 0
{
 int n, i, soma;

 /* Ciclo de leitura de número estritamente positivo -----*/
 n = -1; // => entrar no ciclo
 while(n <= 0)
 {
 printf("De-me um inteiro maior que zero: ");
 scanf("%d", &n);
 } // end while

 /* Ciclo de cálculo do somatório -----*/
 soma = 0; // Valor inicial de soma: elemento neutro da adição
 i = 1; // Conta de 1 ate' n
 while(i <= n)
 {
 soma = soma + i; // soma <- 1 + ... + i
 i = i+1; // prepara parcela seguinte
 } // end while

 printf("A soma de 1 ate %d: %d", n, soma);
 putchar('\n'); /* envia um único char para a consola *****/
 return 0;
}
```

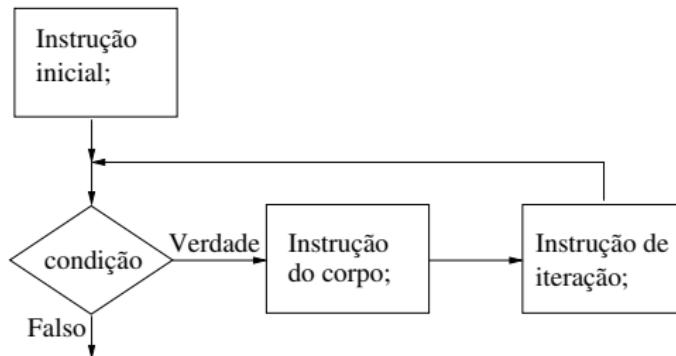
```
De-me um inteiro maior que zero: -4
De-me um inteiro maior que zero: 3
A soma de 1 ate 3: 6
```

# Ciclos: A instrução *for*

O corpo é uma só instrução (1/3)

- ▶ Forma geral quando o corpo é uma só instrução:

```
for (instrução inicial; condição; instrução de iteração)
 instrução do corpo;
```



- ▶ Se a **condição** de entrada é falsa, o corpo do ciclo não é executado nenhuma vez!

# Ciclos: A instrução *for*

O corpo é uma só instrução (2/3)

- ▶ Forma geral quando o corpo é uma só instrução:

```
for (instrução inicial; condição; instrução de iteração)
 instrução do corpo;
```

- ▶ Isto é equivalente a:

```
{
 instrução inicial ;
 while (condição) {
 instrução do corpo;
 instrução de iteração ;
 }
}
```

# Ciclos: A instrução *for*

O corpo é uma só instrução (3/3)

```
for (instrução inicial; condição; instrução de iteração)
 instrução do corpo;
```

- De notar que a *instrução inicial* só é executada uma vez!

# Ciclos: A instrução `for`

O corpo é uma só instrução (3/3)

```
for (instrução inicial; condição; instrução de iteração)
 instrução do corpo;
```

- ▶ De notar que a *instrução inicial* só é executada uma vez!
- ▶ A *condição* é avaliada após a *instrução inicial*. Se tiver o valor `true`, então a *instrução do corpo* é executada pela primeira vez. Caso contrário a instrução `for` termina.

# Ciclos: A instrução `for`

O corpo é uma só instrução (3/3)

```
for (instrução inicial; condição; instrução de iteração)
 instrução do corpo;
```

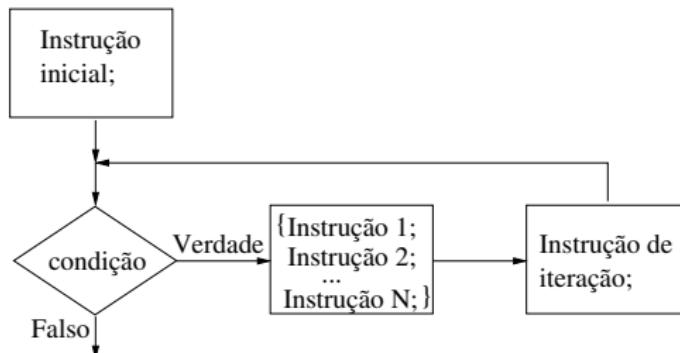
- ▶ De notar que a *instrução inicial* só é executada uma vez!
- ▶ A *condição* é avaliada após a *instrução inicial*. Se tiver o valor `true`, então a *instrução do corpo* é executada pela primeira vez. Caso contrário a instrução `for` termina.
- ▶ Após a *instrução do corpo*; é executada a *instrução de iteração* e seguidamente a *condição* é reavaliada. Se esta ainda tiver o valor `true` a *instrução do corpo*; é novamente executada, seguindo-se a *instrução de iteração* e novamente a reavaliação da *condição*. Este processo é repetido até que *condição* tenha o valor `false`.

# Ciclos: A instrução *for*

O corpo é um bloco de instruções (1/2)

- Forma geral quando o corpo é um bloco de instruções:

```
for (instrução inicial; condição; instrução de iteração) {
 instrução 1 do corpo;
 instrução 2 do corpo;
 ...
 instrução N do corpo;
}
```



# Ciclos: A instrução *for*

O corpo é um bloco de instruções (2/2)

- ▶ Forma geral quando o corpo é um bloco de instruções:

```
for (instrução inicial; condição; instrução de iteração) {
 instrução 1 do corpo;
 instrução 2 do corpo;
 ...
 instrução N do corpo;
}
```

- ▶ Isto é equivalente a:

```
{
 instrução inicial;
 while (condição) {
 instrução 1 do corpo;
 instrução 2 do corpo;
 ...
 instrução N do corpo;
 instrução de iteração;
 }
}
```

# Ciclos: A instrução for

Um exemplo: *for* versus *while* 1/2

- ▶ Este programa é mais compacto do que o equivalente usando *while* (menos 3 linhas no ciclo de impressão):

```
1 // mostra-num-for.c
2 #include <stdio.h>
3 int main() // Programa que mostra os números 1..n
4 {
5 int i, n = -1; // => entrar no ciclo
6 while(n <= 0)
7 {
8 printf("De-me um inteiro maior que zero: ");
9 scanf("%d", &n);
10 } // end while
11
12 for(i = 1; i <= n ; i = i+1)
13 printf("%d\n", i);
14
15 return 0;
16 }
```

# Ciclo for e várias instruções de inicialização e/ou iteração

Novamente, retomando o exemplo anterior

- ▶ No ciclo `for` em [1] a instrução de iteração é designada por *pós-instrução*.
- ▶ É possível inicializar mais do que uma variável, separando essas inicializações por vírgula; o mesmo se aplica se forem necessárias várias instruções de iteração.
- ▶ Não esquecer que `;` separa os 3 blocos: inicialização, condição e iteração.

```
1 // somatorio-do-while-for.c
2 #include <stdio.h>
3 int main() // Programa que calcula \sum_{i=1}^n i, i > 0
4 { int n, i, soma;
5
6 while(n <= 0) ;
7 {
8 printf("De-me um inteiro maior que zero: ");
9 scanf("%d", &n);
10 }
11
12 // Valor inicial de soma e de contador dentro do for
13 for(soma = 0, i = 1; i <= n ; i= i+1)
14 soma = soma + i; // soma <- 1 + ... + i
15
16 printf("A soma de 1 ate %d: %d", n, soma);
17 putchar('\n'); /* envia um único char para a consola */
18 return 0;
19 }
```

# Ciclos: a instrução *do-while*

- ▶ Em alguns problemas pretendemos **executar um dado ciclo pelo menos uma vez** (Exemplo: ciclo de leitura de um inteiro estritamente positivo). Se utilizarmos um ciclo ***while***, isso implica que temos de **Forçar/Garantir a entrada no ciclo *while***.

# Ciclos: a instrução *do-while*

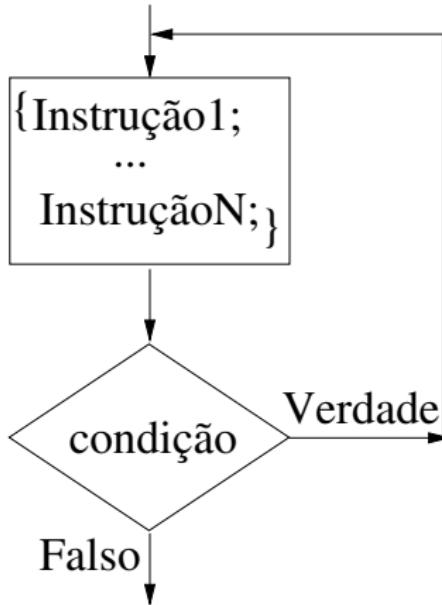
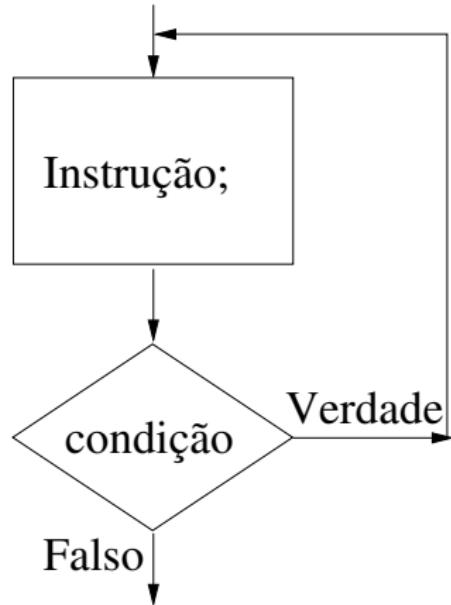
- ▶ Em alguns problemas pretendemos **executar um dado ciclo pelo menos uma vez** (Exemplo: ciclo de leitura de um inteiro estritamente positivo). Se utilizarmos um ciclo **while**, isso implica que temos de **Forçar/Garantir a entrada no ciclo while**.
- ▶ Isso, em geral, significa que deveríamos ter utilizado um ciclo ***do-while***, no qual **o corpo do ciclo é sempre executado pelo menos uma vez!**
- ▶ A forma geral é:

```
do
 instrução do corpo;
 while (condição);
```

```
do {
 instrução 1 do corpo;
 instrução 2 do corpo;
 ...
 instrução N do corpo;
} while (condição);
```

# Ciclos: a instrução *do-while*

Fluxograma



# Ciclos: a instrução *do-while*

Retomando um exemplo anterior (e introduz *puts*)

```
#include <stdio.h>

int main() {
 char opcao;
 do
 {
 printf("\nc/C - Comprar\nv/V - Vender\nnt/T - Trocar\ns/S - Sair");
 printf("\nQual a sua escolha: ");
 scanf(" %c", &opcao); //Note o espaço antes do %c
 switch (opcao)
 {
 case 'c':
 case 'C': puts("*Vai comprar!"); break; //Sai do switch;
 case 'v':
 case 'V': puts("*Vai vender!"); break; //Sai do switch;
 case 't':
 case 'T': puts("*Vai trocar!"); break; //Sai do switch;
 case 's':
 case 'S': puts("*Abandona!"); break; //Sai do switch;
 default: puts("*Opção invalida!"); break; //Sai do switch;
 }
 } while(opcao != 's' && opcao != 'S'); // s ou S é para sair
 return 0;
}
```

# Ciclos: a instrução *do-while*

Retomando o exemplo anterior

```
1 // somatorio-do-while-for.c
2 #include <stdio.h>
3 int main() // Programa que calcula \sum_{i=1}^n i, i > 0
4 {
5 int n, i, soma;
6
7 do // Le número de parcelas
8 {
9 puts("De-me um inteiro maior que zero:");
10 scanf("%d", &n);
11 } while(n <= 0) ;
12
13
14 soma = 0; // Valor inicial de soma: elemento neutro da adição
15 for(i = 1; i <= n ; i= i+1)
16 soma = soma + i; // soma <- 1 + ... + i
17
18 printf("A soma de 1 ate %d: %d", n, soma);
19 putchar('\n'); /* envia um único char para a consola */
20 return 0;
21 }
```

# Ciclos: a instrução *do-while*

Retomando o exemplo anterior - mostrando

```
// somatorio-do-while-for-verbose.c
#include <stdio.h>
int main() // Programa que calcula \sum_{i=1}^n i, i > 0
{
 int n, i, soma;

 do // Le o número de parcelas
 {
 puts("De-me um inteiro maior que zero: ");
 scanf("%d", &n);
 } while(n <= 0) ;

 soma = 0; // Valor inicial de soma: elemento neutro da adição
 for(i = 1; i <= n ; i= i+1)
 {
 printf("\nSoma = %d e vou adicionar i = %d", soma, i);
 soma = soma + i; // soma <- 1 + ... + i
 } // end for
 printf("\nNo final do ciclo i vale %d\n", i);
 printf("\nA soma de 1 ate %d: %d", n, soma);
 putchar('\n'); /* envia um único char para a consola */
 return 0;
}
```

```
De-me um inteiro maior que zero:
4
```

```
Soma = 0 e vou adicionar i = 1
Soma = 1 e vou adicionar i = 2
Soma = 3 e vou adicionar i = 3
Soma = 6 e vou adicionar i = 4
No final do ciclo i vale 5
```

```
A soma de 1 ate 4: 10
```

# A instrução *break*

- ▶ Já vimos que a instrução *break* pode ser utilizada para interromper a sequência de instruções de um *switch* e terminar a sua execução.

# A instrução *break*

- ▶ Já vimos que a instrução *break* pode ser utilizada para interromper a sequência de instruções de um *switch* e terminar a sua execução.
- ▶ Agora ficamos a saber que um *ciclo* pode ser terminado utilizando a instrução *break*.

# A instrução *break*

- ▶ Já vimos que a instrução *break* pode ser utilizada para interromper a sequência de instruções de um *switch* e terminar a sua execução.
- ▶ Agora ficamos a saber que um *ciclo* pode ser terminado utilizando a instrução *break*.

## *break, ciclos e switch*

Um *break* dentro de um *switch* que esteja dentro de um ciclo, não interrompe o ciclo onde se encontra o *switch*, apenas termina a execução do *switch*.

# Exemplo: A instrução *break* 1/2

*break* de ciclo eterno e ciclos encadeados

```
//break-while-true.c
#include <stdio.h>
#include <stdbool.h>
int main() // Programa que calcula \sum_{i=1}^n i
{
 int n, i, soma; // soma <- 1+2+...+n
 //Ciclo infinito que só termina com break
 while (true)
 {
 do // ciclo de leitura do n>=0
 {
 printf("De-me um inteiro maior que zero, ");
 printf("ou zero para terminar: ");
 scanf("%d", &n);
 if(n < 0) puts("Atenção: deu um número negativo.");
 } while(n < 0);
 if (n == 0) break; // Sai do ciclo pois n == 0
 // Calcula somatório
 for(soma = 0, i = 1; i <= n; i = i+1)
 soma = soma + 1;
 //Apresenta resultado corrente
 printf("A soma de 1 até %d vale %d\n\n", n, soma);
 }
 return 0;
}
```

# Exemplo: A instrução *break* 2/2

*break* de ciclo eterno e ciclos encadeados

Exemplo de funcionamento do programa anterior:

```
De-me um inteiro maior que zero, ou zero para terminar: 6
A soma de 1 até 6 vale 21
```

```
De-me um inteiro maior que zero, ou zero para terminar: -8
Atenção: deu um número negativo.
```

```
De-me um inteiro maior que zero, ou zero para terminar: 8
A soma de 1 até 8 vale 36
```

```
De-me um inteiro maior que zero, ou zero para terminar: 0
```

# A instrução *continue*

Exemplo com o ciclo *while*

- A sequência de execução de um ciclo *while* ou *do-while* pode ser interrompida, voltando à condição de controlo usando *continue*:

```
//continue-while.c
#include <stdio.h>

int main()
{
 int i = -1; // Contador

 // Mostra todos os numeros de 0 ate' 10, excepto 5
 while(i < 10)
 {
 i = i + 1; // Passa ao seguinte antes de continue

 if (5 == i) continue; // se i==5 => instrucao controlo

 printf("%d ", i); // mostra numero seguido de espaço
 }
 // muda de linha no fim
 putchar('\n');
 return 0;
}
```

```
0 1 2 3 4 6 7 8 9 10
```

# A instrução *continue*

Exemplo com o ciclo *for*

- ▶ A declaração de *i* no elemento de controlo do *for* foi introduzido no C99.
- ▶ Caso um *continue* ocorra num ciclo *for*, este interrompe a execução do corpo do ciclo, passa à instrução de iteração e depois vai para a condição de controlo do ciclo:

```
//continue-for.c
#include <stdio.h>

int main()
{
 // Mostra todos os numeros de 0 ate' 10, excepto 5
 for(int i = 0; i < 10 ; i = i + 1) // declaração de i, C99, C11
 {
 if (5 == i) continue; // i==5 => instrução de iteração

 printf("%d ", i); // mostra número seguido de espaço
 }
 // muda de linha no fim
 putchar('\n');
 return 0;
}
```

```
0 1 2 3 4 6 7 8 9 10
```

# Ciclos infinitos

- ▶ Como já vimos é possível criar ciclos eternos – que em geral devem ser evitados!!!
- ▶ Exemplos de ciclos eternos (sem incluir *stdbool.h* não podemos usar *true*):

```
while (1)
 instrução;
```

```
for(; ;) // AQUI a condição vazia
 instrução; // tem valor Verdade!!!
```

```
do
 instrução;
while (1);
```

# Operadores Unários de atribuição: ++ e --

- ▶ Já vimos que uma operação frequente é incrementar ou decrementar de uma unidade uma variável.
- ▶ O C tem dois operadores unários de atribuição:

| Operador | Significado     | Exemplos               |
|----------|-----------------|------------------------|
| ++       | incremento de 1 | <code>i++; ++k;</code> |
| --       | decremento de 1 | <code>i--; --k;</code> |

# Operadores Unários de atribuição: ++ e --

- ▶ Já vimos que uma operação frequente é incrementar ou decrementar de uma unidade uma variável.
- ▶ O C tem dois operadores unários de atribuição:

| Operador | Significado     | Exemplos               |
|----------|-----------------|------------------------|
| ++       | incremento de 1 | <code>i++; ++k;</code> |
| --       | decremento de 1 | <code>i--; --k;</code> |

- ▶ Quando escrevemos **numa instrução isolada** ++ à esquerda ou à direita do nome da variável

`i++;      ++k;`

o resultado final mesmo: *i* e *k* são ambos incrementados de uma unidade. Ou seja:

| Operador | Exemplos                  | Equivalente             |
|----------|---------------------------|-------------------------|
| ++       | <code>i++; OU ++i;</code> | <code>i = i + 1;</code> |
| --       | <code>i--; OU --i;</code> | <code>i = i - 1;</code> |

# Operadores Unários de atribuição: ++ e --

Diferença entre pré e pós incremento/decremento

- ▶ Pós-incremento: operador à direita da variável (op. sufixo) `i++`
- ▶ Pré-incremento: operador à esquerda da variável (op. prefixo) `++i`

| $y = x++;$                                                                                                                                                                            | $y = ++x;$                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é atribuído a y</li><li>2. o valor de x é incrementado de 1</li></ol> | <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é incrementado de 1</li><li>2. o valor de x é atribuído a y</li></ol> |

# Operadores Unários de atribuição: ++ e --

Diferença entre pré e pós incremento/decremento

- ▶ Pós-incremento: operador à direita da variável (op. sufixo) i++
- ▶ Pré-incremento: operador à esquerda da variável (op. prefixo) ++i

| $y = x++;$                                                                                                                                                                            | $y = ++x;$                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é atribuído a y</li><li>2. o valor de x é incrementado de 1</li></ol> | <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é incrementado de 1</li><li>2. o valor de x é atribuído a y</li></ol> |

- ▶ Pós-decremento: prefixo à direita da variável (operador sufixo) i--
- ▶ Pré-decremento: op. à esquerda da variável (op. prefixo) --i

| $y = x--;$                                                                                                                                                                            | $y = --x;$                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é atribuído a y</li><li>2. o valor de x é decrementado de 1</li></ol> | <p>Corresponde à sequência,<br/><b>por esta ordem</b></p> <ol style="list-style-type: none"><li>1. o valor de x é decrementado de 1</li><li>2. o valor de x é atribuído a y</li></ol> |

# Operadores Unários de atribuição: ++ e --

Exemplo com pré e pós incremento

```
//operador-unario.c
#include <stdio.h>
int main()
{
 int i, k;
 i = 5;
 printf("i = %d\n", i);

 k = i++;
 printf("i = %d e k = %d \n", i, k);

 k = ++i;
 printf("i = %d e k = %d \n", i, k);

 return 0;
}
```

```
i = 5
i = 6 e k = 5
i = 7 e k = 7
```

# Operadores Unários de atribuição: ++ e --

Observações acerca do pré e pós incremento

## Mais do que uma vez

Não usar estes operadores em variáveis que apareçam mais do que uma vez na mesma expressão, pois o resultado é indefinido.

**Não escrever:**

```
printf("d, %d", i++, --i);
printf("d", i++ + ++i);
```

Mas pode escrever: `printf("d, %d", i++, --k);`

## Numa condição

Se utilizar algum destes operadores numa condição (de um *if-else* ou de um ciclo) a operação de incremento (ou decremento, dependendo do operador) é executada independentemente do valor da condição ser falso ou verdade.

# Operadores binários de atribuição: <op>=

- A linguagem C permite escrever de forma compacta atribuições em que o valor final da variável depende do valor anterior da mesma:

`+ =`   `- =`   `* =`   `/ =`   `% =`

e outros que serão introduzidos mais adiante.

| Exemplo                 | Significado                  |
|-------------------------|------------------------------|
| <code>i += 1;</code>    | <code>i = i + 1;</code>      |
| <code>i -= 11+4;</code> | <code>i = i - (11+4);</code> |
| <code>i *= 5;</code>    | <code>i = i * 5;</code>      |
| <code>i /= 5;</code>    | <code>i = i / 5;</code>      |
| <code>i %= 5;</code>    | <code>i = i % 5;</code>      |

## Sintaxe

O operador (+,-,\*,/,%, etc) e o sinal de igual têm de estar seguidos sem qualquer espaço.

# Resumo: os ciclos

- ▶ Dispomos de 3 ciclos: **while**, **for** e **do-while**. O corpo de cada ciclo é executado enquanto a condição que o controla for verdadeira.
  - ▶ No caso dos ciclos **while** e **for** a condição é testada antes da execução do corpo do ciclo:
  - ▶ No caso do ciclo **do-while** a condição só é testada depois de executado, pelo menos uma vez, o corpo do ciclo.
  - ▶ Dentro de um ciclo pode estar uma instrução simples, um bloco de instruções, ou outro ciclo (um ciclo encadeado)
- ▶ Podem existir ciclos **infinitos**.

# Resumo: os ciclos

- ▶ Dispomos de 3 ciclos: **while**, **for** e **do-while**. O corpo de cada ciclo é executado enquanto a condição que o controla for verdadeira.
  - ▶ No caso dos ciclos **while** e **for** a condição é testada antes da execução do corpo do ciclo;
  - ▶ No caso do ciclo **do-while** a condição só é testada depois de executado, pelo menos uma vez, o corpo do ciclo.
  - ▶ Dentro de um ciclo pode estar uma instrução simples, um bloco de instruções, ou outro ciclo (um ciclo encadeado)
- ▶ Podem existir ciclos **infinitos**.
- ▶ A sequência normal de um ciclo pode ser interrompida, recorrendo às instruções **break** e **continue**:
  - ▶ o **break** termina o ciclo, continuando o programa na instrução que se segue ao ciclo;
  - ▶ o **continue** termina a iteração corrente e faz com o ciclo passe à iteração seguinte.

## Resumo: operadores

- ▶ Os operadores para incremento ( `++` ) e decremento ( `--` ) podem ser utilizados à esquerda e à direita de uma variável, mas o resultados são diferentes se esta não se encontrar isolada:
  - ▶ **Pré-incremento:** quando `++` é colocado **à esquerda** do nome da variável, a variável é incrementada **antes** de ser usada na expressão onde surge.
  - ▶ **Pré-decremento:** quando `--` é colocado **à esquerda** do nome da variável, a variável é decrementada **antes** de ser usada na expressão onde surge.

# Resumo: operadores

- ▶ Os operadores para incremento ( `++` ) e decremento ( `--` ) podem ser utilizados à esquerda e à direita de uma variável, mas o resultados são diferentes se esta não se encontrar isolada:
  - ▶ **Pré-incremento:** quando `++` é colocado **à esquerda** do nome da variável, a variável é incrementada **antes** de ser usada na expressão onde surge.
  - ▶ **Pré-decremento:** quando `--` é colocado **à esquerda** do nome da variável, a variável é decrementada **antes** de ser usada na expressão onde surge.
  - ▶ **Pós-incremento:** quando `++` é colocado **à direita** do nome da variável, a variável é incrementada **depois** do seu valor ter sido usado na expressão onde surge.
  - ▶ **Pós-decremento:** quando `--` é colocado **à direita** do nome da variável, a variável é decrementada **depois** do seu valor ter sido usado na expressão onde surge.

# Resumo: operadores

- ▶ Os operadores para incremento ( `++` ) e decremento ( `--` ) podem ser utilizados à esquerda e à direita de uma variável, mas o resultados são diferentes se esta não se encontrar isolada:
  - ▶ **Pré-incremento:** quando `++` é colocado **à esquerda** do nome da variável, a variável é incrementada **antes** de ser usada na expressão onde surge.
  - ▶ **Pré-decremento:** quando `--` é colocado **à esquerda** do nome da variável, a variável é decrementada **antes** de ser usada na expressão onde surge.
  - ▶ **Pós-incremento:** quando `++` é colocado **à direita** do nome da variável, a variável é incrementada **depois** do seu valor ter sido usado na expressão onde surge.
  - ▶ **Pós-decremento:** quando `--` é colocado **à direita** do nome da variável, a variável é decrementada **depois** do seu valor ter sido usado na expressão onde surge.
- ▶ **Atribuição composta:** forma compacta de fazer uma atribuição quando o valor final de uma variável depende do seu valor anterior: `+=`,  
`-=`, `*=`, `/=`, `%=`.

# Observações

## Atribuições onde?

- ▶ Embora as atribuições possam ocorrer em vários pontos de uma instrução, não é (em geral) boa prática utilizar mais do que uma atribuição numa só expressão.

# Observações

## Atribuições onde?

- ▶ Embora as atribuições possam ocorrer em vários pontos de uma instrução, não é (em geral) boa prática utilizar mais do que uma atribuição numa só expressão.
- ▶ `i = j = k = 3; // k <- 3, j <- k, i <- j`

# Observações

## Atribuições onde?

- ▶ Embora as atribuições possam ocorrer em vários pontos de uma instrução, não é (em geral) boa prática utilizar mais do que uma atribuição numa só expressão.
- ▶ `i = j = k = 3; // k <- 3, j <- k, i <- j`
- ▶ **A evitar:**

```
liquido = iliquido - (imposto = iliquido * taxa);
```

**Fica mais claro:**

```
imposto = iliquido * taxa;
liquido = iliquido - imposto;
```

# Observações

## Atribuições onde?

- ▶ Embora as atribuições possam ocorrer em vários pontos de uma instrução, não é (em geral) boa prática utilizar mais do que uma atribuição numa só expressão.
- ▶ `i = j = k = 3; // k <- 3, j <- k, i <- j`
- ▶ **A evitar:**

```
liquido = iliquido - (imposto = iliquido * taxa);
```

**Fica mais claro:**

```
imposto = iliquido * taxa;
liquido = iliquido - imposto;
```

- ▶ **A evitar:**
- ▶ 

```
while ((pos = base + indice) < 20) {
 cout << "\nVou no " << pos;
 ...
}
```

**Fica mais claro:**

```
while (true) {
 pos = base + indice;
 if(pos >= 20) break;
 cout << "\nVou no " << pos;
 ...
}
```

# Programação de Computadores

## Capítulo 6: Funções



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a  
Professora Teresa Martinez do DEEC-UC.

(PdC)

# Plano: Funções

1. Alcance das variáveis
2. Tempo de existência
3. Funções
4. Parâmetros
5. O corpo da função
6. Funções que retornam um valor
7. Onde colocar as funções
8. Variáveis locais
9. Como programar?
10. Exemplos

Bibliografia:  
Capítulo 5 de [1]

# Alcance das variáveis

- ▶ Até agora temos sempre utilizado variáveis **locais**, ou seja que estão dentro de um bloco (o corpo da função *main*).
- ▶ Um **bloco** tem início numa  e termina na  correspondente.

# Alcance das variáveis

- ▶ Até agora temos sempre utilizado variáveis **locais**, ou seja que estão dentro de um bloco (o corpo da função *main*).
- ▶ Um **bloco** tem início numa **{** } e termina na **}** correspondente.
- ▶ **O alcance (ou âmbito) de uma variável** é a área do programa em que ela é válida.
  - ▶ Uma **variável global** é válida desde o ponto em que é declarada (fora de qualquer bloco) até final do programa.
  - ▶ Uma **variável local** é apenas válida no bloco em que está declarada (e a partir da instrução de declaração).
  - ▶ No caso do ciclo **for**, **se na instrução inicial for declarada uma variável**, essa variável é **local a esse ciclo for**: só tem alcance (e existência) nesse ciclo **for**.

# Alcance das variáveis

- É possível declarar uma variável local com o mesmo nome que uma variável global.  
Nesse caso a **declaração da variável local toma precedência sobre a variável global**, que fica escondida.

# Alcance das variáveis

- ▶ É possível declarar uma variável local com o mesmo nome que uma variável global.  
Nesse caso a **declaração da variável local toma precedência sobre a variável global**, que fica escondida.
- ▶ Dentro de um bloco, contido noutro bloco, podemos igualmente definir variáveis.
- ▶ Se uma variável local tem o mesmo nome doutra variável local num bloco mais externo, a **última declaração** (no respectivo bloco) **toma precedência sobre a anterior**. E assim sucessivamente.

# Alcance das variáveis

Exemplo)

```
// alcance.c
#include <stdio.h>
int g = 0; //variavel global - ''PROIBIDAS!!!
int main()
{
 printf("g = %d é o g global",g);
 int g = 1; // variavel local; esconde g global
 printf("\ng = %d é o g local\n",g);
 int loc = 2; // outra variavel local
 {
 int mloc = 5; // variavel muito local
 loc = 3;
 // Mostra variaveis visiveis neste bloco
 printf("\ng = %d\t\tnloc = %d",g,loc);
 printf("\t\tmloc = %d", mloc);
 } // mloc deixa de existir
 printf("\ng = %d\t\tloc = %d",g,loc);
 // No main, g local esconde g global
 return 0;
}
```

```
g = 0 é o g global
g = 1 é o g local

g = 1 loc = 3 mloc = 5
g = 1 loc = 3
```

# Tempo de existência

- ▶ Uma variável pode ser **permanente ou temporária**.  
As variáveis globais são sempre permanentes.  
As variáveis globais residem numa zona de memória designada por *heap*.

# Tempo de existência

- ▶ Uma variável pode ser **permanente ou temporária**.  
**As variáveis globais são sempre permanentes.**  
As variáveis globais residem numa zona de memória designada por **heap**.
- ▶ As variáveis temporárias, criadas no início de cada bloco reservam espaço numa zona de memória designada por **stack** (a pilha).  
**As variáveis locais são por omissão “automáticas”,** ou seja são criadas sempre que esse bloco é executado e só existem durante a execução desse bloco.

# Tempo de existência

- ▶ Uma variável pode ser **permanente ou temporária**.  
As variáveis globais são sempre permanentes.  
As variáveis globais residem numa zona de memória designada por **heap**.
- ▶ As variáveis temporárias, criadas no início de cada bloco reservam espaço numa zona de memória designada por **stack** (a pilha).  
As variáveis locais são por omissão “automáticas”, ou seja são criadas sempre que esse bloco é executado e só existem durante a execução desse bloco.

---

**Nota:** Existem também variáveis **estáticas**, que têm existência **permanente** e âmbito **local**, mas estas serão introduzidas mais à frente.

# Ciclo for: alcance e existência de variáveis

- ▶ A instrução `for` é semelhante a um par de chavetas, na qual se podem declarar variáveis.
- ▶ O alcance dessas variáveis vai desde o início do `for` até ao final da instrução (incluindo a instrução ou bloco controlado pelo `for`).

```
#include <stdio.h>
int main() // Programa que calcula \sum_{i=1}^n i
{
 int n, soma; // Limite e valor do somatorio;
 do {
 printf("De-me um inteiro maior que zero: ");
 scanf("%d", &n);
 } while (n <= 0); // Repete enquanto n<=0

 // Calcula a soma, começando com o valor neutro da adição
 soma = 0; // O alcance de i limita-se ao for
 for (int i = 1 ; i <= n; ++i) // Só possível em modo C99 ou C11
 soma += i; // soma <- 1 + ... + i

 // A variável i terminou a sua existência
 printf("A soma de 1 ate %d vale %d\n", n, soma);
 return (0);
}
```

# Funções

## Introdução

- ▶ Embora sem saber escrever uma função, temos utilizados funções do sistema, *printf*, *scanf*, *getchar*, *putchar*

# Funções

## Introdução

- ▶ Embora sem saber escrever uma função, temos utilizados funções do sistema, *printf*, *scanf*, *getchar*, *putchar*
- ▶ Vamos agora ver como definir **funções e procedimentos**, como passar parâmetros, e como retornar algum valor usando uma função.
- ▶ Designaremos por *procedimento* uma função que não retorna nenhum valor.

# Funções

## Introdução

**Problema:** Escreva uma programa que apresente no ecrã o seguinte texto. Utilize ciclos **for**.

```

Números entre 1 e 5

1
2
3
4
5

```

```
1 //prog0501.c [1] alterado
2 #include <stdio.h>
3 int main()
4 {
5 int i;
6 /* Escrita do Cabeçalho */
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 puts("Números entre 1 e 5");
11 for (i=1 ; i<=20 ; i++)
12 putchar('*');
13 putchar('\n');
14
15 /* Escrita dos Números */
16 for (i=1; i<=5 ; i++)
17 printf("%d\n",i);
18
19 /* Escrita de linha de * */
20 for (i=1 ; i<=20 ; i++)
21 putchar('*');
22 putchar('\n');
23 return (0);
24 }
```

# Funções

## Introdução

**Problema:** Escreva uma programa que apresente no ecrã o seguinte texto. Utilize ciclos *for*.

```

Números entre 1 e 5

1
2
3
4
5

```

- ▶ De notar os 3 ciclos *for* repetidos: linhas 7-8, 11-12 e 20-21.
- ▶ Solução: escrever esse código *uma única vez*, escutando essa tarefa quando necessário.

```
1 //prog0501.c [1] alterado
2 #include <stdio.h>
3 int main()
4 {
5 int i;
6 /* Escrita do Cabeçalho */
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 puts("Números entre 1 e 5");
11 for (i=1 ; i<=20 ; i++)
12 putchar('*');
13 putchar('\n');
14
15 /* Escrita dos Números */
16 for (i=1; i<=5 ; i++)
17 printf("%d\n",i);
18
19 /* Escrita de linha de * */
20 for (i=1 ; i<=20 ; i++)
21 putchar('*');
22 putchar('\n');
23 return (0);
24 }
```

# Funções

## Introdução

**Problema:** Escreva uma programa que apresente no ecrã uma linha com 20 asteriscos.  
Utilize ciclos *for*.

```
1 //prog0502.c [1] alterado
2 #include <stdio.h>
3
4 int main()
5 {
6 int i;
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 return (0);
11 }
```

- ▶ *void* é o tipo que significa “nada” ou seja ausência de valor retornado pela função.<sup>2</sup>
- ▶ Este ficheiro não pode originar um ficheiro executável porque não tem a função *main*.

```
1 //prog0502b.c [1] alterado
2 #include <stdio.h>
3
4 void linha()
5 {
6 int i;
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 }
```

---

<sup>2</sup>Em [1] o cabeçalho da função *linha* é dado sem tipo (o que por omissão equivale a *int*). Mas em *C11* as funções devem ter um *tipo explícito*.

# Funções

## Introdução

**Problema:** Escreva uma programa que apresente no ecrã uma linha com 20 asteriscos.  
Utilize ciclos *for*.

```
1 //prog0502.c [1] alterado
2 #include <stdio.h>
3
4 int main()
5 {
6 int i;
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 return (0);
11 }
```

- ▶ *void* é o tipo que significa “nada” ou seja ausência de valor retornado pela função.<sup>2</sup>
- ▶ Este ficheiro não pode originar um ficheiro executável porque não tem a função *main*.

```
1 //prog0502b.c [1] alterado
2 #include <stdio.h>
3
4 void linha()
5 {
6 int i;
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 }
```

## A função *main*

Um programa em linguagem C tem de possuir sempre a função *main*.

---

<sup>2</sup>Em [1] o cabeçalho da função *linha* é dado sem tipo (o que por omissão equivale a *int*). Mas em *C11* as funções devem ter um *tipo explícito*.

# Funções

## O meu primeiro programa com uma função

```
1 // prog0503.c [2] alterado
2 #include <stdio.h>
3
4 void linha() /* sem tipo, em C89 devolveria um int */
5 {
6 int i; /* variável local a este bloco */
7 for (i=1 ; i<=20 ; i++)
8 putchar('*');
9 putchar('\n');
10 }
11
12 int main()
13 {
14 int i; /* variável local a este bloco (!= do i de linha)*/
15
16 linha(); /* Escreve uma linha de asteriscos */
17 puts("Números entre 1 e 5");
18 linha(); /* Escreve outra linha de asteriscos */
19
20 for (i=1; i<=5 ; i++)
21 printf("%d\n",i);
22
23 linha();/* Escreve a última linha de asteriscos */
24 return 0;
25 }
```

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- ▶ Uma função pode ser **invocada** a partir de outras funções.

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- ▶ Uma função pode ser **invocada** a partir de outras funções.
- ▶ Uma função deve fazer **uma tarefa bem definida** (o seu nome deve ser informativo).

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- ▶ Uma função pode ser **invocada** a partir de outras funções.
- ▶ Uma função deve fazer **uma tarefa bem definida** (o seu nome deve ser informativo).
- ▶ Uma função deve comportar-se como uma **caixa negra** (não interessa à entidade que a chama saber como foi implementada): deve executar a tarefa para a qual foi implementada, sem efeitos colaterais.

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- ▶ Uma função pode ser **invocada** a partir de outras funções.
- ▶ Uma função deve fazer **uma tarefa bem definida** (o seu nome deve ser informativo).
- ▶ Uma função deve comportar-se como uma **caixa negra** (não interessa à entidade que a chama saber como foi implementada): deve executar a tarefa para a qual foi implementada, sem efeitos colaterais.
- ▶ Uma função pode **receber argumentos** que determinam o seu comportamento.

# Características de uma função

- ▶ Cada função tem um **nome único**, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- ▶ Uma função pode ser **invocada** a partir de outras funções.
- ▶ Uma função deve fazer **uma tarefa bem definida** (o seu nome deve ser informativo).
- ▶ Uma função deve comportar-se como uma **caixa negra** (não interessa à entidade que a chama saber como foi implementada): deve executar a tarefa para a qual foi implementada, sem efeitos colaterais.
- ▶ Uma função pode **receber argumentos** que determinam o seu comportamento.
- ▶ Uma função pode **retornar**, para a função que a invocou, um valor de saída.

# Funcionamento uma função

- ▶ O código de uma função só é executado quando a função é chamada.

# Funcionamento uma função

- ▶ O código de uma função só é executado quando a função é chamada.
- ▶ Se uma função é invocada, a execução do programa que a invoca é temporariamente suspenso, e passa ser executado o corpo dessa função.

# Funcionamento uma função

- ▶ O código de uma função só é executado quando a função é chamada.
- ▶ Se uma função é invocada, a execução do programa que a invoca é temporariamente suspenso, e passa ser executado o corpo dessa função. Uma vez terminada a sua execução o programa que a chamou retoma a execução na instrução que se segue à invocação da função chamada.

# Funcionamento uma função

- ▶ O código de uma função só é executado quando a função é chamada.
- ▶ Se uma função é invocada, a execução do programa que a invoca é temporariamente suspenso, e passa ser executado o corpo dessa função. Uma vez terminada a sua execução o programa que a chamou retoma a execução na instrução que se segue à invocação da função chamada.
- ▶ O programa que invoca uma função pode enviar argumentos que são recebidos pela função, os quais são armazenados em variáveis locais inicializadas com o valor desses argumentos.

# Funcionamento uma função

- ▶ O código de uma função só é executado quando a função é chamada.
- ▶ Se uma função é invocada, a execução do programa que a invoca é temporariamente suspenso, e passa ser executado o corpo dessa função. Uma vez terminada a sua execução o programa que a chamou retoma a execução na instrução que se segue à invocação da função chamada.
- ▶ O programa que invoca uma função pode enviar argumentos que são recebidos pela função, os quais são armazenados em variáveis locais inicializadas com o valor desses argumentos.
- ▶ Ao terminar uma função pode retornar um valor para o programa que a invocou.

# Chamada de várias funções sem argumentos

Ilustrando através de um exemplo

**Problema:** Escreva uma programa que apresente no ecrã o seguinte padrão. Use 3 variantes da função linha.

```



```

```
1 // prog504.c [1] alterado
2 #include <stdio.h>
3
4 void linha3x()
5 {
6 for (int i=1 ; i<=3 ; i++)
7 putchar('*');
8 putchar('\n');
9 }
10
11 void linha5x()
12 {
13 for (int i=1 ; i<=5 ; i++)
14 putchar('*');
15 putchar('\n');
16 }
17
18 void linha7x()
19 {
20 for (int i=1 ; i<=7 ; i++)
21 putchar('*');
22 putchar('\n');
23 }
24
25 int main()
26 {
27 linha3x();
28 linha5x();
29 linha7x();
30 return(0);
31 }
```

# Chamada de várias funções sem argumentos

Ilustrando através de um exemplo

**Problema:** Escreva uma programa que apresente no ecrã o seguinte padrão. Use 3 variantes da função linha.

```



```

- ▶ Observando o código à direita, verifica-se que as três funções diferem apenas no valor da condição do ciclo *for*!
- ▶ Com uma única função que tivesse como parâmetro de entrada o número de asteriscos a escrever numa linha, o *main* passaria a conter:  
*linha(3); linha(5); linha(7);*

```
1 // prog504.c [1] alterado
2 #include <stdio.h>
3
4 void linha3x()
5 {
6 for (int i=1 ; i<=3 ; i++)
7 putchar('*');
8 putchar('\n');
9 }
10
11 void linha5x()
12 {
13 for (int i=1 ; i<=5 ; i++)
14 putchar('*');
15 putchar('\n');
16 }
17
18 void linha7x()
19 {
20 for (int i=1 ; i<=7 ; i++)
21 putchar('*');
22 putchar('\n');
23 }
24
25 int main()
26 {
27 linha3x();
28 linha5x();
29 linha7x();
30 return(0);
31 }
```

# Função com parâmetros

Ilustrando através de um exemplo

```
1 // prog505.c [1] alterado
2 #include <stdio.h>
3
4 /* desenha uma linha com
5 n asteriscos e muda de linha
6 */
7 void linha(int n)
8 {
9 for (int i=1 ; i<=n ; i++)
10 putchar('*');
11 putchar('\n');
12 }
13
14 int main()
15 {
16 linha(3);
17 linha(5);
18 linha(7);
19 return(0);
20 }
```

**Nota:** Na referência [1] o cabeçalho das funções *linhx?* e *linha* é dado sem tipo (o que por omissão equivale a *int*). No *C11* as funções devem ter um *tipo explícito*.

- ▶ Qual o nome da função?  
*linha*
- ▶ Qual o tipo da função?  
*void*
- ▶ Quantos parâmetros (formais) tem?  
*1*
- ▶ Qual o tipo do parâmetro?  
*int*
- ▶ Qual o nome da variável que vai armazenar esse parâmetro?  
*n*
- ▶ Qual o cabeçalho da função?  
*void linha(int n)*
- ▶ Qual o corpo da função?

```
for (int i=1 ; i<=n ; i++)
 putchar('*');
putchar('\n');
```

# Parâmetros formais e argumentos reais de uma função 1/2

- ▶ A **comunicação** com uma função faz-se através dos **argumentos** (reais) que lhe são enviados e dos **parâmetros** (formais) que os recebem.  
O **número de parâmetros** pode ser 0 (como em `linha3x`) ou 1 (como em `linha`) ou outro número dependendo apenas das necessidades do programador.

# Parâmetros formais e argumentos reais de uma função 1/2

- ▶ A **comunicação** com uma função faz-se através dos **argumentos** (reais) que lhe são enviados e dos **parâmetros** (formais) que os recebem.  
O **número de parâmetros** pode ser 0 (como em `linha3x`) ou 1 (como em `linha`) ou outro número dependendo apenas das necessidades do programador.
- ▶ Cada **parâmetro** precisa de ter o seu **tipo especificado** no cabeçalho da função.

# Parâmetros formais e argumentos reais de uma função 1/2

- ▶ A **comunicação** com uma função faz-se através dos **argumentos** (reais) que lhe são enviados e dos **parâmetros** (formais) que os recebem.  
O **número de parâmetros** pode ser 0 (como em `linha3x`) ou 1 (como em `linha`) ou outro número dependendo apenas das necessidades do programador.
- ▶ Cada **parâmetro** precisa de ter o seu **tipo especificado** no cabeçalho da função.
- ▶ Os **parâmetros** de uma função **são separados**, um a um, por uma vírgula  e cada um deles deve ser precedido pelo seu tipo, mesmo quando vários são do mesmo tipo.

Correto:

```
void f1(int n, int k, float x) { /* corpo */ }
```

Errado:

```
void f1(int n, k, float x) { /* corpo */ }
```

# Parâmetros formais e argumentos reais de uma função 2/2

## Número dos argumentos

O número dos argumentos reais enviados a para uma função, quando esta é invocada, deve ser igual ao número dos parâmetros formais que constam do cabeçalho da função.

## Tipo dos argumentos

O tipo de cada um dos argumentos enviados a para uma função, quando esta é invocada, deve corresponder ao tipo de cada um dos parâmetros que constam do cabeçalho da função, parâmetro a parâmetro (ou seja pela mesma ordem).

## Valor dos argumentos

Qualquer expressão válida em C pode ser enviada como argumento para uma função (desde que do tipo adequado ao parâmetro correspondente).

**Nota:** É comum chamar parâmetro tanto aos argumentos de invocação (os argumentos reais) como aos parâmetros formais da função (os do cabeçalho).

# Função com vários parâmetros

Ilustrando através de um exemplo

- ▶ Considere que se deseja que a função linha imprima agora qualquer carácter e não apenas um asterisco.
- ▶ Isso significa que a função deverá ter agora dois argumentos: o número de vezes que se deseja repetir o carácter, o qual será também parâmetro da função.

# Função com vários parâmetros

Ilustrando através de um exemplo

- ▶ Considere que se deseja que a função linha imprima agora qualquer carácter e não apenas um asterisco.
- ▶ Isso significa que a função deverá ter agora dois argumentos: o número de vezes que se deseja repetir o carácter, o qual será também parâmetro da função.

```
1 //prog0506.c [1] alterado
2 #include <stdio.h>
3 /* escreve ch num vezes */
4 void linha(int num, char ch)
5 {
6 for (int i=1 ; i<=num ; i++)
7 putchar(ch);
8 putchar('\n');
9 }
10
11 int main()
12 {
13 linha(3, '+');
14 linha(3+2, '&');
15 linha(3+2+2, '#');
16 return(0);
17 }
```

## Sobrecarga do nome da função não é possível em linguagem C

Não é possível no mesmo programa em linguagem C coexistirem duas funções com o mesmo nome, apesar da lista de parâmetros ser diferente.

# Função com vários parâmetros

Ilustrando através de um exemplo

```
1 //prog0506.c [1] alterado
2 #include <stdio.h>
3
4 void linha(int n, char ch)
5 {
6 for (int i=1 ; i<=n ; i++)
7 putchar(ch);
8 putchar('\n');
9 }
10
11 int main()
12 {
13 char c;
14 unsigned int k;
15 printf("De-me um número inteiro sem sinal: ");
16 scanf("%u", &k);
17 printf("De-me um carácter: ");
18 scanf(" %c", &c); /* Note o espaço antes de % */
19
20 /* k e ch são os argumentos reais:
21 * n (local a linha) toma o valor de k (local a main)
22 * ch (local a linha) toma o valor de c (local a main) */
23 linha(k, c); /* com conversão implícita de unsigned int para int */
24
25 /* ch é local a main e independente do ch de linha
26 * são duas variáveis locais cada uma da respetiva função */
27 char ch = c;
28
29 /* k+2 e ch+1 são os argumentos reais:
30 * n (local a linha) toma o valor de k+2, em que k é local a main
31 * ch (local a linha) toma o valor de ch+1, em que ch é (local a main) */
32 linha(k+2, ch+1); /* com conversão implícita de unsigned int para int */
33
34 return(0);
35 }
```

# Função com vários parâmetros

Ilustrando através de um exemplo

O programa anterior, se o utilizador digitar:

5<enter> h <enter>  
produzirá o resultado:

```
De-me um número inteiro sem sinal: 5
De-me um carácter: h
hhhhh
iiiiiii
```

# Função com vários parâmetros

Ilustrando através de um exemplo

O programa anterior, se o utilizador digitar:

5<enter> h <enter>  
produzirá o resultado:

```
De-me um número inteiro sem sinal: 5
De-me um carácter: h
hhhhh
iiiiiii
```

Os parâmetros formais de uma função são variáveis locais dessa função

Os parâmetros formais de uma função são variáveis locais dessa função.  
Uma função apenas tem conhecimento das variáveis que lhe são locais.

**Nota:** Uma função também tem poderá ter acesso a **variáveis globais**, mas a sua utilização está **vedada** nesta unidade curricular!

# O Corpo da função

- ▶ O corpo da função é formado por **instruções da linguagem C**.
- ▶ O corpo da função vem imediatamente a seguir ao cabeçalho da função e está **compreendido entre chavetas**.

## O cabeçalho da função

O cabeçalho da função **nunca** deve ser seguido de um **;**.

# O Corpo da função

- ▶ O corpo da função é formado por **instruções da linguagem C**.
- ▶ O corpo da função vem imediatamente a seguir ao cabeçalho da função e está **compreendido entre chavetas**.

## O cabeçalho da função

O cabeçalho da função **nunca** deve ser seguido de um **;**.

- ▶ Sempre que uma função é invocada o seu corpo é executado, instrução a instrução, até que este **termina**, ou até encontrar a instrução **return**.
- ▶ A instrução **return** quando é encontrada, **termina a execução da função** e volta ao programa que a invocou.

## O corpo da função

O corpo da função pode conter qualquer instrução válida da linguagem C. Em C standard não se podem definir funções dentro de funções e tal é também **vedado** nesta unidade curricular.

# Exemplo: a instrução return termina uma função

Retomando o exemplo do slide 21, e fazendo *return* "fora de sítio":

```
1 //prog0506c.c [1] alterado
2 #include <stdio.h>
3 void linha(int num, char ch)
4 {
5 for (int i=1 ; i<=num ; i++)
6 putchar(ch);
7 putchar('\n');
8 }
9 int main()
10 {
11 linha(3, '+'); // produz linha com +++
12 linha(3+2, '&'); // produz linha com &&&&
13
14 return 7; // Termina main e retorna 7 ao sistema Operativo
15
16 linha(3+2+2, '#'); // ZZZ - Esta função não é executada!!!
17 }
```

```
+++
&&&&&

(program exited with code: 7)
```

# Funções que retornam um valor

- ▶ Uma função, uma vez terminada a sua tarefa poderá devolver **um único resultado**.
- ▶ O A devolução é feita pela instrução *return*. O valor a devolver está escrito logo a seguir à instrução *return*.
- ▶ O tipo do valor retornado é sempre do tipo da função.
- ▶ Uma função pode ser invocada em qualquer expressão ou instrução válida para o tipo devolvido pela função.
- ▶ Uma função pode ser invocada dentro de outra função (diferente do *main*).
- ▶ Uma função pode conter várias instruções *return*, mas no **máximo** será executada **uma delas**.

## Valor retornado

A seguir à instrução *return* pode estar qualquer expressão válida em C (incluindo uma chamada a uma função).

# Exemplo: Invocação de funções que retornam um valor

```
1 // prog0507.c [1] alterado
2 #include <stdio.h>
3
4 int soma(int a, int b) /* Devolve a soma de dois inteiros */
5 {
6 return a+b;
7 }
8
9 int dobro(int k) /* Devolve o dobro de qualquer inteiro */
10 {
11 return 2*k;
12 }
13 int main()
14 {
15 int n,i,total;
16 printf("Introduza dois Números: ");
17 scanf("%d%d",&n,&i);
18 total = soma(n,i); /* Atrib. do result de função a uma var */
19 printf("%d+%d=%d\n",n,i,total);
20 printf("2*%d=%d e 2*%d=%d\n",n,dobro(n),i,dobro(i));
21 if(dobro(i) > n)
22 printf("O dobro de %d é maior que %d\n", i , n);
23 else
24 printf("O dobro de %d é menor ou igual que %d\n", i , n);
25 return(0);
26 }
```

# Exemplo: Invocação de funções que retornam um valor

- ▶ O programa do slide anterior produziria a saída (de acordo com os valores digitados):

```
Introduza dois Números: 78
15
78+15=93
2*78=156 e 2*15=30
O dobro de 15 é menor ou igual que 78
```

# Exemplo: Invocação de funções que retornam um valor

- ▶ O programa do slide anterior produziria a saída (de acordo com os valores digitados):

```
Introduza dois Números: 78
15
78+15=93
2*78=156 e 2*15=30
O dobro de 15 é menor ou igual que 78
```

- ▶ Qual seria o resultado da instrução:

```
printf("%d", soma(soma(1, dobro(5)), 3));
```

caso tivesse sido colocada no *main* no programa anterior?

# Exemplo: Invocação de funções que retornam um valor

- ▶ O programa do slide anterior produziria a saída (de acordo com os valores digitados):

```
Introduza dois Números: 78
15
78+15=93
2*78=156 e 2*15=30
O dobro de 15 é menor ou igual que 78
```

- ▶ Qual seria o resultado da instrução:

```
printf("%d", soma(soma(1, dobro(5)), 3));
```

caso tivesse sido colocada no *main* no programa anterior?

|                           |                                                 |
|---------------------------|-------------------------------------------------|
| dobro(5)                  | vale 10                                         |
| soma(1, dobro(5))         | soma(1,10)                                      |
| soma(soma(1, dobro(5)),3) | soma(soma(1,10),3)      soma(11,3)      vale 14 |

Ou seja surgiria 14 no ecrã.

# O tipo *void*

## Revisitando o tipo *void*

Recorda-se que função `linha` foi declarada *void*, para indicar que não devolve nenhum tipo:

```
1 void linha()
2 {
3 int i;
4 for (i=1 ; i<=20 ; i++)
5 putchar('*');
6 putchar('\n');
7 }
```

# O tipo *void*

## Revisitando o tipo *void*

Recorda-se que função `linha` foi declarada `void`, para indicar que não devolve nenhum tipo:

```
1 void linha()
2 {
3 int i;
4 for (i=1 ; i<=20 ; i++)
5 putchar('*');
6 putchar('\n');
7 }
```

A função `linha` não tem parâmetros, pelo que o tipo `void` poderia ter sido usado para explicitar que não recebe quaisquer argumentos.

```
1 void linha(void)
2 {
3 int i;
4 for (i=1 ; i<=20 ; i++)
5 putchar('*');
6 putchar('\n');
7 }
```

# O tipo *void*

## Revisitando o tipo *void*

Recorda-se que função *linha* foi declarada *void*, para indicar que não devolve nenhum tipo:

```
1 void linha()
2 {
3 int i;
4 for (i=1 ; i<=20 ; i++)
5 putchar('*');
6 putchar('\n');
7 }
```

A função *linha* não tem parâmetros, pelo que o tipo *void* poderia ter sido usado para explicitar que não recebe quaisquer argumentos.

```
1 void linha(void)
2 {
3 int i;
4 for (i=1 ; i<=20 ; i++)
5 putchar('*');
6 putchar('\n');
7 }
```

## Funções *void* e a instrução *return*

Pode terminar-se a execução de uma função que retorna *void* usando a instrução *return*:

## Funções e procedimentos

Em C apenas existem funções. Na Ref. [1] é sugerido que se designe por *procedimento* uma função que retorne *void*.

# Funções que retornam um valor

Assumindo que o programa se encontra em num único ficheiro:

- ▶ As funções podem ser colocadas em qualquer local do ficheiro, desde que seja antes de serem invocadas.
- ▶ Para não precisarmos de estar atentos à ordem precisa pela qual uma função chama outra, podemos em alternativa **declarar** as funções no início do ficheiro. As funções podem em seguida ser **definidas** por qualquer ordem no ficheiro.
- ▶ A **declaração** de uma função consiste na escrita do seu **protótipo** ou **assinatura** o qual não é mais do que **o cabeçalho seguido de um ponto e vírgula**.

---

<sup>3</sup> Na Ref. [1, pág. 177] é dito que o protótipo poderia omitir a lista de argumentos. Essa opção deixou de ser válida desde a norma ANSI de 1989 (ver Ref. [3, pág 26-27]) e atualmente dá **erro de compilação**.

# Funções que retornam um valor

Assumindo que o programa se encontra em num único ficheiro:

- ▶ As funções podem ser colocadas em qualquer local do ficheiro, desde que seja antes de serem invocadas.
- ▶ Para não precisarmos de estar atentos à ordem precisa pela qual uma função chama outra, podemos em alternativa **declarar** as funções no início do ficheiro. As funções podem em seguida ser **definidas** por qualquer ordem no ficheiro.
- ▶ A **declaração** de uma função consiste na escrita do seu **protótipo** ou **assinatura** o qual não é mais do que **o cabeçalho seguido de um ponto e vírgula**.

## Protótipos de uma função

O protótipo<sup>3</sup> de uma função pode ser igual ao cabeçalho seguido de um ponto e vírgula ou dado pelo tipo retornado seguido do nome da função e entre parêntesis curvos apenas os tipos dos parâmetros da função, separados por vírgulas.

---

<sup>3</sup> Na Ref. [1, pág. 177] é dito que o protótipo poderia omitir a lista de argumentos. Essa opção deixou de ser válida desde a norma ANSI de 1989 (ver Ref. [3, pág 26-27]) e atualmente dá **erro de compilação**.

# Exemplo: programa do slide 27, com protótipos

```
1 // prog0507b.c [1] alterado
2 #include <stdio.h>
3
4 /* Devolve a soma de dois inteiros */
5 int soma(int, int); /* protótipo sem parâmetros (A EVITAR: pouco legível!) */
6
7 /* Devolve o dobro de qualquer inteiro */
8 int dobro(int k); /* protótipo com parâmetro */
9
10 int main()
11 {
12 int n,i,total;
13 printf("Introduza dois Números: ");
14 scanf("%d%d", &n,&i);
15 total = soma(n,i); /* Atrib. do result de função a uma var */
16 printf("%d+%d=%d\n",n,i,total);
17 printf("2*%d=%d e 2*%d=%d\n",n,dobro(n),i,dobro(i));
18 if(dobro(i) > n)
19 printf("O dobro de %d é maior que %d\n", i , n);
20 else
21 printf("O dobro de %d é menor ou igual que %d\n", i , n);
22 return(0);
23 }
24 /* Devolve a soma de dois inteiros */
25 int soma(int a, int b)
26 {
27 return a+b;
28 }
29 /* Devolve o dobro de qualquer inteiro */
30 int dobro(int k)
31 {
32 return 2*k;
33 }
```

# Variáveis locais

Como já foi referido, as variáveis declaradas dentro do corpo de uma função só são **visíveis** (ou seja conhecidas) dentro da função.

## Variáveis declaradas dentro do corpo de uma função

Variáveis declaradas dentro do corpo de uma função são denominadas de variáveis **locais** à função.

## Existência das variáveis declaradas dentro do corpo de uma função

Variáveis declaradas dentro do corpo de uma função são **criadas** quando a função é invocada e são **destruídas** quando a função termina. Por esse motivo são também designadas por variáveis **automáticas**.

# Variáveis locais

Como já foi referido, as variáveis declaradas dentro do corpo de uma função só são **visíveis** (ou seja conhecidas) dentro da função.

## Variáveis declaradas dentro do corpo de uma função

Variáveis declaradas dentro do corpo de uma função são denominadas de variáveis **locais** à função.

## Existência das variáveis declaradas dentro do corpo de uma função

Variáveis declaradas dentro do corpo de uma função são **criadas** quando a função é invocada e são **destruídas** quando a função termina. Por esse motivo são também designadas por variáveis **automáticas**.

- ▶ Duas funções distintas podem ter variáveis locais com o **mesmo nome**. Estas são **variáveis distintas** sem qualquer relação entre si, que o compilador distingue pelo local onde são utilizadas.
- ▶ Os efeitos **colaterais** que podem ocorrer quando se usam variáveis globais são de evitar. Por esse motivo nesta unidade curricular estão **proibidas as variáveis globais**.

# Variáveis locais: parâmetros da função

Recordando:

A *comunicação* com uma função faz-se através dos *argumentos* (reais) que lhe são enviados e dos *parâmetros* (formais) que os recebem.

- ▶ Os parâmetros de uma função são variáveis locais dessa função.
- ▶ O valor do argumento real é copiado para o parâmetro da função, e a ligação entre os dois termina nesse momento.
- ▶ Analise-se o programa que se segue.

# Variáveis locais: parâmetros da função

## Exemplo

```
1 #include <stdio.h> /* prog0507.c [1] alterado */
2
3 int dobro(int k) /* Devolve o dobro de qualquer inteiro */
4 {
5 /* para ilustrar a imunidade dos argumentos às alterações nos
6 * parâmetros k é um parâmetro, ou seja é local a dobro */
7 k = 2*k ; /* calcula o dobro de k e armazena em k */
8 return k;
9 }
10 int main()
11 {
12 int n,total;
13 printf("Introduza um Número: ");
14 scanf("%d",&n);
15
16 /* o valor de n é copiado para k; n não é alterado por dobro */
17 total = dobro(n);
18 printf("O dobro de %d vale %d\n", n , total);
19
20 /* o valor de n+2 é copiado para k; n não é alterado por dobro*/
21 total = dobro(n+2);
22 printf("O dobro de %d vale %d\n", n+2, total);
23 return(0);
24 }
```

# Variáveis locais: parâmetros da função

## Exemplo

Um possível resultado do programa anterior seria:

```
Introduza um Número: 5
O dobro de 5 vale 10
O dobro de 7 vale 14
```

Passo a passo o programa do slide anterior.

- ▶ O valor dado pelo utilizador (5), é lido e armazenado na variável *n* - ver linha 14.
- ▶ Na linha 17, o valor de *n* é o argumento real de **dobro** e o parâmetro *k* da função toma o valor 5.
- ▶ A função **dobro** coloca em *k* o valor 10 e termina retornando esse valor à função **main**, que o armazena em total.
- ▶ O valor de *n* (**inalterado**) e de total é apresentado – ver linha 18
- ▶ Na linha 21, o valor de  $n + 2$  é o argumento real de **dobro** e o parâmetro *k* da função toma o valor 7.
- ▶ A função **dobro** coloca em *k* o valor 14 e termina retornando esse valor à função **main**, que o armazena em total.
- ▶ O valor de  $n + 2$  (*n* **inalterado**) e de total é apresentado – ver linha 22

# Programação estruturada

## Funções

As funções são importantes porque:

- ▶ Permitem reduzir a complexidade de um programa.
- ▶ Evitam a repetição de código ao longo do programa:
  - ▶ É mais simples corrigir erros.
  - ▶ O programa executável é de menor dimensão.
- ▶ Contribuem para a boa estruturação de um programa.

# Como Programar? Programação estruturada.

- ▶ Como programar tem sido objecto de estudo da Ciência da Computação.

# Como Programar? Programação estruturada.

- ▶ Como programar tem sido objecto de estudo da Ciência da Computação.
- ▶ Numerosas metodologias têm sido propostas: fluxogramas (já utilizados na disciplina), programação descendente, programação ascendente, programação estruturada e programação orientada aos objectos.

# Como Programar? Programação estruturada.

- ▶ Como programar tem sido objecto de estudo da Ciência da Computação.
- ▶ Numerosas metodologias têm sido propostas: fluxogramas (já utilizados na disciplina), programação descendente, programação ascendente, programação estruturada e programação orientada aos objectos.
- ▶ Agora que já sabemos usar funções podemos falar de *programação estruturada*: estruturar ou dividir um programa em pequenas funções bem definidas.  
Isso simplifica a escrita e a interpretação do programa.

# Como Programar? Programação estruturada.

- ▶ Como programar tem sido objecto de estudo da Ciência da Computação.
- ▶ Numerosas metodologias têm sido propostas: fluxogramas (já utilizados na disciplina), programação descendente, programação ascendente, programação estruturada e programação orientada aos objectos.
- ▶ Agora que já sabemos usar funções podemos falar de *programação estruturada*: estruturar ou dividir um programa em pequenas funções bem definidas.  
Isso simplifica a escrita e a interpretação do programa.
- ▶ A programação estruturada foca-se no código do programa. A fusão do código e dos dados para criar classes e a programação orientada aos objectos só será abordada em Estruturas de Dados e Algoritmos.

# Funções

Programação estruturada: *programação descendente*

- ▶ O primeiro passo em programação é decidir o que se pretende fazer:  
**especificar o problema** a resolver.

# Funções

Programação estruturada: *programação descendente*

- ▶ O primeiro passo em programação é decidir o que se pretende fazer: **especificar o problema** a resolver.
- ▶ Em segundo lugar escolher **as estruturas de dados** que vão ser usadas.

# Funções

Programação estruturada: *programação descendente*

- ▶ O primeiro passo em programação é decidir o que se pretende fazer: **especificar o problema** a resolver.
- ▶ Em segundo lugar escolher **as estruturas de dados** que vão ser usadas.
- ▶ **Finalmente** pode começar-se a escrever o código. Em geral começa-se com um plano (tal como fazemos na composição de um texto). Os detalhes ficam para depois. Chama-se a isto **programação descendente** (*top-down programming*).

# Funções

Programação estruturada: *programação descendente*

- ▶ O primeiro passo em programação é decidir o que se pretende fazer: **especificar o problema** a resolver.
- ▶ Em segundo lugar escolher **as estruturas de dados** que vão ser usadas.
- ▶ **Finalmente** pode começar-se a escrever o código. Em geral começa-se com um plano (tal como fazemos na composição de um texto). Os detalhes ficam para depois. Chama-se a isto **programação descendente** (*top-down programming*).
- ▶ Começa-se por escrever o programa principal, a função `main`, identificando as funções necessárias.  
Depois da função `main` estar pronta, podemos começar a escrever as restantes funções.

# Funções

Programação estruturada: *programação descendente*

- ▶ O primeiro passo em programação é decidir o que se pretende fazer: **especificar o problema** a resolver.
- ▶ Em segundo lugar escolher **as estruturas de dados** que vão ser usadas.
- ▶ **Finalmente** pode começar-se a escrever o código. Em geral começa-se com um plano (tal como fazemos na composição de um texto). Os detalhes ficam para depois. Chama-se a isto **programação descendente** (*top-down programming*).
- ▶ Começa-se por escrever o programa principal, a função `main`, identificando as funções necessárias.  
Depois da função `main` estar pronta, podemos começar a escrever as restantes funções.
- ▶ O programa principal nunca deve exceder duas páginas, idealmente deveria estar numa única. A dimensão das funções não deve exceder as 3 páginas.

# Funções

Programação estruturada: *programação ascendente*

- ▶ Uma outra forma de abordar a escrita do código é a **programação ascendente** (*bottom-up programming*)

# Funções

Programação estruturada: *programação ascendente*

- ▶ Uma outra forma de abordar a escrita do código é a **programação ascendente** (*bottom-up programming*)
- ▶ Neste tipo de aproximação, escrevem-se em primeiro lugar as funções de mais baixo nível, testam-se e só depois de avança para a construção da função (ou funções) que as utilizam.

# Funções

Programação estruturada: *programação ascendente*

- ▶ Uma outra forma de abordar a escrita do código é a **programação ascendente** (*bottom-up programming*)
- ▶ Neste tipo de aproximação, escrevem-se em primeiro lugar as funções de mais baixo nível, testam-se e só depois de avança para a construção da função (ou funções) que as utilizam.

Conselhos básicos:

- ▶ Pensar antes de agir! Resista à tentação de começar logo a escrever o código! Sente-se e reflecta sobre o problema que quer resolver. Planei o código que precisará implementar. Procure a solução mais simples e flexível

# Funções

Programação estruturada: *programação ascendente*

- ▶ Uma outra forma de abordar a escrita do código é a **programação ascendente** (*bottom-up programming*)
- ▶ Neste tipo de aproximação, escrevem-se em primeiro lugar as funções de mais baixo nível, testam-se e só depois de avança para a construção da função (ou funções) que as utilizam.

Conselhos básicos:

- ▶ Pensar antes de agir! Resista à tentação de começar logo a escrever o código! Sente-se e reflecta sobre o problema que quer resolver. Planei o código que precisará implementar. Procure a solução mais simples e flexível
- ▶ Organize a informação de que irá necessitar: documentação, gráficos, diagramas, etc.

# Funções

Programação estruturada: *programação ascendente*

- ▶ Uma outra forma de abordar a escrita do código é a **programação ascendente** (*bottom-up programming*)
- ▶ Neste tipo de aproximação, escrevem-se em primeiro lugar as funções de mais baixo nível, testam-se e só depois de avança para a construção da função (ou funções) que as utilizam.

Conselhos básicos:

- ▶ Pensar antes de agir! Resista à tentação de começar logo a escrever o código! Sente-se e reflecta sobre o problema que quer resolver. Planei o código que precisará implementar. Procure a solução mais simples e flexível
- ▶ Organize a informação de que irá necessitar: documentação, gráficos, diagramas, etc.
- ▶ Teste o código à medida que o vai implementando!

# Resumo: formato das função e sua importância

## Funções: formato genérico

```
tipo nome_da_funcao(tipo1 param1, tipo2 param2,
 ..., tipon paramn)
{
 corpo da função
}
```

## Importância das funções

- ▶ Permitem reduzir a complexidade de um programa.
- ▶ Evitam a repetição de código ao longo do programa.
- ▶ Permitem uma **programação estruturada**.

Devem funcionar como uma **caixa negra** que executa a tarefa para a qual foi implementada, sem efeitos colaterais.

# Resumo: características das funções

## Características

- ▶ Cada função tem um **nome único**, que deve ser informativo.
- ▶ Uma função pode ser **invocada** a partir de outras funções.
- ▶ Uma função só é **executada** quando é **invocada**.
- ▶ Uma função **pode** receber **argumentos** que determinam o seu comportamento.
- ▶ Uma função **pode** **retornar**, para a função que a invocou, um **único** valor de saída. Esse valor deve ser do tipo da função e deve surgir após a palavra **return**.
- ▶ Assim que a instrução *return* é executada a função termina a sua execução.
- ▶ Uma função que não devolve nada é do tipo **void** e pode ser designada por *procedimento*.

# Resumo: Localização das funções

## Localização

- ▶ Uma função pode ser **definida** (colocada) em qualquer posição do ficheiro, antes ou depois do *main*.
- ▶ O **protótipo** de uma função consiste no seu cabeçalho terminado por ponto e vírgula.
- ▶ O protótipo **declara** a função: passa a ser um símbolo conhecido que pode ser usada para obter um programa que pode ser *compilado*.  
O programa *executável* correspondente (i.e. que chama essa função) requer a definição dessa função.
- ▶ Uma função só pode ser invocada se estiver **previamente** definida ou declarada.

Recomenda-se que os protótipos das funções estejam no início do programa, antes do código de qualquer função (i.e., da sua definição).

# Programação de Computadores

## Capítulo 7: Tabelas/Arrays



Os materiais desta disciplina são o resultado de um processo de criação com a contribuição de vários docentes, destacando-se a  
Professora Teresa Martinez do DEEC-UC.

(PdC)

# Plano: Tabelas

1. Tabelas Uni-dimensionais
2. Passagem de tabelas para funções
3. Constantes
4. Tabelas multi-dimensionais
5. Passagem de tabelas multi-dimensionais para funções

Bibliografia:  
Capítulo 6 de [1]

# Tabelas Uni-dimensionais

- ▶ Ao declaramos uma variável obtemos espaço para armazenar um valor do tipo dessa variável.
- ▶ Assumindo que os inteiros ocupam 32 bits (4 octetos) e que a memória é endereçada octeto a octeto:

```
int idadeA; // idade do sujeito A
idadeA = 19;
```

Resulta em (o valor escolhido para endereço é apenas exemplificativo):



# Tabelas Uni-dimensionais

Sintaxe – tabelas de capacidade fixa

- ▶ Utilizamos **tabelas** (*arrays* em língua inglesa) quando pretendermos reservar um **bloco de memória** para guardar um **conjunto de valores do mesmo tipo**.
- ▶ Desta forma, ficamos com um **conjunto de elementos consecutivos, do mesmo tipo**, que podem ser acedidos **individualmente** usando o **mesmo nome de variável**.

## Sintaxe

**tipo nome\_da\_variavel [nº de elementos];**  
em que o **nº de elementos** é uma expressão constante conhecida no momento da compilação.

# Tabelas Uni-dimensionais

Sintaxe – tabelas de capacidade fixa

- ▶ Utilizamos **tabelas** (*arrays* em língua inglesa) quando pretendermos reservar um **bloco de memória** para guardar um **conjunto de valores do mesmo tipo**.
- ▶ Desta forma, ficamos com um **conjunto de elementos consecutivos, do mesmo tipo**, que podem ser acedidos **individualmente** usando o **mesmo nome de variável**.

## Sintaxe

`tipo nome_da_variavel [nº de elementos];`

em que o **nº de elementos** é uma expressão constante conhecida no momento da compilação.

## Sintaxe com inicialização

`tipo nome_da_variavel [ ] ={valor_1, valor_2, ..., valor_n};`

em que o **nº de elementos** é dado pelo número de elementos listados (n).

# Tabelas Uni-dimensionais

Sintaxe – tabelas de capacidade variável

- Com a introdução do C99 e C11 passou a ser possível declarar *variable length array types*, ou seja tabelas cuja capacidade só é conhecida no momento da execução.

## Sintaxe

`tipo nome_da_variavel [nº de elementos];`

em que o *nº de elementos* deve ser uma *expressão* (de valor > 0) conhecida no *momento da execução*.

# Tabelas Uni-dimensionais de capacidade fixa

Ilustrando com vetor de inteiros

- ▶ Assumindo que os inteiros ocupam 32 bits (4 octetos) e que a memória é endereçada octeto a octeto:

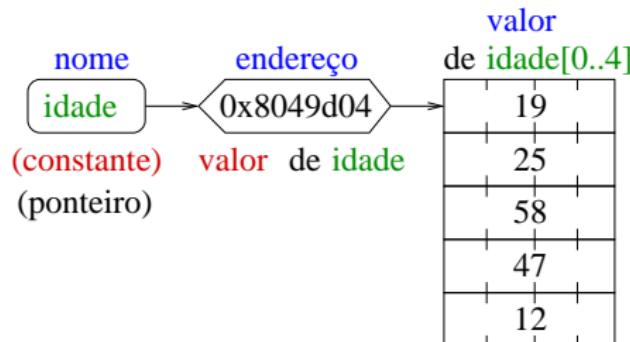
```
int idade[5];
/* Colocando valores
na tabela */
idade[0] = 19;
idade[1] = 25;
idade[2] = 58;
idade[3] = 47;
idade[4] = 12;
```

# Tabelas Uni-dimensionais de capacidade fixa

Ilustrando com vetor de inteiros

- ▶ Assumindo que os inteiros ocupam 32 bits (4 octetos) e que a memória é endereçada octeto a octeto:

```
int idade[5];
/* Colocando valores
na tabela */
idade[0] = 19;
idade[1] = 25;
idade[2] = 58;
idade[3] = 47;
idade[4] = 12;
```



- ▶ E o endereço de `idade[1]` será dado por `0x8049d08` (de acordo com o exemplo apresentado)

---

**Nota:** A palavra ponteiro ou apontador será utilizada para ser referir a símbolos cuja informação associada seja o endereço de alguma entidade.

# Tabelas Uni-dimensionais de capacidade fixa

## Exemplificando

```
1 //idades.c
2 #include <stdio.h>
3 int main()
4 {
5 int d[5]; // tabela com 5 idades
6 float idmedia; // media das idades
7 // as 5 idades
8 d[0] = 19; // Primeira posição é [0]
9 d[1] = 25;
10 d[2] = 58;
11 d[3] = 47;
12 d[4] = 12; // Quinta posição é [5-1]
13 // Conversão implícita de int para float
14 idmedia = d[0] + d[1] + d[2] + d[3] + d[4];
15 // calcula a média
16 idmedia = idmedia / 5.0 ;
17 // Apresenta valor calculado com 1 casa decimal
18 printf("\nA idade media vale: %.1f\n", idmedia);
19 return(0);
20 }
```

Produziria a saída:

```
A idade media vale: 32.2
```

# Tabelas Uni-dimensionais

Indices e número de elementos

## Acesso aos elementos de uma tabela

O índice  $k$  colocado entre [ ] à frente do nome de uma tabela, indica a distância a que esse elemento se encontra do primeiro elemento da tabela.

### Índice do primeiro elemento

O índice do **primeiro** elemento (ou seja do elemento na primeira posição) é sempre **0 (zero)**.

O primeiro elemento está à distância 0 de si próprio!

### Índice do último elemento

O índice do **último** elemento de uma tabela com  $n$  elementos é sempre  **$n - 1$** .

---

**Nota:** A Ref. [1] **não é rigorosa** na utilização da palavra **posição**, pois algumas vezes é usada como sinónimo de **índice**.

# Tabelas Uni-dimensionais de capacidade variável

Exemplo: tabelas de capacidade variável (C99 e C11)

```
1 #include <stdio.h> /*variable-length-array.c */
2 int main()
3 {
4 unsigned int n, i;
5 /* Inicializando n com valor dado pelo utilizador*/
6 printf("\nde-me inteiro > 0: ");
7 scanf("%u", &n);
8 /*-----
9 /* Cria tabela com o número de elementos dado pelo utilizador */
10 float av[n];
11 /* Preenche tabela com valores digitados pelo utilizador */
12 for(i=0; i<n; i++)
13 {
14 printf("de-me av[%u] : ", i);
15 scanf("%f", &av[i]);
16 }
17 for(i=0; i<n; i++) printf("av[%u]=%5.1f\t", i, av[i]);
18 return(0);
19 }
```

```
de-me inteiro > 0: 3
de-me av[0] : 34.5
de-me av[1] : -23.455
de-me av[2] : 12.34144124
av[0]= 34.5 av[1]=-23.5 av[2]= 12.3
```

# Inicialização de Tabelas Uni-dimensionais

- ▶ Quando uma tabela é definida ( e não é preenchida), o valor dos seus elementos é **indefinido**.
- ▶ Apenas na **inicialização**, é possível colocar numa tabela, toda a informação **de uma só vez**:
  - ▶ Listando todos os valores:

```
int primavera[4] = {3, 4, 5, 6};
```
  - ▶ Preenchendo tudo a 0 (zero):

```
int primavera[4] = {};
```
  - ▶ Listando os primeiros, os restantes ficam com zero:

```
int primavera[4] = {3, 4};
```
  - ▶ Listando todos os valores que determinam o tamanho da tabela:

```
int primavera[] = {3, 4, 5, 6};
```

# Inicialização de Tabelas Uni-dimensionais

- ▶ Quando uma tabela é definida ( e não é preenchida), o valor dos seus elementos é **indefinido**.
- ▶ Apenas na **inicialização**, é possível colocar numa tabela, toda a informação **de uma só vez**:
  - ▶ Listando todos os valores:

```
int primavera[4] = {3, 4, 5, 6};
```
  - ▶ Preenchendo tudo a 0 (zero):

```
int primavera[4] = {};
```
  - ▶ Listando os primeiros, os restantes ficam com zero:

```
int primavera[4] = {3, 4};
```
  - ▶ Listando todos os valores que determinam o tamanho da tabela:

```
int primavera[] = {3, 4, 5, 6};
```
- ▶ **Exceção:** leitura/escrita de um bloco de dados, dado o endereço de início do bloco onde se pretende ler/escrever – só será explicado no final do semestre (quando for dada a **leitura/escrita em ficheiros**).

# Inicialização de Tabelas Uni-dimensionais

- ▶ Quando uma tabela é definida ( e não é preenchida), o valor dos seus elementos é **indefinido**.
- ▶ Apenas na **inicialização**, é possível colocar numa tabela, toda a informação **de uma só vez**:
  - ▶ Listando todos os valores:

```
int primavera[4] = {3, 4, 5, 6};
```
  - ▶ Preenchendo tudo a 0 (zero):

```
int primavera[4] = {};
```
  - ▶ Listando os primeiros, os restantes ficam com zero:

```
int primavera[4] = {3, 4};
```
  - ▶ Listando todos os valores que determinam o tamanho da tabela:

```
int primavera[] = {3, 4, 5, 6};
```
- ▶ **Exceção:** leitura/escrita de um bloco de dados, dado o endereço de início do bloco onde se pretende ler/escrever – só será explicado no final do semestre (quando for dada a **leitura/escrita em ficheiros**).
- ▶ Assim (durante quase todo o semestre) **os elementos de uma tabela devem ser escritos/lidos um de cada vez** (como no exemplo dos slides 7 ou 9).

# Exemplo de inicialização de Tabelas Uni-dimensionais

```
1 #include <stdio.h> /* tabela-incipia-int.c */
2 int main()
3 {
4 int a[5] = {1,2,3,4,5}; /* inicializa tudo */
5 int b[5] = {19,25,58}; /* 3 valore e 2 zeros */
6 int c[5]; /* inicializa tudo a zero */
7 int d[5]; /* sem inicializar*/
8
9 int i;
10 for(i=0; i < 5; i++) printf("a[%d]=%d\t", i, a[i]);
11 putchar('\n');
12 for(i=0; i < 5; i++) printf("b[%d]=%d\t", i, b[i]);
13 putchar('\n');
14 for(i=0; i < 5; i++) printf("c[%d]=%d\t", i, c[i]);
15 putchar('\n');
16 for(i=0; i < 5; i++) printf("d[%d]=%d\t", i, d[i]);
17
18 return(0);
19 }
```

Uma saída (a última linha será diferente em cada execução):

```
a[0]=1 a[1]=2 a[2]=3 a[3]=4 a[4]=5
b[0]=19 b[1]=25 b[2]=58 b[3]=0 b[4]=0
c[0]=0 c[1]=0 c[2]=0 c[3]=0 c[4]=0
d[0]=4196272 d[1]=0 d[2]=4195552 d[3]=0 d[4]=-1002533456
```

## Exemplo: Cálculo do rendimento anual 1/2

```
1 #include <stdio.h> /* prog0601.c de [1] alterado */
2
3 int main()
4 {
5 float sal[12]; /* 12 meses */
6 float total;
7 int i;
8
9 for (i=0; i<12 ; i++)
10 {
11 printf("Introd. o salário do mes %d:",i+1);
12 scanf("%f",&sal[i]);
13 }
14
15 /* Mostrar os valores Mensais e calcular o total */
16 puts(" Mes Valor ");
17 for (i=0, total=0.0 ; i<12 ; i++)
18 {
19 printf(" %3d %9.2f\n",i+1,sal[i]);
20 total+=sal[i];
21 }
22
23 printf("Total Anual: %9.2f\n",total);
24 return 0;
25 }
```

## Exemplo: Cálculo do rendimento anual 2/2

```
Introd. o salário do mes 1:1231
Introd. o salário do mes 2:1232
Introd. o salário do mes 3:1313
Introd. o salário do mes 4:1313
Introd. o salário do mes 5:1222
Introd. o salário do mes 6:1232
Introd. o salário do mes 7:1231
Introd. o salário do mes 8:1331
Introd. o salário do mes 9:1231
Introd. o salário do mes 10:1231
Introd. o salário do mes 11:1231
Introd. o salário do mes 12:1321
```

| Mes | Valor   |
|-----|---------|
| 1   | 1231.00 |
| 2   | 1232.00 |
| 3   | 1313.00 |
| 4   | 1313.00 |
| 5   | 1222.00 |
| 6   | 1232.00 |
| 7   | 1231.00 |
| 8   | 1331.00 |
| 9   | 1231.00 |
| 10  | 1231.00 |
| 11  | 1231.00 |
| 12  | 1321.00 |

Total Anual: 15119.00

# Tabelas e acessos indevidos

- ▶ O compilador não verifica se o índice que estamos a utilizar está no intervalo correcto ( $0, \dots, n - 1$ , para um tabela de dimensão  $n$ ), ou seja não há qualquer erro de compilação ou aviso.
- ▶ Se durante a execução de um programa se verificar um acesso **fora da gama** dos índices de uma tabela:
  - ▶ Se for um acesso **fora da zona** de memória do nosso processo, o programa é terminado pelo sistema operativo.
  - ▶ Se for um acesso de **leitura na zona** de memória do nosso processo...
  - ▶ Se for um acesso de **escrita na zona** de memória do nosso processo...

# Passagem de tabelas para funções

- ▶ Considere-se uma função que preenche uma tabela de 10 elementos (de tipo *int*) com um valor que é também passado como argumento.
- ▶ Considere-se outra função que preenche uma tabela de 20 elementos(de tipo *int*) com um valor que é também passado como argumento.

Veja-se o exemplo que se segue no slide seguinte.

# Passagem de tabelas para funções

Exemplo: uma função por cada tabela com diferente número de elementos!

```
1 /* prog0603a.c, prog0603.c de [1] alterado */
2 #include <stdio.h>
3 /* inicializa 10 elementos de s com o valor k */
4 void inic1(int s[10], int k)
5 {
6 for(int i=0;i<10;i++) s[i]=k;
7 }
8 /* inicializa 20 elementos de s com o valor k */
9 void inic2(int s[20], int k)
10 {
11 for(int i=0;i<20;i++) s[i]=k;
12 }
13 int main()
14 {
15 int d = 5;
16 int v[10];
17 int w[20];
18
19 /* O nome da tabela tem o endereço do primeiro elemento */
20 /* Apenas o nome da tabela é passado */
21 inic1(v, d); /* inicializa v usando inic1 */
22 inic2(w, 2); /* inicializa w usando inic2 */
23
24 /* Mostra valores colocados nas tabelas */
25 printf("v = ");
26 for(int i=0; i<10; i++) printf("%d ", v[i]);
27 printf("\nw = ");
28 for(int i=0; i<20; i++) printf("%d ", w[i]);
29 return 0;
}
```

```
v = 5 5 5 5 5 5 5 5 5 5
w = 2
```

# Passagem de tabelas para funções

## Funções e capacidade de uma tabela

- ▶ Uma função quando recebe um valor (no exemplo anterior, o valor de  $d$  na linha 20) como argumento de entrada, não consegue alterar  $d$  (pois apenas recebe o seu valor).
- ▶ Uma função recebe o valor associado ao nome da tabela, este consiste no endereço do primeiro elemento da tabela, e sabendo onde está o bloco de dados da tabela a função fica com a possibilidade de o alterar!

Uma analogia pode ser o que acontece quando damos a alguém a morada com a chave da porta: essa pessoa pode modificar o conteúdo da casa.

## Funções e capacidade de uma tabela

Uma função recebe o **valor associado ao nome da tabela**, que consiste no **endereço do primeiro elemento** da tabela. Mas não recebe informação acerca do tamanho desse bloco.

# Passagem de tabelas para funções

Exemplo: função que tem como parâmetro a capacidade de tabela

- ▶ No exemplo do slide da página 16, a **capacidade das tabelas** colocadas no parâmetro *s* de *inic1* e *inic2* **são ignoradas** pelo compilador.  
Essa dimensões, se colocadas no programa, serão um "lembrete" para o programador de que em *inic1* e em *inic2* as tabelas devem ter sempre 10 e 20 elementos, respectivamente, para que as funções funcionem corretamente.

# Passagem de tabelas para funções

Exemplo: função que tem como parâmetro a capacidade de tabela

- ▶ No exemplo do slide da página 16, a **capacidade das tabelas** colocadas no parâmetro *s* de *inic1* e *inic2* **são ignoradas** pelo compilador.  
Essa dimensões, se colocadas no programa, serão um "lembrete" para o programador de que em *inic1* e em *inic2* as tabelas devem ter sempre 10 e 20 elementos, respectivamente, para que as funções funcionem corretamente.
- ▶ No exemplo que se segue, *inic1* e *inic2* foram substituídas pela função *inic*, que tem agora como **parâmetro *n***, o número de elementos a inicializar.
- ▶ É da **responsabilidade do programador** garantir, em cada chamada, que o valor passado a *n* seja o adequado a cada uma das tabelas a inicializar.

# Passagem de tabelas para funções

Exemplo: função que tem como parâmetro a capacidade da tabela

```
1 /* prog0603b.c, prog0603.c de [1] alterado */
2 #include <stdio.h>
3 /* inicializa os primeiros n elementos de s com o valor k */
4 void inic(int s[], int n, int k)
5 {
6 for(int i=0;i<n;i++) s[i]=k;
7 //Informação para DEBUG
8 printf("\n sizeof(s) = %lu", sizeof(s)); /* OU sizeof s */
9 }
10 int main()
11 {
12 int d = 5;
13 int v[10];
14 int w[20];
15
16 /* O nome da tabela tem o endereço do primeiro elemento */
17 /* Apenas o nome da tabela é passado */
18 inic(v, 10, d); /* inicializa v usando inic */
19 inic(w, 20, 2); /* inicializa w usando inic */
20
21 /* Mostra N° Elementos e valores colocados nas tabelas */
22 printf("\nelementos(v) = %lu e v = ", sizeof(v)/sizeof(int));
23 for(int i=0; i<10; i++) printf("%d ", v[i]);
24 printf("\nelementos(w) = %lu e w = ", sizeof(w)/sizeof(int));
25 for(int i=0; i<20; i++) printf("%d ", w[i]);
26 }
```

O compilador avisa que `sizeof(s)` (linha 8) não vai dar o tamanho do array `s` em bytes, mas sim do espaço (em bytes) necessário para armazenar um endereço.

A divisão por `sizeof(int)` (linhas 21 e 23) serve para obter a capacidade (em No de inteiros) das tabelas.

```
sizeof(s) = 8
sizeof(s) = 8
elementos(v) = 10 e v = 5 5 5 5 5 5 5 5 5 5
elementos(w) = 20 e w = 2
```

# Constantes

Uma constante é um valor fixo que não pode ser alterado ao longo da execução de um programa.

## Duas sintaxes para obter valores constantes

Duas formas de obter valores constantes em um programa:

- ▶ Através da palavra reservada *const*:

```
const tipo nome_da_constante = valor ;
```

- ▶ Através da diretiva de pré-processamento *# define*:

```
#define nome_da_constante valor
```

Note a ausência de ; no fim da directiva!

# Constantes

Uma constante é um valor fixo que não pode ser alterado ao longo da execução de um programa.

## Duas sintaxes para obter valores constantes

Duas formas de obter valores constantes em um programa:

- ▶ Através da palavra reservada *const*:

```
const tipo nome_da_constante = valor ;
```

- ▶ Através da diretiva de pré-processamento *#define*:

```
#define nome_da_constante valor
```

Note a ausência de ; no fim da directiva!

Diferença entre *const* e *#define*:

- ▶ A palavra *const* faz parte das palavras reservadas do C.
- ▶ A palavra *#define* faz parte das directivas do *pré-processador*.
- ▶ Uma constante definida com *const* tem um tipo e ocupa espaço de memória.
- ▶ Uma constante definida com *#define* é substituída, pelo pré-processador, em todo o programa pelo valor correspondente. O seu tipo será o que resultar dessa substituição.

# Exemplo: Constantes

## Exemplo: const versus #define

```
/* porg0605.c de [1] alterado */
#include <stdio.h>

const int num = 10;

void inic(int s[])
{
 for(int i=0;i<num;i++)
 s[i]=0;
}

int main()
{
 int v[num], i;

 inic(v);

 for(i=0;i<num;i++)
 v[i]=i;

 for(i=num-1;i>=0;i--)
 printf("%d\n",v[i]);

 return 0;
}
```

```
/* porg0606.c de [1] alterado */
#include <stdio.h>
/* Sem Ponto e Vírgula */
#define NUM 10

void inic(int s[])
{
 for(int i=0;i<NUM;i++)
 s[i]=0;
}

int main()
{
 int v[NUM], i;

 inic(v);

 for(i=0;i<NUM;i++)
 v[i]=i;

 for(i=NUM-1;i>=0;i--)
 printf("%d\n",v[i]);

 return 0;
}
```

# Tabelas multi-dimensionais

- ▶ Tabelas uni-dimensionais permitem representar vetores.
- ▶ Tabelas multi-dimensionais permitem representar matrizes.
- ▶ Não existe qualquer limite para o número de dimensões de uma tabela - tal só é limitado pela memória disponível para armazenar todos esses valores.

## Sintaxe: Tabelas multi-dimensionais

Para declarar uma tabela de dimensão  $n$ :

tipo nome\_da\_variavel [dim\_1][dim\_2][...][dim\_n]

- ▶ Uma matriz dimensional pode se utilizarada para representar o jogo do galo:

|   |   |   |
|---|---|---|
| X |   | O |
|   | X |   |
| X |   | O |

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[DIM] [DIM];
```

# Tabelas multi-dimensionais

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[DIM] [DIM];
```

- ▶ Em C uma matriz bidimensional é também um vetor de vetores:

galo[0] vale 

|   |  |  |   |
|---|--|--|---|
| X |  |  | 0 |
|---|--|--|---|

galo[1] vale 

|  |   |  |  |
|--|---|--|--|
|  | X |  |  |
|--|---|--|--|

galo[2] vale 

|  |  |   |
|--|--|---|
|  |  | O |
|--|--|---|

- ▶ Ou seja:

galo é uma tabela/matriz bi-dimensional

galo[i] é uma tabela/vetor de 3 elementos

galo[i][j] é o *char* na linha *i* coluna *j* da tabela/matriz galo

- ▶ Acerca dos valores.

galo tem o valor do endereço de galo[0] e dimensão DIM\*DIM

galo[i] tem o valor do endereço de galo[i][0] e dimensão DIM

# Inicialização de Tabelas multi-dimensionais

## Exemplificando

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[DIM][DIM] = { {'X', ' ', 'O'}, { ' ', 'X', ' '}, { ' ', ' ', 'O'} };
```

OU

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[DIM][DIM] = { 'X', ' ', 'O', ' ', 'X', ' ', ' ', ' ', 'O' };
```

OU (inicialização automática)

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[][DIM] = { {'X', ' ', 'O'}, { ' ', 'X', ' '}, { ' ', ' ', 'O'} };
```

OU (inicialização automática)

```
#define DIM 3
...
// matriz bi-dimensional de char
char galo[] [DIM] = { 'X', ' ', 'O', ' ', 'X', ' ', ' ', ' ', 'O' };
```

## Inicialização automática

Na inicialização automática só é possível omitir a primeira dimensão da matriz (o número de linhas).

# Inicialização de Tabelas multi-dimensionais

## Regras

### Inicialização automática

A passagem de tabelas multi-dimensionais (com  $n$  dimensões) para uma função é realizada indicando no cabeçalho desta, obrigatoriamente, o número de elementos de cada uma das  $n - 1$  dimensões à direita.

Apenas a primeira dimensão (a 1<sup>a</sup> à esquerda) pode ser omitida, colocando [ ] (ou um no caso do **protótipo** também pode colocar [\*])

# Inicialização de Tabelas multi-dimensionais

Exemplificando com o jogo do Galo (Ref. [1])

```
#include <stdio.h> /* prog0607.c de [1] alterado */
#define DIM 3
#define ESPACO ' '
void inic(char s[][DIM]){ /* Omitir a primeira dimensão */
 for(int i=0;i<DIM;i++)
 for(int j=0;j<DIM;j++)
 s[i][j]=ESPAÇO;
}
void mostra(char s[DIM][DIM]){ /* Ambas as Dimensões */
 for (int i=0;i<DIM;i++)
 { for (int j=0;j<DIM;j++)
 printf("%c %c",s[i][j],j==DIM-1?' ':'|');
 if (i!=DIM-1) printf("\n-----");
 putchar('\n');
 }
}
int main() {
 char Galo[DIM][DIM];
 int posx, posy;
 char ch = '0'; /* Carácter a Jogar */
 int n_jogadas = 0;
 inic(Galo);
 while (1) /* Ciclo Infinito */
 { mostra(Galo);
 printf("\nIntroduza a Posição de Jogo Linha Coluna: ");
 scanf("%d %d",&posx,&posy);
 posx--;posy--; /* Pois os indices do vector começam em 0 */
 if (Galo[posx][posy]==ESPAÇO) /* Casa Livre */
 { Galo[posx][posy] = ch = (ch == '0') ? 'X' : '0';
 n_jogadas++;
 } else printf("Posição já ocupada\nJogue Novamente!!!\n");
 if (n_jogadas==DIM*DIM) break; /* Acabar o Programa */
 }
 mostra(Galo); return(0);
}
```

# Resumindo: tabelas uni-dimensionais

## Uma tabela/vetor

Uma tabela uni-dimensional é um conjunto de elementos, do mesmo tipo, consecutivos, que podem ser acedidos através do nome da tabela e do índice de cada elemento.

# Resumindo: tabelas uni-dimensionais

## Uma tabela/vetor

Uma tabela uni-dimensional é um conjunto de elementos, do mesmo tipo, consecutivos, que podem ser acedidos através do nome da tabela e do índice de cada elemento.

## Declaração de uma tabela/vetor

- ▶ tipo nome\_da\_tabela[valor\_constante];
- ▶ tipo nome\_da\_tabela[valor\_de\_variavel\_na\_execucao];

# Resumindo: tabelas uni-dimensionais

## Uma tabela/vetor

Uma tabela uni-dimensional é um conjunto de elementos, do mesmo tipo, consecutivos, que podem ser acedidos através do nome da tabela e do índice de cada elemento.

## Declaração de uma tabela/vetor

- ▶ tipo nome\_da\_tabela[valor\_constante];
- ▶ tipo nome\_da\_tabela[valor\_de\_variavel\_na\_execucao];

## O acesso é feito elemento a elemento

- ▶ O índice do **primeiro** elemento é **ZERO** (0)
- ▶ O índice do **último** elemento de uma tabela com  $n$  elementos é  $n - 1$

O acesso ao conteúdo é conseguido usando **parêntesis retos**. Exemplo:

```
int tab[3] = {12, 45, 6}; // aqui [3] dá a dimensão
int k = tab[1]; // k <- 45 // aqui [1] permite aceder
```

# Resumindo: tabelas uni-dimensionais

## Inicialização **com todos** os elementos

- ▶ `char letras[4] = { 'a', 'b', 'c', 'd' };`
- ▶ É equivalente a:  
`char letras[] = { 'a', 'b', 'c', 'd' };`

# Resumindo: tabelas uni-dimensionais

## Inicialização **com todos** os elementos

- ▶ `char letras[4] = { 'a', 'b', 'c', 'd' };`
- ▶ É equivalente a:  
`char letras[] = { 'a', 'b', 'c', 'd' };`

## Inicialização **com sem todos** os elementos

- ▶ `int num[4] = {13,17};`
- ▶ É equivalente a:  
`int num[4] = {13,17,0,0};`

# Resumindo: tabelas e funções

## Passagem de uma tabela/vetor a uma função

- ▶ É um abuso de linguagem dizer que uma função recebe como parâmetro (de entrada e/ou saída) uma tabela.
- ▶ O que a função recebe é o **endereço do primeiro elemento<sup>4</sup>** da tabela.
- ▶ Por esse motivo a função precisa adicionalmente de ter como parâmetro o **número de elementos da tabela** (exceto se a função assumir que a tabela tem um número de elementos fixo).

---

<sup>4</sup>O endereço não precisa necessariamente de ser do primeiro elemento da tabela, como iremos ver mais à frente nesta unidade curricular.

# Resumindo: tabelas e funções

## Passagem de uma tabela/vetor a uma função

No **cabeçalho** de uma função:

- ▶ o nome da tabela uni-dimensional é seguida de um par de parêntesis retos, entre os quais pode estar a sua dimensão (a qual pode ser omitida, pois é ignorada pelo compilador) – mas os parêntesis retos são sempre necessários.
- ▶ No caso das tabelas multi-dimensionais apenas se pode omitir a primeira dimensão (a mais à esquerda).

Na **invocação** da função basta escrever o nome da tabela (que contém um endereço que é convertido em ponteiro para o tipo dos elementos que constituem a tabela).

## Definição de constantes em linguagem C: *const* e *define* (exemplos)

- ▶ `const double pi = 3.141592654 ;`
- ▶ `#define PI 3.141592654 // SEM "=" E SEM ";"`

# Programação de Computadores

## Capítulo 8: *Strings*



(PdC)

# Plano: *Strings*

1. Strings
2. Inicialização automática
3. Leitura e escrita de Strings
4. Passagem de strings para funções
5. Biblioteca de manipulação de strings

Bibliografia:  
Capítulo 7 de [1]

# *Strings*

## *Strings*

Uma *string* é uma sequência de caracteres, devidamente terminada, armazenado numa tabela. O fim da sequência de carateres é assinalado pelo carácter NUL (\0).

- ▶ Em C não é possível usar os operadores = ou + para manipular strings, mas existe uma poderosa biblioteca de funções para a manipulação de strings.
- ▶ Em C as *string* são representadas usando aspas (duplas) enquanto os caracteres são representados entre plicas.
  - ▶ Exemplos de *strings*: "Feliz Ano Novo"  
"Carta"  
"A"
  - ▶ Exemplos de caracteres: 'B'  
'+'  
'6'

'A' é o carácter A e ocupa 1 *byte*.

"A" é uma string que ocupa 2 *bytes*: 'A' seguido de '\0'.

## Como terminar uma *string*? 1/2

- ▶ Considere que foi declarada uma tabela de *char*, de capacidade 10:

```
char t[10];
```

e que desejamos armazenar nela a palavra "Dar".

|   |   |   |     |     |     |     |     |     |     |
|---|---|---|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 2 | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| D | a | r | ... | ... | ... | ... | ... | ... | ... |

- ▶ Sem um carácter de terminação uma função que recebesse a tabela  t não saberia quantas posições tinham informação útil!

## Como terminar uma *string*? 2/2

- Convencionando que o carácter \0 (não visível) é o **terminador** da *string* o problema anterior fica resolvido:

|   |   |   |    |     |     |     |     |     |     |
|---|---|---|----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 2 | 3  | 4   | 5   | 6   | 7   | 8   | 9   |
| D | a | r | \0 | ... | ... | ... | ... | ... | ... |

Todos os caracteres que estão depois do carácter \0 são geralmente designados por "lixo".

## Como terminar uma *string*? 2/2

- Convencionando que o carácter \0 (não visível) é o **terminador** da *string* o problema anterior fica resolvido:

|   |   |   |    |     |     |     |     |     |     |
|---|---|---|----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 2 | 3  | 4   | 5   | 6   | 7   | 8   | 9   |
| D | a | r | \0 | ... | ... | ... | ... | ... | ... |

Todos os caracteres que estão depois do carácter \0 são geralmente designados por "lixo".

- Uma tabela de *char* para ser uma *string* precisa de ter o carácter terminador \0 assinalando o fim da parte **útil** da tabela (aquela que tem informação **útil**).
- O '\0' (NUL) na tabela ASCII tem o valor 0 (zero) - 8 bits a 0.

As *Strings* são tabelas de *char*

As *strings* são tabelas de *char*.

Mas nem todas as tabelas de *char* são *strings*.

# As *Strings* não são um tipo básico

## As *Strings* não são um tipo básico

As *strings* não são um tipo básico. Ou seja não existe de facto o tipo *string*. Contudo, por omissão, o C assume que uma tabela (i.e., um *array*) de *char* é uma *string*, ou seja é uma tabela de *char* devidamente terminada pelo \0.

- ▶ Se desejamos usar uma tabela para armazenar uma linha de texto com um máximo de 20 caracteres úteis, qual a dimensão mínima dessa tabela?

# As *Strings* não são um tipo básico

## As *Strings* não são um tipo básico

As *strings* não são um tipo básico. Ou seja não existe de facto o tipo *string*. Contudo, por omissão, o C assume que uma tabela (i.e., um *array*) de *char* é uma *string*, ou seja é uma tabela de *char* devidamente terminada pelo \0.

- ▶ Se desejamos usar uma tabela para armazenar uma linha de texto com um máximo de 20 caracteres úteis, qual a dimensão mínima dessa tabela?

$$20+1!$$

Exemplo:

```
char s[20+1]; /* 20 caracteres úteis +1 para terminar */
```

# Inicialização automática

- A inicialização de *strings* segue a sintaxe da inicialização das tabelas (e não só):

```
char t1[20] = { 'A', 'n', 'a' } ; /* restantes posições a 0 */
char t2[20] = "Ana"; /* 'A', 'n', 'a' '\0' e resto é lixo */
char t3[] = "Ana"; /* 'A', 'n', 'a' '\0' */
char t4[3+1] = "Ana"; /* Equivalente ao anterior */
```

# Inicialização automática

- ▶ A inicialização de *strings* segue a sintaxe da inicialização das tabelas (e não só):

```
char t1[20] = { 'A', 'n', 'a' } ; /* restantes posições a 0 */
char t2[20] = "Ana"; /* 'A', 'n', 'a' '\0' e resto é lixo */
char t3[] = "Ana"; /* 'A', 'n', 'a' '\0' */
char t4[3+1] = "Ana"; /* Equivalente ao anterior */
```

- ▶ Mas

```
char t1[3] = { 'A', 'n', 'a' } ; /* Não é uma string*/
```

é uma tabela com 3 *char*, mas não é uma *string*.

- ▶ E porque não é uma *string*?
- ▶ Porque não foi colocado o carácter terminador, pelo que os 3 caracteres irão ser usados **individualmente** e não como **um todo**.

# Inicialização automática e ponteiros

Na página 233 da Ref. [1] é dito que,

```
char nome[] = "Andre"; /* Equivalente a char nome[5+1]="Andre" */
char * nome = "Andre"; /* idem */
```

Mas está **errado**: `char * nome = "Andre";`  
é diferente de `char nome[] = "Andre";`

# Inicialização automática e ponteiros

Na página 233 da Ref. [1] é dito que,

```
char nome[] = "Andre"; /* Equivalente a char nome[5+1]="Andre" */
char * nome = "Andre"; /* idem */
```

Mas está **errado**: `char * nome = "Andre";`

é diferente de `char nome[] = "Andre";`

Antes de descrever as diferenças entre a duas instruções em cima é preciso explicar o significado de `char * nome`:

## Sintaxe: ponteiro ou apontador

`tipo * p;`

- ▶ A variável *p* é capaz de armazenar um **endereço de memória** e o **valor na posição de memória** com esse endereço deve ser do **tipo que precede o asterisco**.
- ▶ O tipo **é importante** porque dado um endereço de memória, a interpretação dos bits que aí se encontram depende do tipo da informação armazenada.

# Inicialização automática com ponteiros e tabelas

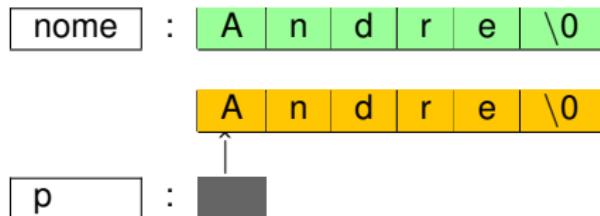
As declarações;

```
char nome[] = "Andre"; /* 'A', 'n', 'd', 'r', 'e', '\0' */
char * p = "Andre"; /* Não é idem (erro em Ref. [1]) */
```

Correspondem à criação:

- ▶ da tabela *nome*, que reserva 6 *bytes*, onde coloca "Andre"; estes *bytes* podem ser reescritos;
- ▶ do apontador *p* para *char*, que é inicializado com o endereço de memória onde foi guardada a *string constante* "Andre", a qual não pode ser alterada (mas o valor de *p* pode ser alterado!).

Graficamente:



# Inicialização automática com ponteiros e tabelas

Dadas as declarações:

```
char nome[] = "Andre"; /* 'A', 'n', 'd', 'r', 'e', '\0' */
char * p = "Andre"; /* Não é idem (erro em Ref. [1]) */
```

## Ponteiro ou apontador e tabelas

O valor de *nome* é o endereço de *nome[0]*, mas *nome* é do tipo **tabela (array)**, não é um ponteiro:

- ▶ *sizeof(nome) == 6 bytes* (a capacidade da tabela)
- ▶ *sizeof(p) == 8 bytes* o espaço necessário para armazenar um endereço de memória (uma máquina de 64 bits).

```
/* Altera a tabela: tudo em maiúsculas */
nome[1]='N'; nome[2]='D'; nome[3]='R'; nome[4] ='E';
/* NÃO posso fazer p[1]='N'; etc */
```

# Leitura e escrita de *strings*

- ▶ A escrita de strings já algo é que temos feito desde a primeira aula.

- ▶ Função *printf*:

```
printf("Bom dia!");
```

- ▶ Função *puts*:

```
puts("Bom dia!"); /* escreve e muda de linha/*
É idêntico a
printf("Bom dia!\n");
```

Estas *strings* são constantes. São armazenadas algures na memória e não podem ser modificadas.

- ▶ A leitura de strings pode ser feita usando *scanf* com o formato "%s", mas apenas permite ler 1 palavra – os *whitespace* funcionam como terminadores da leitura.
- ▶ Como o nome da tabela já é um endereço (tem como valor o endereço de memória do elemento de índice 0), não é preciso o uso de **&**.

# Exemplos de leitura

## Exemplos de leitura com *scanf*

```
1 #include <stdio.h> /*prog0701.c de [1] alterado */
2
3 int main()
4 {
5 char Nome[50],Apelido[50];
6
7 printf("Introduza o Nome: ");
8 scanf("%s",Nome); /* sem & */
9
10 printf("Introduza o Apelido: ");
11 scanf("%s",Apelido); /* sem & */
12
13 printf("Nome Completo: %s %s\n",Nome, Apelido);
14 return(0);
15 }
```

Poderia ter como resultado:

```
Introduza o Nome: Fernando
Introduza o Apelido: Pessoa
Nome Completo: Fernando Pessoa
```

# Exemplos de leitura com *scanf* – uma só palavra

O *\n* é removido do *buffer* (não é guardado na *string*) e termina *string* com o *\0*.

```
1 #include <stdio.h> /*prog0701.c de [1] alterado */
2
3 int main()
4 {
5 char Nome[50],Apelido[50];
6
7 printf("Introduza o Nome: ");
8 scanf("%s",Nome); /* sem & */
9
10 printf("Introduza o Apelido: ");
11 scanf("%s",Apelido); /* sem & */
12
13 printf("Nome Completo: %s %s\n",Nome, Apelido);
14 return(0);
15 }
```

Poderia ter como resultado:

```
Introduza o Nome: Fernando
Introduza o Apelido: Pessoa
Nome Completo: Fernando Pessoa
```

# Exemplos de leitura – mais do que uma palavra

É possível limitar o número de `char` lidos. Com o `scanf` é possível ler várias palavras usando `[^\n]`: pára no `\n` que fica no buffer.

```
1 #include <stdio.h> /* prog0702b.c de [1] alteradao */
2 int main()
3 {
4 char nome[50]="", outro[50]; /* strings vazias (ZZZ)*/
5 printf("Introduza o nome Completo: ");
6 /* Lê no máximo 49 char e acrescenta \0 */
7 scanf("%49[^\\n]",nome); /* Aceita tudo menos \\n*/
8 /* para poder ler outra string */
9 /* remove \\n do buffer ou char além do 49 */
10 while(getchar() != '\\n');
11 printf("Introduza Outro nome Completo: ");
12 /* Lê no máximo 49 char e acrescta \\0 */
13 scanf("%49[^\\n]",Outro); /* Aceita tudo menos \\n*/
14 printf("nome Completo: %s\\n",nome);
15 printf("nome Completo: %s\\n",outro);
16 return 0;
17 }
```

Poderia ter como resultado:

```
Introduza o Nome Completo: Guerra Junqueiro
Introduza Outro Nome Completo: Fernando Pessoa
Nome Completo: Guerra Junqueiro
Nome Completo: Fernando Pessoa
```

# Exemplos de leitura

As duas declarações que se seguem criam *strings* vazias:

```
char nome[100+1] = ""; /* Sring vazia com 100 caracteres úteis */
char vazio[] = "" /* String vazia inútil: Capacidade útil 0! */
```

```
1 #include <stdio.h> /* prog0704.c de [1] alterado */
2 int main() /* Lê nomes até o utilizador digitar simplesmente \n */
3 {
4 char nome[100]="Lixo"; /* para testes...*/
5 while (1) /* Ciclo Infinito */
6 {
7 puts("Nome:");
8 /* garante vazia antes de ler com scanf*/
9 nome[0]='\0';
10
11 /* Só adiciona \0 no fim, se tiver lido algo ZZZ */
12 scanf("%99[^\\n]",nome); /* aceita tudo menos \n*/
13 /* nome inalterada caso utilizador digite \n */
14
15 while(getchar() !='\n'); // remove chars extra e \n do buffer
16
17 if (nome[0]=='\0') /* Se a string estiver vazia */
18 break; /* Terminar o ciclo */
19 else printf("Nome Introduzido: %s\\n",nome);
20 }
21 return 0;
22 }
```

# Exemplos de leitura

Outro exemplo de leitura com `scanf` – mais do que uma palavra

O Programa anterior terá o comportamento que se segue, conforme o que for digitado.

Se o utilizador digitar <enter>:

```
Nome :
```

Se o utilizador digitar  
Ana Dias<enter>  
e em seguida se digitar  
Pedro Silva<enter>.

```
Nome :
Ana Dias
Nome Introduzido: Ana Dias
Nome :
Pedro Silva
Nome Introduzido: Pedro Silva
Nome :
```

# Exemplos de leitura

## Remoção de *gets*

- ▶ A função *gets* (Pag. 236 de [1]) permite ler uma string composta por várias palavras, mas **não é segura** pois lê tudo o que o utilizador digitar, sem levar em conta capacidade da *string* destino.
- ▶ Com o C11 a função *gets* foi removida da `<stdio.h>` e substituída nalguns compiladores (mas não no gcc) por outra função.

# Exemplos de leitura

## Remoção de *gets*

- ▶ A função *gets* (Pag. 236 de [1]) permite ler uma string composta por várias palavras, mas **não é segura** pois lê tudo o que o utilizador digitar, sem levar em conta capacidade da *string* destino.
- ▶ Com o C11 a função *gets* foi removida da `<stdio.h>` e substituída nalguns compiladores (mas não no gcc) por outra função.
- ▶ A leitura de strings usando a função *scanf* também **não é segura**, pois não garante que não é ultrapassada a capacidade da *string* destino, a **não ser** que seja colocado um **limite explícito** (ver exemplo anterior).

# Exemplos de leitura

## Remoção de *gets*

- ▶ A função *gets* (Pag. 236 de [1]) permite ler uma string composta por várias palavras, mas **não é segura** pois lê tudo o que o utilizador digitar, sem levar em conta capacidade da *string* destino.
- ▶ Com o C11 a função *gets* foi removida da `<stdio.h>` e substituída nalguns compiladores (mas não no gcc) por outra função.
- ▶ A leitura de strings usando a função *scanf* também **não é segura**, pois não garante que não é ultrapassada a capacidade da *string* destino, **a não ser** que seja colocado um **limite explícito** (ver exemplo anterior).
- ▶ Uma alternativa **segura** é a função *fgets*:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

# Exemplos de leitura

## A função *fgets*

A função *fgets* (na `<stdio.h>`), lê no máximo  $n - 1$  *char*. Se o `\n` for lido é armazenado na string! O `stdin` é do tipo *FILE \**

```
char *fgets(char *s, int n, FILE *stream);
```

```
1 #include <stdio.h> /*fgets.c */
2 int main()
3 {
4 char Nome[20];
5 printf("Introduza o Nome Completo: ");
6 fgets(Nome, sizeof(Nome), stdin);
7 printf("Nome Completo: \"%s\"\n", Nome);
8 return(0);
9 }
```

Nome dado tem mais de 19 *char*, logo é truncado em 19 e faltaria limpar o *buffer*, caso o programa continuasse.

```
Introduza o Nome Completo: Miguel de Sousa Tavares
Nome Completo: "Miguel de Sousa Tav"
```

Nome tem menos de 19 *char* – note que o `\n` foi armazenado na *string*!

```
Introduza o Nome Completo: Fernando Pessoa
Nome Completo: "Fernando Pessoa
"
```

# Passagem de *strings* para funções

- ▶ A passagem de *strings* para funções é igual à passagem de qualquer tabela uni-dimensional, pois uma *string* não é mais que uma tabela uni-dimensional de *char* devidamente terminada com um \0.

Seguem-se alguns exemplos:

- ▶ **Exemplo:** determinar o comprimento de uma *string*.  
Quando se fala em comprimento estamos a referir aos *char* iniciais antes de \0 (o qual não é contado).
- ▶ **Exemplo:** Remover o \n deixado numa string após leitura com *fgets*.

# Passagem de *strings* para funções

Função que determina o comprimento de uma *string*

```
1 #include <stdio.h>
2
3 int comprimento (char s[]); /* protótipo */
4
5 int main()
6 {
7 char nome[50] = ""; /* garante inicialização como string */
8 printf("Introduza o Nome Completo: ");
9 scanf("%49[^\\n]", nome); // nome sem \n
10
11 printf(" Nome Completo: \"%s\"\n", nome);
12 printf("Tem comprimento: %d\n", comprimento(nome));
13 return 0;
14 }
15
16 /* determina e retorna o comprimento da string s*/
17 int comprimento (char s[])
18 {
19 int k = 0;
20 while(s[k] != '\\0') k++; // \\0 não é contado
21 return k;
22 }
```

```
Introduza o Nome Completo: Ana Dias
Nome Completo: "Ana Dias"
Tem comprimento: 8
```

# Passagem de *strings* para funções

Função que remove \n no final de uma *string*, se este existir

```
1 #include <stdio.h>
2 /* determina e retorna o comprimento da string s*/
3 int comprimento (char s[]); // estilo tabela
4
5 /* remove fim de linha e devolve ponteiro para a string alterada*/
6 char * remove_fim_de_linha (char * s); // estilo ponteiro para
7
8 int main()
9 {
10 char nome[20]; /* Pouco espaço para encher.. */
11 printf("Introduza o Nome Completo: ");
12 /* lê no máximo sizeof(nome)-1 char e coloca \0 no fim*/
13 /* guarda \n em nome se couber*/
14 fgets (nome, sizeof(nome),stdin); //optional
15
16 printf("\n Antes Nome Completo: \"%s\"\n",nome);
17 printf("Depois Nome Completo: \"%s\"\n",remove_fim_de_linha(nome));
18 return 0;
19 }
20
21 /* remove fim de linha e devolve ponteiro para a string alterada*/
22 char * remove_fim_de_linha (char *s)
23 {
24 int k = comprimento(s); /* k é o índice de \0 */
25 if(k>0 && '\n' == s[k-1]) /* usa ordem de avaliação*/
26 s[k-1] = '\0'; /* remove \n da linha */
27
28 return (s); /* valor de s é o endereço de s[0] */
29 }
30 int comprimento (char s[]) /* corpo desta função devia estar aqui */
```

Exemplos de utilização no slide seguinte.

# Passagem de *strings* para funções

Função que remove \n no final de uma *string*, se este existir

Se o nome dado tem mais de 19 *char*, não há \n para remover:

```
Introduza o Nome Completo: Miguel Sousa Tavares
```

```
Antes Nome Completo: "Miguel Sousa Tavare"
```

```
Depois Nome Completo: "Miguel Sousa Tavare"
```

Se o nome dado tem menos de 19 *char*, já há \n para remover:

```
Introduza o Nome Completo: Ana Dias
```

```
Antes Nome Completo: "Ana Dias
```

```
"
```

```
Depois Nome Completo: "Ana Dias"
```

# Biblioteca de manipulação de strings

- O dispõe de uma biblioteca de manipulação de strings extensa.  
Consulte o sítio: <https://en.cppreference.com/w/c/string/byt>

# Biblioteca de manipulação de strings

- ▶ O dispõe de uma biblioteca de manipulação de strings extensa.  
Consulte o sítio: <https://en.cppreference.com/w/c/string/byte>
- ▶ Funções de *Classificação de caracteres*, definidas em `<ctype.h>` , tais como: `isalnum`, `isalpha`, `islower` etc;

# Biblioteca de manipulação de strings

- ▶ O dispõe de uma biblioteca de manipulação de strings extensa. Consulte o sítio: <https://en.cppreference.com/w/c/string/byte>
- ▶ Funções de *Classificação de caracteres*, definidas em `<ctype.h>` , tais como: `isalnum`, `isalpha`, `islower` etc;
- ▶ Funções de *Manipulação de caracteres*: `tolower`, `toupper`. Se a conversão não é possível, devolve o valor do argumento de entrada.

# Biblioteca de manipulação de strings

- ▶ O dispõe de uma biblioteca de manipulação de strings extensa. Consulte o sítio: <https://en.cppreference.com/w/c/string/byte>
- ▶ Funções de *Classificação de caracteres*, definidas em `<ctype.h>`, tais como: `isalnum`, `isalpha`, `islower` etc;
- ▶ Funções de *Manipulação de caracteres*: `tolower`, `toupper`. Se a conversão não é possível, devolve o valor do argumento de entrada.
- ▶ Funções de *Conversão em formatos numéricos*, definidas em `<stdlib.h>`, tais como: `atof` e `atoi`.

# Biblioteca de manipulação de strings

- ▶ O dispõe de uma biblioteca de manipulação de strings extensa. Consulte o sítio: <https://en.cppreference.com/w/c/string/byte>
- ▶ Funções de *Classificação de caracteres*, definidas em `<ctype.h>`, tais como: `isalnum`, `isalpha`, `islower` etc;
- ▶ Funções de *Manipulação de caracteres*: `tolower`, `toupper`. Se a conversão não é possível, devolve o valor do argumento de entrada.
- ▶ Funções de *Conversão em formatos numéricos*, definidas em `<stdlib.h>`, tais como: `atof` e `atoi`.
- ▶ Funções de manipulação de *strings*, definidas em `<string.h>`:
  - ▶ `char *strcpy( char *dest, const char *src );`
  - ▶ `char *strncpy( char *dest, const char *src, size_t count );`
  - ▶ `char *strcat( char *dest, const char *src );`
  - ▶ `char *strncat( char *dest, const char *src, size_t count );`

# Biblioteca de manipulação de strings

- ▶ O dispõe de uma biblioteca de manipulação de strings extensa.  
Consulte o sítio: <https://en.cppreference.com/w/c/string/byte>
- ▶ Funções de *Classificação de caracteres*, definidas em `<ctype.h>`, tais como: `isalnum`, `isalpha`, `islower` etc;
- ▶ Funções de *Manipulação de caracteres*: `tolower`, `toupper`. Se a conversão não é possível, devolve o valor do argumento de entrada.
- ▶ Funções de *Conversão em formatos numéricos*, definidas em `<stdlib.h>`, tais como: `atof` e `atoi`.
- ▶ Funções de manipulação de `strings`, definidas em `<string.h>`:
  - ▶ `char *strcpy( char *dest, const char *src );`
  - ▶ `char *strncpy( char *dest, const char *src, size_t count );`
  - ▶ `char *strcat( char *dest, const char *src );`
  - ▶ `char *strncat( char *dest, const char *src, size_t count );`
- ▶ Funções de inspecção de `strings`, definidas em `<string.h>`:
  - ▶ `size_t strlen( const char *str );` (igual a comprimento)
  - ▶ `int strcmp( const char *lhs, const char *rhs );` – devolve um valor negativo, 0 ou positivo se lexicograficamente a `lhs` é anterior, igual ou posterior a `rhs`, respetivamente;
  - ▶ entre outras...

**Problema:** Se alguma das strings não estiver terminada...

# Resumindo: strings

- ▶ As *strings* são apenas tabelas uni-dimensionais que contêm um caractere especial – o caractere '\0'.
- ▶ O caractere '\0' sinaliza o fim da zona útil dos caracteres de uma *string*.
- ▶ As *strings* são representadas entre aspas duplas e os caracateres entre pelicas: "A" e 'a' são diferentes (ver slide 3).
- ▶ As *strings* **não sendo um tipo básico** da linguagem **não podem ser manipuladas diretamente** (i.e., **não** podem ser atribuídas usando o operador = nem comparadas usando os operadores relacionais <, >, <=, >=, ==).
- ▶ Existe um conjunto *standard* de rotinas de manipulação (ver slide anterior), as quais ficam disponíveis se escrevermos:  
`#include <string.h>`

# Programação de Computadores

## Capítulo 9: Ponteiros (*Pointers*)



(PdC)

# Plano: *Ponteiros*

1. O tipo ponteiro
2. Endereço de uma variável: o operador &
3. Manipulação de ponteiros: o operador \*
4. Operadores de declaração
5. Os símbolos & e \*
6. Ponteiros e Tipos de Dados
7. Ponteiros e Tabelas
8. Aritmética de Ponteiros
9. Ponteiros e Tabelas: acesso aos elementos
10. Funções: passagem de tabelas
11. Ponteiros para Ponteiros

Bibliografia:  
Capítulo 8 de [1]

# A memória do computador

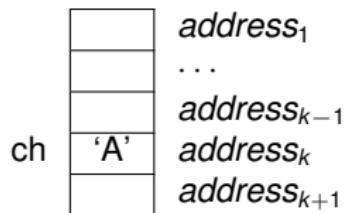
- ▶ A memória RAM (*Random Access Memory*) onde os nossos programas são carregados, pode ser vista (de forma simplificada) como um bloco contínuo de *bytes*.

# A memória do computador

- ▶ A memória RAM (*Random Access Memory*) onde os nossos programas são carregados, pode ser vista (de forma simplificada) como um bloco contínuo de *bytes*.
- ▶ Cada um desses *bytes* ocupa uma posição bem determinada nesse bloco, posição essa que é identificada por um número (o seu endereço).

# A memória do computador

- ▶ A memória RAM (*Random Access Memory*) onde os nossos programas são carregados, pode ser vista (de forma simplificada) como um bloco contínuo de *bytes*.
- ▶ Cada um desses *bytes* ocupa uma posição bem determinada nesse bloco, posição essa que é identificada por um número (o seu endereço).
- ▶ Quando fazemos: `char ch = 'A';`  
é reservado 1 *byte* (porque é tipo *char*) para armazenar 'A', e esse *byte* tem um endereço (*address* ).



# O tipo ponteiro

- ▶ A palavra *ponteiro* (ou apontador) foi introduzido no capítulo anterior.

# O tipo ponteiro

- ▶ A palavra *ponteiro* (ou apontador) foi introduzido no capítulo anterior.
- ▶ Um ponteiro com informação válida deve conter o endereço de alguma coisa: a entidade para a qual aponta.

# O tipo ponteiro

- ▶ A palavra *ponteiro* (ou apontador) foi introduzido no capítulo anterior.
- ▶ Um ponteiro com informação válida deve conter o endereço de alguma coisa: a entidade para a qual aponta.
- ▶ Até ao momento só lidámos com endereços de forma indirecta: o nome dumha tabela é sempre o endereço do seu primeiro elemento.

# O tipo ponteiro

- ▶ A palavra *ponteiro* (ou apontador) foi introduzido no capítulo anterior.
- ▶ Um ponteiro com informação válida deve conter o endereço de alguma coisa: a entidade para a qual aponta.
- ▶ Até ao momento só lidámos com endereços de forma indirecta: o nome dumha tabela é sempre o endereço do seu primeiro elemento.
- ▶ Na Ref. [1] é apresentada uma analogia com moradas e endereços.

# O tipo ponteiro

- ▶ A palavra *ponteiro* (ou apontador) foi introduzido no capítulo anterior.
- ▶ Um ponteiro com informação válida deve conter o endereço de alguma coisa: a entidade para a qual aponta.
- ▶ Até ao momento só lidámos com endereços de forma indirecta: o nome dumha tabela é sempre o endereço do seu primeiro elemento.
- ▶ Na Ref. [1] é apresentada uma analogia com moradas e endereços.
- ▶ Vamos; usar o prefixo “*p*” para as variáveis do tipo ponteiro; ou acrescentar o sufixo “*\_ptr*” às variáveis declaradas do tipo ponteiro, para as distinguir do valor para que apontam.
- ▶ O formato `%p` permite apresentar endereços que surgem em formato hexadecimal. A Ref. [1] usa `%ld` porque o formato `%p` não faz parte do K&R C.

# O tipo ponteiro

Declaração de uma variável do tipo ponteiro

- ▶ Como já foi referido, um ponteiro é declarado colocando *o operador de declaração prefixo “\*”* antes do nome da variável no momento da declaração:

```
int algo; // valor do tipo int
int *algo_ptr; // val. tipo ponteiro p/ int
```

# O tipo ponteiro

Declaração de uma variável do tipo ponteiro

- ▶ Como já foi referido, um ponteiro é declarado colocando o *operador de declaração prefixo “\*”* antes do nome da variável no momento da declaração:

```
int algo; // valor do tipo int
int *algo_ptr; // val. tipo ponteiro p/ int
```

- ▶ `algo` ainda não tem qualquer valor atribuído e `algo_ptr` ainda não aponta para lado nenhum!



Os endereços das variáveis são atribuídos no momento da execução!

# Endereço de uma variável

O operador &

- ▶ O operador & (endereço de) permite obter o endereço de uma variável.

```
1 int algo; // tipo int
2 int * algo_ptr; // tipo ponteiro para int
3 algo = 4; // algo <- 4
4 algo_ptr = & algo; //algo_ptr <- end. de algo
```

# Endereço de uma variável

O operador &

- ▶ O operador & (endereço de) permite obter o endereço de uma variável.

```
1 int algo; // tipo int
2 int * algo_ptr; // tipo ponteiro para int
3 algo = 4; // algo <- 4
4 algo_ptr = & algo; // algo_ptr <- end. de algo
```

- ▶ Inicialmente (linhas 1 e 2):



# Endereço de uma variável

O operador &

- O operador & (endereço de) permite obter o endereço de uma variável.

```
1 int algo; // tipo int
2 int * algo_ptr; // tipo ponteiro para int
3 algo = 4; // algo <- 4
4 algo_ptr = & algo; // algo_ptr <- end. de algo
```

- `algo` tem o valor 4 e `algo_ptr` aponta para `algo` (linhas 3 e 4).



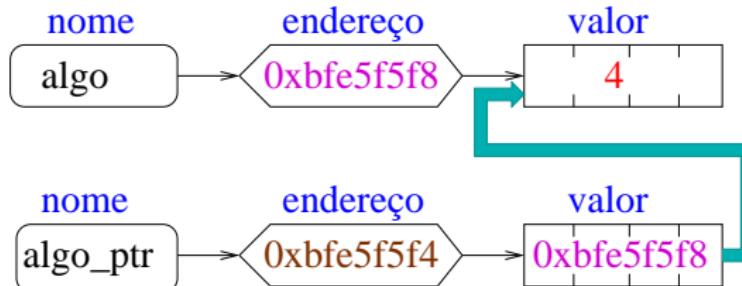
# Endereço de uma variável

O operador &

- O operador & (endereço de) permite obter o endereço de uma variável.

```
1 int algo; // tipo int
2 int * algo_ptr; // tipo ponteiro para int
3 algo = 4; // algo <- 4
4 algo_ptr = & algo; // algo_ptr <- end. de algo
```

- Ou seja, algo e \*algo\_ptr referem-se ao mesmo valor, guardado numa dada posição de memória.



# Inicialização de ponteiros

- ▶ Uma dúvida que surge algumas vezes: onde deve estar o prefixo \* na declaração de uma variável de tipo ponteiro?  
As três declarações que se seguem são sintaticamente correctas:

```
int * p1; /* Asterisco separado do tipo e da variável */
int* p2; /* Asterisco junto ao tipo
int *p3; /* Asterisco junto à variável
```

- ▶ Mas cuidado, na instrução que se segue só *p1* é do tipo *int \**:

```
// p2 e p3 sao variaveis do tipo int
int* p1, p2, p3; // só p1 é do tipo *int ie, pointer
```

# Inicialização de ponteiros

- ▶ Uma dúvida que surge algumas vezes: onde deve estar o prefixo \* na declaração de uma variável de tipo ponteiro?  
As três declarações que se seguem são sintaticamente correctas:

```
int * p1; /* Asterisco separado do tipo e da variável */
int* p2; /* Asterisco junto ao tipo */
int *p3; /* Asterisco junto à variável */
```

- ▶ Mas cuidado, na instrução que se segue só *p1* é do tipo *int \**:

```
// p2 e p3 sao variaveis do tipo int
int* p1, p2, p3; // só p1 é do tipo *int ie, pointer
```

- ▶ Inicialização automática (no momento da declaração):

```
int k = 5;
float pi = 3.14;
int * k_ptr = &k;
float * pi_ptr = π
double *p = NULL; // p <- NULL <=> p não aponta para nada
```

## Inicialização de ponteiros

É boa prática inicializar sempre os ponteiros! A atribuição da constante NULL a um ponteiro sinaliza que este ainda não aponta para nada.

# Manipulação de ponteiros

## O operador \*

O operador **\*** (desreferência) permite aceder ao (valor do) objecto apontado por um ponteiro. Note o formato **%p** para imprimir ponteiros.

```
1 #include <stdio.h> /* desrefencia.c */
2 int main()
3 {
4 int algo;
5 int * algo_ptr = NULL; /* Não aponta para nada */
6 printf("\nOs enderecos sao definidos no momento da execucao:");
7 printf("\n&algo = %p", &algo);
8 /* Atribui valores ----- */
9 algo = 4;
10 algo_ptr = & algo; /* algo_ptr aponta para algo */
11 printf("\n\nApos a atribuicao, os valores ficam definidos:");
12 printf("\nalgo_ptr = %p", algo_ptr);
13 printf("\n*algo_ptr == algo <=> %d == %d\n",
14 *algo_ptr, algo);
15 return 0;
16 }
```

# Manipulação de ponteiros

## O operador \*

O operador **\*** (desreferência) permite aceder ao (valor do) objecto apontado por um ponteiro. Note o formato **%p** para imprimir ponteiros.

```
1 #include <stdio.h> /* desrefencia.c */
2 int main()
3 {
4 int algo;
5 int * algo_ptr = NULL; /* Não aponta para nada */
6 printf("\nOs enderecos sao definidos no momento da execucao:");
7 printf("\n&algo = %p", &algo);
8 /* Atribui valores ----- */
9 algo = 4;
10 algo_ptr = & algo; /* algo_ptr aponta para algo */
11 printf("\n\nApos a atribuicao, os valores ficam definidos:");
12 printf("\nalgo_ptr = %p", algo_ptr);
13 printf("\n*algo_ptr == algo <=> %d == %d\n",
14 *algo_ptr, algo);
15 return 0;
16 }
```

Os enderecos sao definidos no momento da execucao:  
&algo = 0x7ffd2e7d6efc

Apos a atribuicao, os valores ficam definidos:  
algo\_ptr = 0x7ffd2e7d6efc  
\*algo\_ptr == algo <=> 4 == 4

# Operadores de declaração

## O operador \*

Outros operadores de declaração:

- ▶ \* (ponteiro) – prefixo.
- ▶ \*const (ponteiro constante) – prefixo (não vamos utilizar)
- ▶ [] (tabela) – sufixo
- ▶ () (função) – sufixo

É possível declarar vários nomes numa mesma declaração, mas os operadores de declaração só se aplicam a nomes individuais, e não a nomes que se sigam na mesma declaração:

```
int* i, j; // int *i; int j;
int u, *v; // int u; int *v;
int d[10], *pv; // int d[10]; int *pv;
```

Este tipo de definições tornam os programas pouco legíveis e devem ser evitados!

# Os símbolos & e \*

- ▶ Como podemos ver o símbolo \* pode ser usado em 3 situações diferentes:
  - ▶ Operador binário em expressões aritméticas, representando o produto:  
 $3 * 4$  ou  $3 * i$
  - ▶ Na declaração do tipo ponteiro:  
 $tipo * var\_ptr; // ponteiro$
  - ▶ Operador unário de desreferenciação:  
 $* var\_ptr$

# Os símbolos & e \*

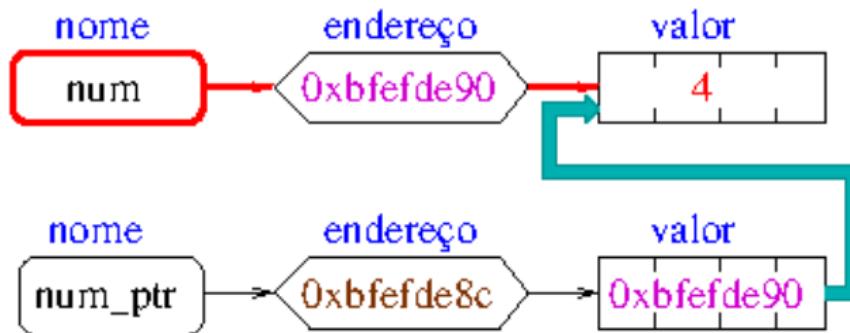
- ▶ Como podemos ver o símbolo \* pode ser usado em 3 situações diferentes:
  - ▶ Operador binário em expressões aritméticas, representando o produto:  
 $3 * 4$  ou  $3 * i$
  - ▶ Na declaração do tipo ponteiro:  
 $tipo * var\_ptr; //$  ponteiro
  - ▶ Operador unário de desreferenciação:  
 $* var\_ptr$
- ▶ E o símbolo & pode ser usado em 3 situações diferentes:
  - ▶ && – operador binário, E lógico (palavra a palavra)
  - ▶ & – operador binário, E lógico bit a bit (a introduzir)
  - ▶ Operador unário, endereço de uma variável, &nomeDeVar, assumindo que previamente:  
 $tipo nomeDeVar ; //$  declaração de variável

# Os símbolos & e \*

Exemplificando

Numa máquina de 32 bits com endereços de 32 bits:

```
int num; // espaço p/ num
int * num_ptr = & num; // espaço p/ ponteiro
num = 4; // num <- 4
```

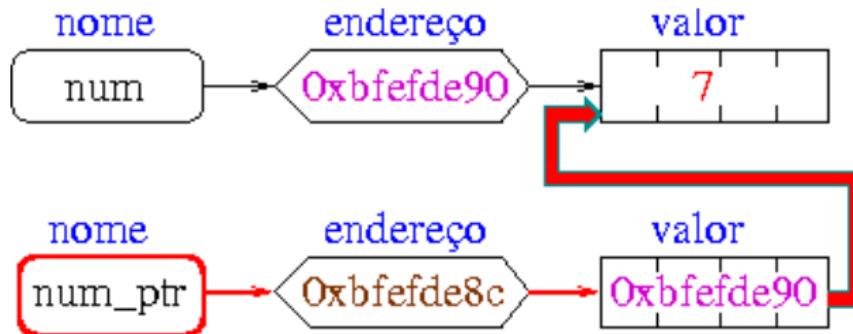


# Os símbolos & e \*

## Atribuição por desreferenciação

Numa máquina de 32 bits com endereços de 32 bits:

```
int num; // inteiro
int * num_ptr = & num; // ponteiro
* num_ptr = 7; // num <- 7
```



# Ponteiros e Tipos de Dados

- ▶ Já foi referido no capítulo anterior que os **ponteiros têm tipo**.
- ▶ No capítulo anterior foi referido que o tipo é necessário à interpretação da informação (sequência de bits) da zona de memória referenciada.
- ▶ Diferentes tipos de dados ocupam um número diferente de *bytes*, logo o tipo também é importante para, dado um endereço, saber quantos *bytes* têm de ser lidos (interpretados).

# Ponteiros e Tipos de Dados

- ▶ Já foi referido no capítulo anterior que os **ponteiros têm tipo**.
- ▶ No capítulo anterior foi referido que o tipo é necessário à interpretação da informação (sequência de bits) da zona de memória referenciada.
- ▶ Diferentes tipos de dados ocupam um número diferente de *bytes*, logo o tipo também é importante para, dado um endereço, saber quantos *bytes* têm de ser lidos (interpretados).

## Menor endereço

O endereço de memória de uma variável é sempre o do menor dos endereços que esta ocupa em memória.

## Conjunto apontado

Na declaração `tipo_apontado * ptr;`  
`ptr` endereça o número de *bytes* dado por `sizeof(tipo_apontado)` (onde `tipo_apontado` é um qualquer tipo válido).

# Ponteiros e Tipos de Dados

## Exemplo

```
#include <stdio.h> /* ponteiros-e-size-of.c */
int main() {
 /* variáveis e ponteiros para as mesmas */
 char letra = 'J'; char * letra_ptr = & letra;
 int i = 125; int * i_ptr = & i;
 double e = 2.71828; double * e_ptr = &e;
 /* Recordando o numero de bytes ocupados das variaveis */
 printf("sizeof(char) = %lu\t", sizeof(letra));
 printf(" sizeof(int) = %lu\t", sizeof(i));
 printf(" sizeof(double) = %lu\n", sizeof(e));

 /* Mostra valores atuais */
 printf("\nMostra valores Através das variáveis\n");
 printf("\tletra = '%c'\t i = %d\t e = %.2f\n", letra, i, e);
 printf("Mostra valores Através dos ponteiros\n");
 printf("\t*letra_ptr = '%c'\t *i_ptr = %d\t *e_ptr = %.2f\n",
 *letra_ptr, *i_ptr, *e_ptr);
 return(0);
}
```

```
sizeof(char) = 1 sizeof(int) = 4 sizeof(double) = 8

Mostra valores Através das variáveis
letra = 'J' i = 125 e = 2.72
Mostra valores Através dos ponteiros
*letra_ptr = 'J' *i_ptr = 125 *e_ptr = 2.72
```

# Ponteiros e Tabelas

Diferença entre ponteiros e tabelas

Seja `T tab [numero_de_elementos_de_tipo_ T];`  
e `T` um tipo válido em C (`char, int, float, double`, etc).

## Valor do nome de uma tabela

O valor do `nome de uma tabela` (no exemplo `tab`) corresponde ao `endereço do seu 1º elemento`, i.e. de `&tab[0]`. O nome de uma tabela é um valor `constante`, que indica endereço o primeiro `byte` do bloco de memória reservado para os elementos da tabela.

# Ponteiros e Tabelas

Diferença entre ponteiros e tabelas

Seja `T tab [numero_de_elementos_de_tipo_ T];`  
e `T` um tipo válido em C (`char, int, float, double`, etc).

## Valor do nome de uma tabela

O valor do *nome de uma tabela* (no exemplo `tab`) corresponde ao *endereço do seu 1º elemento*, i.e. de `&tab[0]`. O nome de uma tabela é *um valor constante*, que indica endereço o primeiro *byte* do bloco de memória reservado para os elementos da tabela.

## Tipo do nome de uma tabela

`tab` é do tipo *tabela-de-T* (*array of T*). O compilador, a partir do nome da tabela (no exemplo `tab`) sabe quantos *bytes* foram reservados para essa tabela.

A conversão de *tabela-de-T* para `* T` é possível, mas com perda de informação: a partir do ponteiro já não é possível saber o tamanho do bloco reservado para a tabela.

# Ponteiros e Tabelas

## Exemplo 1: Diferença entre ponteiros e tabelas 1/2

```
#include <stdio.h> /* tabelas-e-ponteiros.c */

int main()
{
 int a[] = {5,6,7,8}; // espaço designado por a, para bloco com 4 int
 int * p = &a[0]; // espaço designado por p, para endereço de int
/*
+---+---+---+---+
a: | 5 | 6 | 7 | 8 |
+---+---+---+---+
^
|
+-|-+
p: | * |
+---+
*/
 printf("\nTrês expressões com o valor do endereço\ndo 1º elemento do bloco");
 printf(" de 4 int\nMas que são de tipo diferente:\n");
 printf("\n &a = %p\t tipo *(int[])", &a);
 printf("\n a = %p\t tipo int[]", a);
 printf("\n&a[0] = %p\t tipo *int\n", &a[0]);

 printf("\n\n0 valor de p (duas vezes)");
 printf("\n p = %p\t tipo * int", p);
 printf("\n&p[0] = %p\t tipo * int\n", &p[0]);
 return (0);
}
```

# Ponteiros e Tabelas

## Exemplo 1: Diferença entre ponteiros e tabelas 2/2

```
a[] = {5,6,7,8}; // espaço designado por a, para bloco com 4 int
* p = &a[0]; // espaço designado por p, para endereço de int
```

Três expressões com o valor do endereço  
do 1º elemento do bloco de 4 int  
Mas que são de tipo diferente:

```
&a = 0x7fff9e3c03e0 tipo *(int[])
a = 0x7fff9e3c03e0 tipo int[]
&a[0] = 0x7fff9e3c03e0 tipo *int
```

O valor de p (duas vezes)  
p = 0x7fff9e3c03e0 tipo \* int  
&p[0] = 0x7fff9e3c03e0t tipo \* int

# Ponteiros e Tabelas

## Exemplo 2: Ponteiros e tabelas

```
#include <stdio.h> /* tabelas-e-ponteiros.c */
int main()
{
 int v[] = {5,6,7,8}; /* a do tipo Tabela-de-int, com 4 int */
 int * p = v; /* p aponta para v[0]; conversão implícita
 de Tabela-de-int para int*/
 +---+---+---+---+
 v: | 5 | 6 | 7 | 8 |
 +---+---+---+---+
 ^
 |
 +-|-
 p: | * |
 +---+ */

 printf("v[0] == %d == %d\n", v[0], *p);
/*-----*/ */
 p = & v[2] ; /* p passa a apontar para v[2] */
/*-----*/
 +---+---+---+---+
 v: | 5 | 6 | 7 | 8 |
 +---+---+---+---+
 ^
 |
 +-|-
 p: | * |
 +---+ */
 printf("v[2] == %d == %d\n", v[2], *p);
 return (0);
}
```

```
v[0] == 5 == 5
v[2] == 7 == 7
```

# Aritmética de Ponteiros

- ▶ Um ponteiro pode ser incrementado como qualquer variável.
- ▶ Mas adicionar 1 a um ponteiro não significa que este passe a apontar para o *byte* seguinte, mas sim para o elemento seguindo do tipo para o qual aponta.

## Incremento de um ponteiro

Um apontador para o tipo T avança sempre `sizeof(T) bytes`, por cada unidade de incremento.

## Decremento de um ponteiro

Um apontador para o tipo T recua sempre `sizeof(T) bytes`, por cada unidade de decremento.

# Aritmética de Ponteiros

Exemplo: incremento de um ponteiro

```
#include <stdio.h> /* prog0801.c de [1] alterado */
int main()
{
 int x=5 ;
 int *px = &x;
 double y=5.1;
 double *py = &y;

 printf("%d %p\n",x,px);
 printf("%d %p\n",x+1, (px+1));

 printf("%f %p\n",y, py);
 printf("%f %p\n",y+1, (py+1));
 return (0);
}
```

Resultará em:

5 0x7ffd6ed388c**c**

6 0x7ffd6ed388d**0**

5.1 0x7ffd6ed388d**0**

6.1 0x7ffd6ed388d**8**

# Aritmética de Ponteiros

Exemplo: mostra string invertida usando ponteiros

```
#include <stdio.h> /* prog0802a.c adaptado de prog0802.c [1] */
#include <string.h>
int main() {
 char s[100] = ""; /* inicialmente está vazia */
 char *ptr = s; /* ptr aponta para s[0] */

 printf("Introduza uma String: ");
 scanf("%99[^\\n]", s);

 if (*ptr == '\\0') /* Testa String Vazia usando ptr*/
 return 1; /* return 1 */

 puts(ptr); /* Imprimir a string Normalmente usando ptr*/

 /* Imprimir a string ao contrário usando ptr*/
 ptr = s + strlen(s);
 ptr--; // para não imprimir o \0

 while (ptr >= s) /* Enquanto ptr for >= que &s[0] */
 putchar(*ptr--);
 return 0;
}
```

```
Introduza uma String: O crime do padre Amaro
O crime do padre Amaro
oramA erdap od emirc O
```

# Aritmética de Ponteiros

Exemplo: Diferença de ponteiros (para elementos do mesmo tipo)

```
#include <stdio.h> /* diferenca de-ponteiros.c */

int main()
{
 int v[] = {5,6,7,8}; /* a do tipo Tabela-de-int, com 4 int */

 printf("&v[3] = %ld\n", (long) &v[3]);
 printf("&v[0] = %ld\n", (long) &v[0]);
 printf("\nDiferença de Ponteiros (3 inteiros):");
 printf("\n&v[3] - &v[0] = %ld\n", &v[3] - &v[0]);
 printf("\nDiferença Numérica:");
 printf("\n(long)&v[3] - (long)&v[0] = %ld\n",
 (long) &v[3] - (long) &v[0]);
 return (0);
}
```

```
&v[3] = 140737224748156
&v[0] = 140737224748144
```

```
Diferença de Ponteiros (3 inteiros):
&v[3] - &v[0] = 3
```

```
Diferença Numérica:
(long)&v[3] - (long)&v[0] = 12
```

# Aritmética de Ponteiros

Exemplo: Comparação de Ponteiros - inversão de uma string

```
#include <stdio.h> /* inverte-string.c */
#include <string.h>
int main() {
 char s[100] = ""; /* inicialmente está vazia */
 printf("Introduza uma String: "); scanf("%99[^\\n]", s);

 if (s[0] == '\\0') /* Testa String Vazia */
 return 1; /* return 1 */

 /* inverter a string usando dois ponteiros*/
 char * pi = &s[0];
 char * pf = &s[strlen(s)-1]; /* strlen(s) >= */

 char aux;
 while (pi < pf) /* Enquanto pi < pf troca */
 { aux = *pi; /* troca as letras */
 *pi = *pf;
 *pf = aux;
 pf--; pi++; /* desloca ponteiros */
 }
 puts(s); /* Mostra string invertida */
 return 0;
}
```

Introduza uma String: Luis Damas  
samaD siuL

# Ponteiros e Tabelas: acesso aos elementos

## Acesso aos elementos

Se  $v$  for uma tabela, ou um apontador para o primeiro elemento de uma tabela então o elemento de índice  $n$  pode ser obtido de  $v[n]$  ou  $*(v + n)$ .

Considere-se o troço de programa:

```
char s[] = "aeiou";
char *ptr = s; /* ptr aponta para a[0] - cast implicito */
```

|                   |                                                                                                                                 |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------|
| $s[2]$            | Carácter existente no índice 2                                                                                                  |
| $*(\text{ptr}+2)$ | $\text{ptr}+2$ aponta para o carácter no índice 2 (a letra i); usando o $*$ obtém-se o valor nesse endereço.                    |
| $(\text{s}+2)$    | Como o valor de $s$ é igual a $\&s[0]$ , isto equivale ao caso anterior                                                         |
| $\text{ptr}[2]$   | Equivale a $*(\text{ptr}+2)$ . O acesso a elementos através de parêntesis retos pode ser também ser feito através de ponteiros. |

# Funções: passagem de tabelas

Já sabemos que:

- ▶ Ao declarar uma tabela, é reservado um bloco contínuo de memória, e o nome da tabela tem o endereço do 1º elemento da tabela.
- ▶ Ao invocar uma função, que tem como parâmetro uma tabela, a função não recebe toda a tabela, mas apenas o endereço do primeiro elemento da tabela.
- ▶ Esse endereço tem de ser do tipo correcto.

# Funções: passagem de tabelas

Já sabemos que:

- ▶ Ao declarar uma tabela, é reservado um bloco contínuo de memória, e o nome da tabela tem o endereço do 1º elemento da tabela.
- ▶ Ao invocar uma função, que tem como parâmetro uma tabela, a função não recebe toda a tabela, mas apenas o endereço do primeiro elemento da tabela.
- ▶ Esse endereço tem de ser do tipo correcto.
- ▶ Assim, a função que determina o comprimento de uma cadeia de caracteres, tem o protótipo:

```
size_t strlen(const char *str); /* ponteiro */
em vez de
```

```
size_t strlen(const char str[]); /* tabela */
```

para nos recordar que o que função recebe é um ponteiro e não toda a tabela (*const* para que a função não possa alterar a *string*).

# Funções: passagem de tabelas

Função que determina o comprimento de uma *string* usando ponteiros

```
1 #include <stdio.h> /*comprimento-ptr.c */
2
3 int comprimento (char *s); /* protótipo */
4
5 int main()
6 {
7 char nome[50] = ""; /* garante inicialização como string */
8 printf("Introduza o Nome Completo: ");
9 scanf("%49[^\\n]", nome); //nome sem \n
10
11 printf(" Nome Completo: \"%s\"\n", nome);
12 printf("Tem comprimento: %d\n", comprimento(nome));
13 return 0;
14 }
15
16 /* implementação com ponteiros */
17 /* determina e retorna o comprimento da string s*/
18 int comprimento (char *s)
19 {
20 char *salva = s; /* salva endereço do 1º char */
21 while(*s != '\\0') s++; /* \\0 não é contado */
22 return ((int)(s-salva)); /*número de elementos */
23 }
```

# Ponteiros para Ponteiros

- ▶ Uma variável do tipo ponteiro tem associado um espaço de memória, pelo que é possível obter o correspondente endereço através do operador &.

## Sintaxe: Ponteiro para Ponteiro

T \* \* pptr;

- ▶ `**pptr` é do tipo `T`.
- ▶ `*pptr` é do tipo `T *` do tipo ponteiro para valor do tipo `T`.
- ▶ `pptr` é do tipo `T **` ou seja do tipo ponteiro para ponteiro de valor do tipo `T`.

O número de \* que podem preceder a declaração de um ponteiro não tem qualquer limitação!

# Ponteiros para Ponteiros

Exemplo: 3 printf's com o mesmo resultado

```
1 #include <stdio.h> /*prog0805.c de [1] alterado */
2
3 int main()
4 {
5 int k = 5;
6 int * ptr_k; /* Apontador para k */
7 int ** ptr_ptr_k; /* Apontador para o Apontador de k */
8
9 /* Inicialização dos apontadores */
10
11 ptr_k = &k;
12 ptr_ptr_k = &ptr_k;
13
14 printf("k = %d : &k = %p\n",k, &k);
15 printf("k = %d : &k = %p\n",*ptr_k, ptr_k);
16 printf("k = %d : &k = %p\n",**ptr_ptr_k, *ptr_ptr_k);
17
18 return 0;
19 }
```

```
k = 5 : &k = 0x7ffd1e335804
k = 5 : &k = 0x7ffd1e335804
k = 5 : &k = 0x7ffd1e335804
```

# Resumindo: ponteiros e endereços

- ▶ Um **apontador ou ponteiro** é uma variável que contém o endereço de outra variável<sup>5</sup>

---

<sup>5</sup>Em Estruturas de Dados e Algoritmos utilizarão funções que permitem obter blocos de memória e um ponteiro para o seu início.

# Resumindo: ponteiros e endereços

- ▶ Um **apontador ou ponteiro** é uma variável que contém o endereço de outra variável<sup>5</sup>
- ▶ A **declaração de um ponteiro** é feita usando o tipo da variável para o qual este deve apontar, seguido de um asterisco (*tipo \* ptr*)

---

<sup>5</sup>Em Estruturas de Dados e Algoritmos utilizarão funções que permitem obter blocos de memória e um ponteiro para o seu início.

# Resumindo: ponteiros e endereços

- ▶ Um **apontador ou ponteiro** é uma variável que contém o endereço de outra variável<sup>5</sup>
- ▶ A **declaração de um ponteiro** é feita usando o tipo da variável para o qual este deve apontar, seguido de um asterisco (*tipo \* ptr*)
- ▶ O **endereço de uma variável** obtém-se usando o operador **&** (endereço de)

---

<sup>5</sup>Em Estruturas de Dados e Algoritmos utilizarão funções que permitem obter blocos de memória e um ponteiro para o seu início.

# Resumindo: ponteiros e endereços

- ▶ Um **apontador ou ponteiro** é uma variável que contém o endereço de outra variável<sup>5</sup>
- ▶ A **declaração de um ponteiro** é feita usando o tipo da variável para o qual este deve apontar, seguido de um asterisco (*tipo \* ptr*)
- ▶ O **endereço de uma variável** obtém-se usando o operador **&** (endereço de)
- ▶ Se uma variável apontador **ptr** contiver um endereço de outra variável, o valor dessa variável pode obter-se usando o operador **\*** (**desreferência**), i.e., escrevendo **\*ptr**

---

<sup>5</sup>Em Estruturas de Dados e Algoritmos utilizarão funções que permitem obter blocos de memória e um ponteiro para o seu início.

# Resumindo: ponteiros e endereços

- ▶ Um **apontador ou ponteiro** é uma variável que contém o endereço de outra variável<sup>5</sup>
- ▶ A **declaração de um ponteiro** é feita usando o tipo da variável para o qual este deve apontar, seguido de um asterisco (*tipo \* ptr*)
- ▶ O **endereço de uma variável** obtém-se usando o operador **&** (endereço de)
- ▶ Se uma variável apontador **ptr** contiver um endereço de outra variável, o valor dessa variável pode obter-se usando o operador **\*** (**desreferência**), i.e., escrevendo **\*ptr**
- ▶ Os apontadores devem ser inicializados com o valor **NULL**, que indica que este não aponta para nenhum endereço

---

<sup>5</sup>Em Estruturas de Dados e Algoritmos utilizarão funções que permitem obter blocos de memória e um ponteiro para o seu início.

# Resumindo: ponteiros, tabelas e funções

- ▶ Os apontadores possuem uma **aritmética** própria (incremento, decremento, adição, subtração, comparação)

# Resumindo: ponteiros, tabelas e funções

- ▶ Os apontadores possuem uma **aritmética** própria (incremento, decremento, adição, subtração, comparação)
- ▶ O **nome de uma tabela** tem o valor do **endereço do primeiro** elemento da tabela

## Resumindo: ponteiros, tabelas e funções

- ▶ Os apontadores possuem uma **aritmética** própria (incremento, decremento, adição, subtração, comparação)
- ▶ O **nome de uma tabela** tem o valor do **endereço do primeiro elemento** da tabela
- ▶ Uma **função** que tem como argumento uma tabela, recebe **apenas o endereço do primeiro elemento da tabela** e não a totalidade da tabela

# Resumindo: ponteiros, tabelas e funções

- ▶ Os apontadores possuem uma **aritmética** própria (incremento, decremento, adição, subtração, comparação)
- ▶ O **nome de uma tabela** tem o valor do **endereço do primeiro elemento** da tabela
- ▶ Uma **função** que tem como argumento uma tabela, recebe **apenas o endereço do primeiro elemento da tabela** e não a totalidade da tabela
- ▶ A chamada de uma função que tem como argumento uma tabela, resulta sempre numa **conversão implícita de tipos**

## Funções, tabelas e aritmética de ponteiros

- ▶ O operador **[n]** pode ser usado a seguir a nome de uma variável do tipo apontador.
- ▶ Devido à aritmética de ponteiros tal permite obter o **n-ésimo** elemento, do tipo apontado, após o valor apontado pelo apontador.
- ▶ Desta forma, uma função que recebe apenas o endereço do primeiro elemento de uma tabela pode aceder aos elementos desta!

# Ponteiros e saídas para o ecrã

- ▶ Já vimos que é possível mostrar o valor de um ponteiro:

```
1 int num = 4; // inteiro
2 int * num_ptr = & num; // ponteiro para inteiro
3 cout << "\nnum_ptr = " << num_ptr << endl;
```

surgindo algo do tipo:

num\_ptr = 0xbfbfab98

em que este valor seria o endereço de memória onde estaria armazenado o valor de num (4 no exemplo).

# Ponteiros e saídas para o ecrã

## Ponteiros para caracteres

- O C++ trata de forma diferente (na saída) as **ponteiros-para-char** dos restantes ponteiros.

```
char linha[] = "Bom dia!"; // string C
char * linha0_ptr = &linha[0]; // ponteiro para char
std::cout << " linha: " << linha << endl;
std::cout << "linha0_ptr: " << linha0_ptr << endl;
```

Produziria a saída:

```
 linha: Bom dia!
linha0_ptr: Bom dia!
```

Isto acontece porque um ponteiro-para-char é por omissão assumido como sendo uma string no estilo C.

# Ponteiros e saídas para o ecrã

## Ponteiros para caracteres

- ▶ O C++ trata de forma diferente (na saída) as **ponteiros-para-char** dos restantes ponteiros.

```
char linha[] = "Bom dia!"; // string C
char * linha0_ptr = &linha[0]; // ponteiro para char
std::cout << " linha: " << linha << endl;
std::cout << "linha0_ptr: " << linha0_ptr << endl;
```

Produziria a saída:

```
 linha: Bom dia!
linha0_ptr: Bom dia!
```

Isto acontece porque um ponteiro-para-char é por omissão assumido como sendo uma string no estilo C.

- ▶ No caso de ponteiros para char o que é apresentado é a string e não o valor do ponteiro!  
Ou seja espera-se que nesse endereço (no local apontado) esteja uma sequência de char devidamente terminada.

# Ponteiros e Tabelas

## Diferença entre ponteiros e tabelas

- ▶ Notou que no exemplo anterior se escreveu `&p[0]` para obter o endereço de `p[0]` (ou seja de `*p`).

Dado um ponteiro para um bloco, podemos utilizar os `[]` para aceder aos elementos nesse bloco, tal como fazemos numa tabela.

---

<sup>6</sup> Originalmente, na linguagem C, um `lvalue` é algo que pode estar no lado esquerdo de uma atribuição (*left value*). Em C++ refere-se mais a uma zona de armazenamento definida, ou seja uma zona na qual é possível fazer um armazenamento através do seu endereço (*locator value*).

# Ponteiros e Tabelas

## Diferença entre ponteiros e tabelas

- ▶ Notou que no exemplo anterior se escreveu `&p[0]` para obter o endereço de `p[0]` (ou seja de `*p`).  
Dado um ponteiro para um bloco, podemos utilizar os `[]` para aceder aos elementos nesse bloco, tal como fazemos numa tabela.
- ▶ Um lvalue<sup>6</sup> O tipo `tabela-de-T` que surja numa expressão, transforma-se (com 3 excepções) num ponteiro para o seu primeiro elemento; o tipo resultante é `ponteiro-para-T`.
- ▶ Uma dessas 3 excepções é quando a tabela é o operando da função `sizeof`.

---

<sup>6</sup> Originalmente, na linguagem C, um lvalue é algo que pode estar no lado esquerdo de uma atribuição (*left value*). Em C++ refere-se mais a uma zona de armazenamento definida, ou seja uma zona na qual é possível fazer um armazenamento através do seu endereço (*locator value*).

# Ponteiros e Tabelas

Parâmetro formais de funções: ponteiros e tabelas

- ▶ Assim, o parâmetro formal de uma função que representa uma tabela, `p` no exemplo:

```
// mostra e muda de linha
void mostra(const int p[], int dim){
 for(int i = 0; i < dim; i++) cout << p[i] << ' ';
 cout << endl;
}
```

É sempre transformado em ponteiro para (neste caso `int *`) no momento da chamada.

- ▶ Por esse mesmo motivo, a função `sizeof`, se chamada dentro da função anterior (`sizeof(p)`) devolveria o tamanho do ponteiro, o parâmetro que a função efectivamente recebeu (e não o tamanho da tabela para a qual o ponteiro aponta).

# Ponteiros e Tabelas

Parâmetro formais de funções: ponteiros e tabelas

Assim podemos alternar entre tabelas e ponteiros nos parâmetros formais de uma função:

```
// Mostra uma tabela p (dim = nº de elementos)
void mostra(const int p[], int dim);

// Calcula o produto interno de p e q (dim = nº de elementos)
int prod_interno(int p[], int *q, int dim);
```

OU

```
// Mostra uma tabela p (dim = nº de elementos)
void mostra(const int* p, int dim);

// Calcula o produto interno de p e q (dim = nº de elementos)
int prod_interno(int *p, int *q, int dim);
```

OU misturando estilos (mau estilo!):

```
// Calcula o produto interno de p e q (dim = nº de elementos)
int prod_interno(int *p, int q[], int dim);
```

# Ponteiros e Tabelas

## Parâmetro formais de funções: ponteiros e tabelas

```
#include <iostream>
using namespace std;

int prod_interno(int *p, int * q, int dim){// Produto interno
 int soma = 0;
 for(int i = 0; i< dim; i++) soma += p[i]*q[i];
 return soma;
}
//void mostra(const int *p, int dim){ // mostra array e muda de linha
//OU
void mostra(const int p[], int dim){ // mostra array e muda de linha
 for(int i = 0; i< dim; i++) cout << p[i] << ' ';
 cout << endl;
 cout << "DEBUG: mostra(int[],int) - Tamanho de p: " << sizeof(p) << "\n\n";
}
int main(){
 int a[] = {5,6,7,8,9}; // espaço designado por a para bloco com 5 int

 int prod = prod_interno(a, a, 5);

 cout << "Produto interno\nde ";
 mostra(a, 5); cout << "por "; mostra(a, 5);

 cout << "vale " << prod << endl;

 cout << "Tamanho da tabela: " << sizeof(a) << endl;
 return (0);
}
```

# Ponteiros e Tabelas

Parâmetro formais de funções: ponteiros e tabelas

O programa anterior produziria a saída:

```
Produto interno
de 5 6 7 8 9
DEBUG: mostra(int[],int) - Tamanho de p: 4

por 5 6 7 8 9
DEBUG: mostra(int[],int) - Tamanho de p: 4

vale 255
Tamanho da tabela: 20
```

Como se pode verificar a função mostra recebe `a`, do tipo `int[4]`, com o valor do endereço do 1º elemento do bloco de 5 inteiros. A atribuição desse valor ao parâmetro formal `int p[]` transforma `p` num valor do tipo  
`int *`

# Aritmética de ponteiros

## Tabelas de caracteres

- Declarando e inicializando:

```
char tab[10];
char * tab_ptr = & tab[0];
```

Então `*tab_ptr` é o mesmo que `tab[0]` e `*tab_ptr + 1` é o mesmo que `tab[1]`.

# Aritmética de ponteiros

## Tabelas de caracteres

- Declarando e inicializando:

```
char tab[10];
char * tab_ptr = & tab[0];
```

Então `*tab_ptr` é o mesmo que `tab[0]` e `*(tab_ptr + 1)` é o mesmo que `tab[1]`.

- De notar que `*(tab_ptr + 1)` é diferente de `(*tab_ptr + 1)`.

Ao escrever `(*tab_ptr + 1)` em primeiro lugar é aplicado o operador de desreferenciação a `tab_ptr` e só depois é adicionado 1, ou seja equivale a `tab[0]+1`.

# Aritmética de ponteiros

## Tabelas de caracteres

- Declarando e inicializando:

```
char tab[10];
char * tab_ptr = & tab[0];
```

Então `*tab_ptr` é o mesmo que `tab[0]` e `* (tab_ptr + 1)` é o mesmo que `tab[1]`.

- De notar que `* (tab_ptr + 1)` é diferente de `(*tab_ptr + 1)`.

Ao escrever `(*tab_ptr + 1)` em primeiro lugar é aplicado o operador de desreferenciação a `tab_ptr` e só depois é adicionado 1, ou seja equivale a `tab[0]+1`.

- O pedaço de programa anterior é equivalente a:

```
char tab[10];
char * tab_ptr = tab; //Conv. impl. char[] para char*
```

porque o valor associado ao nome de uma tabela é igual ao endereço do seu primeiro elemento. Mas `tab` é do tipo `char[]` e não é do tipo `char*`.

# Aritmética de ponteiros

## Tabelas de caracteres

- **(unsigned int) (tab + i) equivale a  
(unsigned int) (tab) + (i \* sizeof(char)).**

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
 char tab[] = "Ana";
 int dim = strlen(tab);
 for(int i = 0; i <= dim; i++){ // Mostra '\0' inclusive
 cout << dec << "\n\ttab[" << i << "] = 0x" << hex << (int) (&tab[i]);
 cout << dec << " (tab + " << i << ") = 0x" << hex << (int) (tab + i);
 cout << dec << " tab[" << i << "] = " << tab[i] << " == "
 << hex << (int) tab[i];
 }
 cout << endl;
}
```

# Aritmética de ponteiros

## Tabelas de caracteres

- ▶ **(unsigned int) (tab + i) equivale a  
(unsigned int) (tab) + (i \* sizeof(char)).**

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
 char tab[] = "Ana";
 int dim = strlen(tab);
 for(int i = 0; i <= dim; i++){ // Mostra '\0' inclusive
 cout << dec << "\n\ttab[" << i << "] = 0x" << hex << (int) (&tab[i]);
 cout << dec << " (tab + " << i << ") = 0x" << hex << (int) (tab + i);
 cout << dec << " tab[" << i << "] = " << tab[i] << " == "
 << hex << (int) tab[i];
 }
 cout << endl;
}
```

- ▶ Como um char ocupa 1 octeto os endereços surgem com valores consecutivos:

```
&tab[0] = 0xbffa341b (tab + 0) = 0xbffa341b tab[0] = A == 41
&tab[1] = 0xbffa341c (tab + 1) = 0xbffa341c tab[1] = n == 6e
&tab[2] = 0xbffa341d (tab + 2) = 0xbffa341d tab[2] = a == 61
&tab[3] = 0xbffa341e (tab + 3) = 0xbffa341e tab[3] = == 0
```

# Aritmética de ponteiros

## Tabelas de inteiros

- `(unsigned int) (tabN + i)` equivale a  
`(unsigned int) tabN + (i * sizeof(int)).`

```
#include <iostream>
using namespace std;
int main(){
 int tabN[] = {2,34,23,1,23,0,45};
 unsigned int dim = sizeof(tabN)/sizeof(int);

 for(unsigned int i = 0; i < dim; i++){ // Mostra todos
 cout << dec << "\n&tabN[" << i << "] = " << hex << (&tabN[i]);
 cout << dec << " (tabN + " << i << ") = " << hex << (tabN + i);
 cout << dec << " tabN[" << i << "] = " << tabN[i] ;
 }
 cout << endl;
}
```

# Aritmética de ponteiros

## Tabelas de inteiros

- `(unsigned int) (tabN + i)` equivale a  
`(unsigned int) tabN + (i * sizeof(int)).`

```
#include <iostream>
using namespace std;
int main(){
 int tabN[] = {2,34,23,1,23,0,45};
 unsigned int dim = sizeof(tabN)/sizeof(int);

 for(unsigned int i = 0; i < dim; i++){ // Mostra todos
 cout << dec << "\n&tabN[" << i << "] = " << hex << (&tabN[i]);
 cout << dec << " (tabN + " << i << ") = " << hex << (tabN + i);
 cout << dec << " tabN[" << i << "] = " << tabN[i];
 }
 cout << endl;
}
```

- Como um int ocupa 4 octetos, endereços de elementos consecutivos em tabN surgem espaçados de 4 unidades:

```
&tabN[0] = 0xbff9598cc (tabN + 0) = 0xbff9598cc tabN[0] = 2
&tabN[1] = 0xbff9598d0 (tabN + 1) = 0xbff9598d0 tabN[1] = 34
&tabN[2] = 0xbff9598d4 (tabN + 2) = 0xbff9598d4 tabN[2] = 23
&tabN[3] = 0xbff9598d8 (tabN + 3) = 0xbff9598d8 tabN[3] = 1
&tabN[4] = 0xbff9598dc (tabN + 4) = 0xbff9598dc tabN[4] = 23
&tabN[5] = 0xbff9598e0 (tabN + 5) = 0xbff9598e0 tabN[5] = 0
&tabN[6] = 0xbff9598e4 (tabN + 6) = 0xbff9598e4 tabN[6] = 45
```

# Aritmética de ponteiros

Exemplo: Tabela de inteiros

- ▶ Conta o número de elementos antes do primeiro zero (ou até terminar), usando ponteiros:

```
#include <iostream>
using namespace std;
int main(){
 int tabN[] = {2,34,23,1,23,0,45};
 unsigned dim = sizeof(tabN)/sizeof(int);
 int * tabN_ptr = &(tabN[0]);
 unsigned i;

 for (i = 0; i < dim && *tabN_ptr != 0 ; i++)
 tabN_ptr++;// Passa ao seguinte

 if(i < dim)
 cout << "\nExistem " << i << " elementos antes do primeiro 0.";
 else
 cout << "\nExistem " << i << " elementos nao nulos.";
 cout << endl;
}
```

# Aritmética de ponteiros

Exemplo: Tabela de inteiros

- ▶ Conta o número de elementos antes do primeiro zero (ou até terminar), usando ponteiros:

```
#include <iostream>
using namespace std;
int main(){
 int tabN[] = {2,34,23,1,23,0,45};
 unsigned dim = sizeof(tabN)/sizeof(int);
 int * tabN_ptr = &(tabN[0]);
 unsigned i;

 for (i = 0; i < dim && *tabN_ptr != 0 ; i++)
 tabN_ptr++;// Passa ao seguinte

 if(i < dim)
 cout << "\nExistem " << i << " elementos antes do primeiro 0.";
 else
 cout << "\nExistem " << i << " elementos nao nulos.";
 cout << endl;
}
```

- ▶ Existem 5 elementos antes do primeiro 0.

# Aritmética de ponteiros

Exemplo: Tabela de double e operador & e \*

- Mostra os elementos menores que 5, usando ponteiros:

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
 double tabD[] = {2.3, 3.4, 5.1, 8.9};
 unsigned dim = sizeof(tabD)/sizeof(double);
 double * tab_ptr = &tabD[0];

 cout << fixed << setprecision(2);

 // Mostra todos os elementos menores que 5 usando o apontador
 for(unsigned i = 0; i < dim; i++, tab_ptr++)
 if(*tab_ptr < 5) cout << *tab_ptr << endl;

 return(0);
}
```

# Aritmética de ponteiros

Exemplo: Tabela de double e operador & e \*

- Mostra os elementos menores que 5, usando ponteiros:

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
 double tabD[] = {2.3, 3.4, 5.1, 8.9};
 unsigned dim = sizeof(tabD)/sizeof(double);
 double * tab_ptr = &tabD[0];

 cout << fixed << setprecision(2);

 // Mostra todos os elementos menores que 5 usando o apontador
 for(unsigned i = 0; i < dim; i++, tab_ptr++)
 if(*tab_ptr < 5) cout << *tab_ptr << endl;

 return(0);
}
```

- Faria surgir no ecrã:

```
2.30
3.40
```

# Exemplo com tabela de char

## Inversão de uma string no estilo C

```
#include <iostream>
#include <cstring>
using namespace std;
void inverte(char a[]);
void inverte_ptr(char a[]);
int main() {
 char nome[] = "Ana Maria";
 cout << nome << endl;
 inverte(nome); //Se inverte(&nome[0]); => conv.impl. de char* p/ char[]
 cout << nome << endl;
 inverte_ptr (nome); //Se inverte_ptr(&nome[0]); => char* p/ char[]
 cout << nome << endl;
 return(0);
}
void inverte(char a[]) {
 char aux;
 for(int i = 0, j = strlen(a) - 1; i < j ; i++, j--) {
 aux = a[i];
 a[i] = a[j];
 a[j] = aux;
 }
}
void inverte_ptr(char a[]) {
 char aux;
 for(char *pi = a, *pj = &a[strlen(a) - 1]; pi < pj; pi++, pj--) {
 aux = *pi;
 *pi = *pj;
 *pj = aux;
 }
}
```

# Ponteiros simples

Exemplo - Inversão de uma string no estilo C

- ▶ O programa anterior apresentaria no ecrã:

```
Ana Maria
airam anA
Ana Maria
```

# Funções: passagem de parâmetros por endereço

- A linguagem C não dispõe do tipo referência. Ou seja em C não possível escrever:

```
int i = 4;
int & i_ref = i; // i_red é uma referência para i
```

Por conseguinte, em C não existe a possibilidade de fazer passagem por referência numa função.

# Funções: passagem de parâmetros por endereço

- A linguagem C não dispõe do tipo referência. Ou seja em C não possível escrever:

```
int i = 4;
int & i_ref = i; // i_ref é uma referência para i
```

Por conseguinte, em C não existe a possibilidade de fazer passagem por referência numa função.

- Na linguagem C só existe passagem por valor!

# Funções: passagem de parâmetros por endereço

- A linguagem C não dispõe do tipo referência. Ou seja em C não possível escrever:

```
int i = 4;
int & i_ref = i; // i_red é uma referência para i
```

Por conseguinte, em C não existe a possibilidade de fazer passagem por referência numa função.

- Na linguagem C só existe passagem por valor!
- **Questão:** Como é que em linguagem C uma função pode modificar o valor duma variável?

# Funções: passagem de parâmetros por endereço

- A linguagem C não dispõe do tipo referência. Ou seja em C não possível escrever:

```
int i = 4;
int & i_ref = i; // i_red é uma referência para i
```

Por conseguinte, em C não existe a possibilidade de fazer passagem por referência numa função.

- Na linguagem C só existe passagem por valor!
- **Questão:** Como é que em linguagem C uma função pode modificar o valor duma variável?
- **Resposta:** Utiliza ponteiros (passados por valor) para aceder a valores de variáveis exteriores à função.

Este mecanismo costuma designar-se por **passagem por endereço** ou **apontador** (mas o que está subjacente é o mecanismo de passagem por valor de um endereço).

# Funções: passagem de parâmetros por endereço

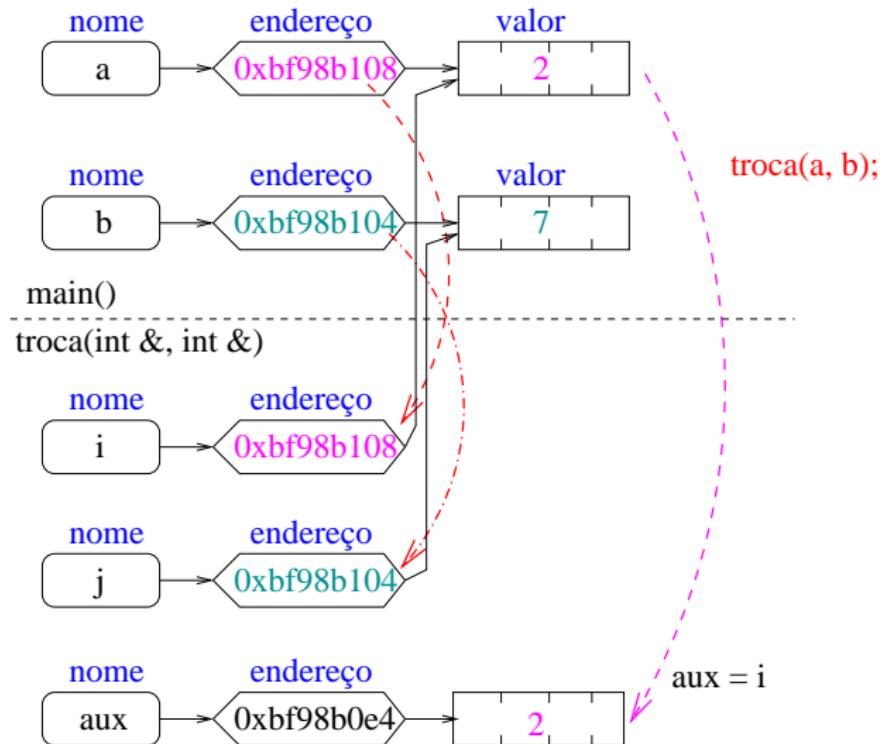
Exemplo: troca do valor de duas variáveis

```
#include <iostream>
using namespace std;
void troca(int & i, int & j);
void troca(int * i_ptr, int * j_ptr);
int main() {
 int a = 2, b = 7;
 cout << "\n(a, b) = (" << a << ", " << b << ")";
 troca(a, b);
 cout << "\nApos \"troca(a, b);\" (a, b) = (" << a << ", " << b << ")";
 troca(&a, &b);
 cout << "\nApos \"troca(&a, &b);\" (a, b) = (" << a << ", " << b << ")";
 return(0);
}
// Troca o valor de i e j, usando referencias
void troca(int & i, int & j) {
 int aux = i;
 i = j;
 j = aux;
}
// Troca o valor de *i_ptr e *j_ptr (usando ponteiros)
void troca(int * i_ptr, int * j_ptr) {
 int aux = *i_ptr;
 *i_ptr = *j_ptr;
 *j_ptr = aux;
}
```

```
(a, b) = (2, 7)
Apos "troca(a, b);" (a, b) = (7, 2)
Apos "troca(&a, &b);" (a, b) = (2, 7)
```

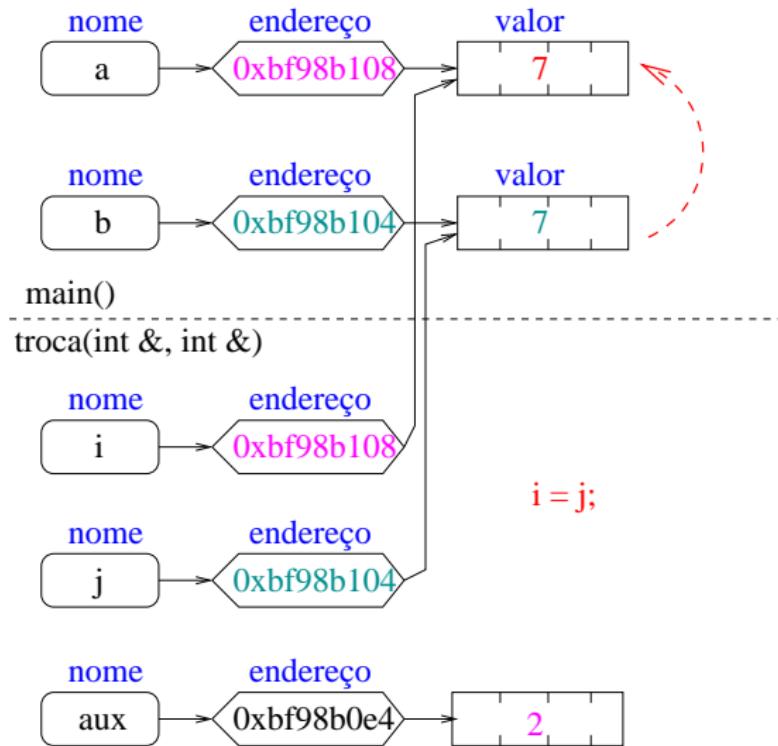
# Funções: passagem de parâmetros por endereço

Exemplo (referência)



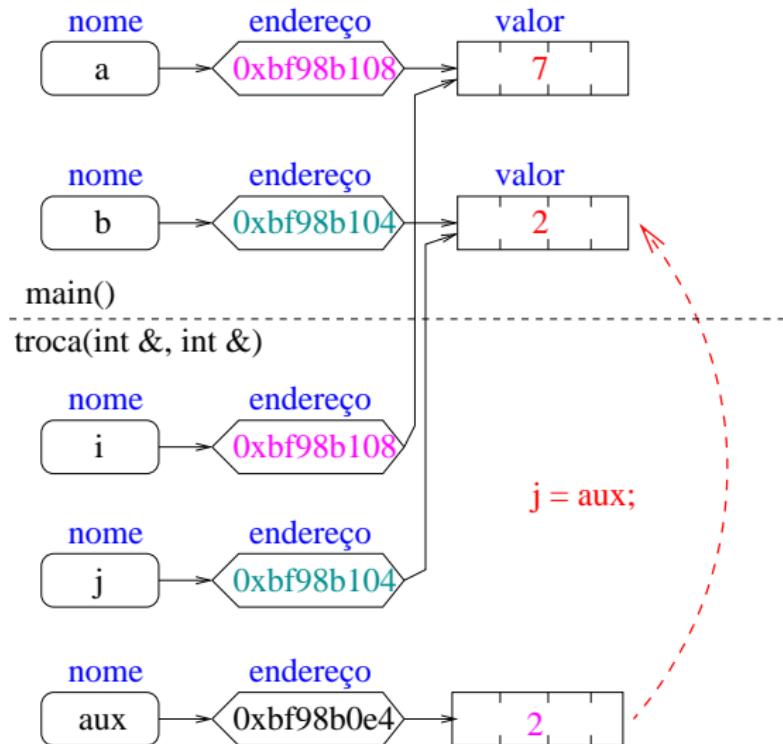
# Funções: passagem de parâmetros por endereço

Exemplo (referência)



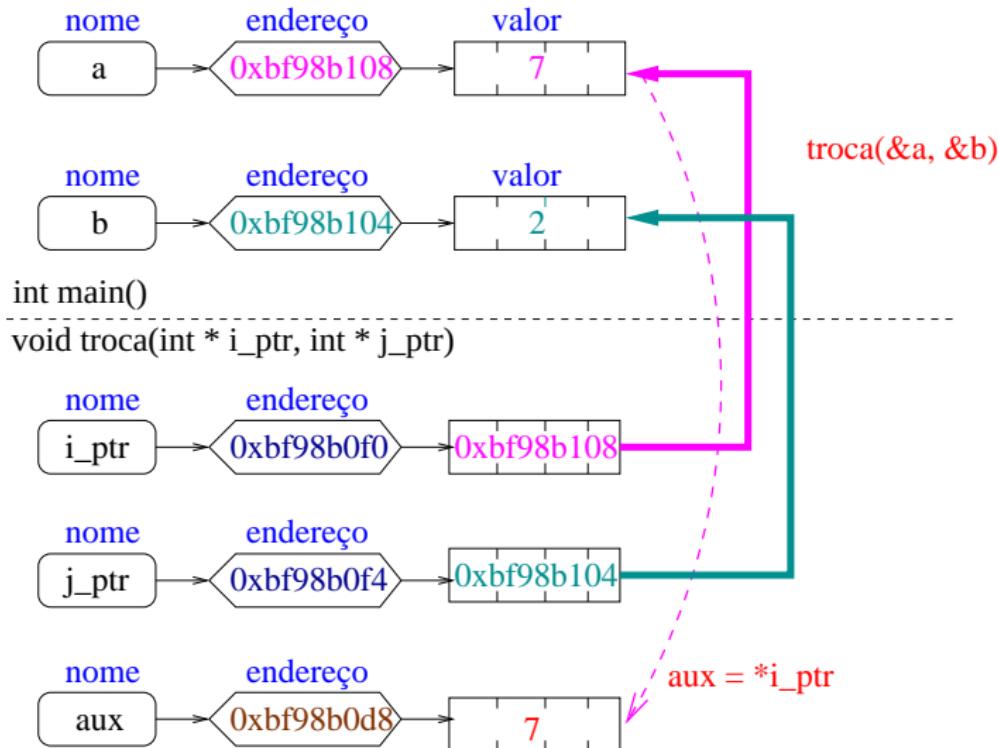
# Funções: passagem de parâmetros por endereço

Exemplo (referência)



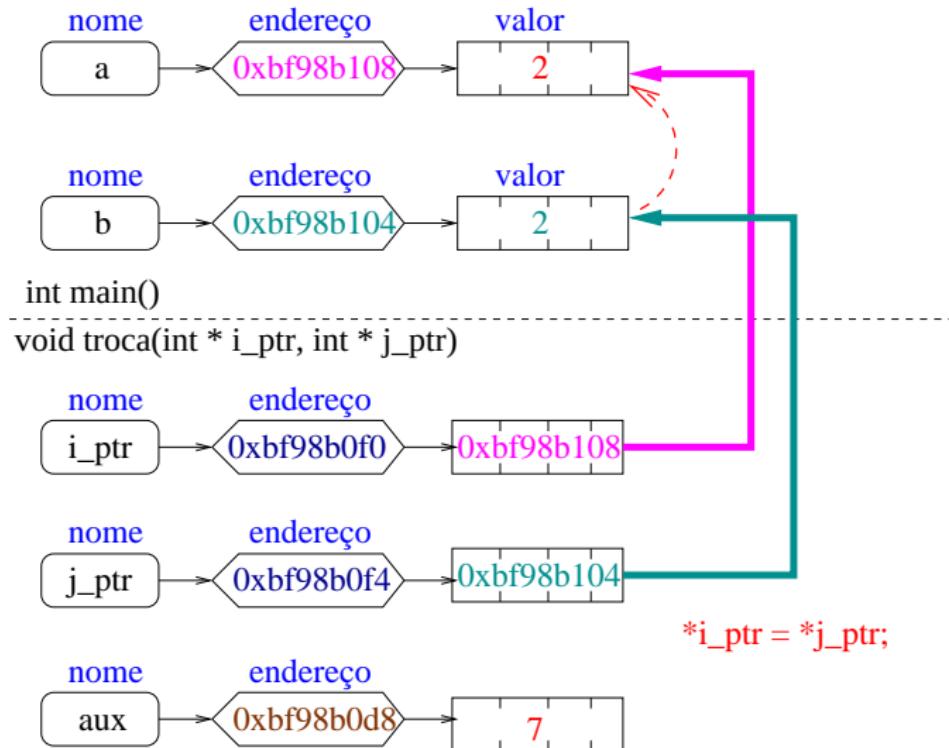
# Funções: passagem de parâmetros por endereço

Exemplo (apontador)



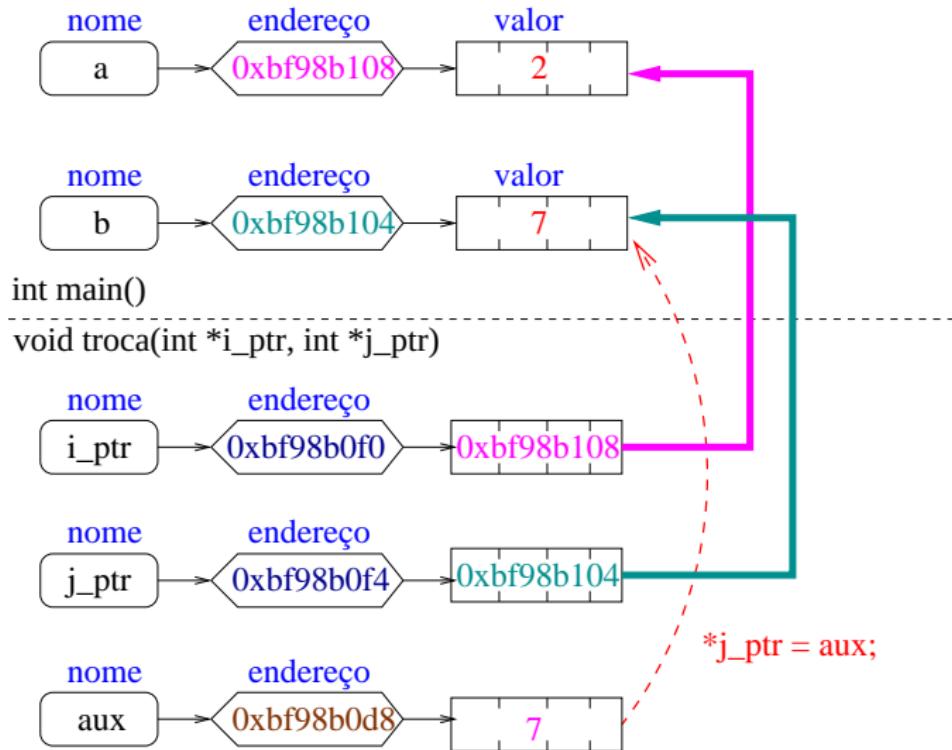
# Funções: passagem de parâmetros por endereço

Exemplo (apontador)



# Funções: passagem de parâmetros por endereço

Exemplo (apontador)



# Entrada/Saída no estilo C

## printf

A forma completa do especificador de formato de *printf*:

%[flags][width][.precision][length]specifier

| specifier | Tipo do argumento                                           | Saída                                      |
|-----------|-------------------------------------------------------------|--------------------------------------------|
| c         | char ou int ou unsigned int                                 | uma letra                                  |
| d         | int ou short int ou char                                    | número decimal inteiro, com sinal          |
| f         | float                                                       | número decimal real, com sinal             |
| lf        | double                                                      | número decimal real, com sinal             |
| u         | unsigned int ou unsigned short int                          | número decimal inteiro, sem sinal          |
| lu        | unsigned long int                                           | número decimal inteiro, sem sinal          |
| s         | char * (C style string)                                     | cadeia de caracteres                       |
| o         | int ou unsigned int                                         | Número inteiro (sem sinal) em octal        |
| x         | int ou unsigned int                                         | Número inteiro (sem sinal) em hexadecimal  |
| e         | float                                                       | Número decimal real no formato $d.ddde+dd$ |
| E         | float                                                       | Número decimal real no formato $d.ddE+dd$  |
| g         | opta entre %e e %f, o que conduzir a um número mais pequeno | Número decimal real                        |
| G         | opta entre %E e %f, o que conduzir a um número mais pequeno | Número decimal real                        |
| %         | Um % seguido de %                                           | permite escrever o carácter %              |

flags Justificação à esquerda, colocar ou não o sinal.

width Número mínimo de caracteres a imprimir (ou No máximo a ler no caso de **scanf**).

precision Número mínimo de dígitos a imprimir.

length Modifica o comprimento do tipo de dados (se possível).

# Entrada/Saída no estilo C

Exemplo: printf

- ▶ Para imprimir um valor (armazenado numa variável ou resultado de uma expressão) a função `printf` requer o símbolo de conversão `%` seguido dos especificadores adequados na string `formato`.  
O C lê o formato e sempre que encontra o símbolo `%d`, toma o parâmetro seguinte (que neste caso deve ser um inteiro) e imprime-o.

# Entrada/Saída no estilo C

Exemplo: printf

- ▶ Para imprimir um valor (armazenado numa variável ou resultado de uma expressão) a função `printf` requer o símbolo de conversão `%` seguido dos especificadores adequados na string formato.  
O C lê o formato e sempre que encontra o símbolo `%d`, toma o parâmetro seguinte (que neste caso deve ser um inteiro) e imprime-o.
- ▶ O seguinte pedaço de código:

```
int i = 7;
printf("Metade (inteira) de %d vale %d.\n", i, i/2);
```

iria resultar na saída (para ecrã):

```
Metade (inteira) de 7 vale 3.
```

# Entrada/Saída no estilo C

scanf

- ▶ `int scanf ( const char * format, ... );` lê dados formatados do `stdin`, de acordo com o parâmetro `format` e armazena a informação na localizações de memória apontados pelos argumentos que se seguem. Esses argumentos adicionais devem apontar para objectos (zona de memória) do tipo especificado pelo formato.
- ▶ A função lê e ignora qualquer carácter designado por *Whitespace* (espaço, tabulações e mudanças de linha), o qual funciona como separador de argumentos a ler (com a excepção do `%c`). Retorna o número de argumentos com valor atribuído.

# Entrada/Saída no estilo C

## scanf

- A forma completa do especificador de formato (alguns precedidos da *length*, por exemplo "l"): %[\*][width][length]specifier  
Se usado o \*, a função lê do *buffer* sem a armazenar em qualquer argumento.
- Lista de *alguns* especificadores de formato de *scanf*.

| Specifier  | Tipo do argumento             | Funcionamento                                                                                                                                                                 |
|------------|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| i          | int unsigned int              | Lê qualquer número de dígitos, opcionalmente precedidos de sinal (+ ou -). Por por omissão os dígitos (0-9), o prefixo 0 indica octal (0-7), e o prefixo 0x hexadecimal (0-f) |
| d, u       | int, unsigned int             | Lê qualquer número de dígitos (0-9), opcionalmente precedidos de sinal (+ ou -). d número pode ter sinal, u número sem sinal                                                  |
| o          | inteiro na base octal         | pode ter sinal                                                                                                                                                                |
| x          | inteiro na base hexadecimal   | pode ter sinal                                                                                                                                                                |
| f, e, g    | número em vírgula flutuante   | float: número decimal real, com sinal                                                                                                                                         |
| lf, le, lg | número em vírgula flutuante   | double: número decimal real, com sinal                                                                                                                                        |
| lu         | unsigned long int             | número decimal inteiro, sem sinal                                                                                                                                             |
| s          | char * (C style string)       | cadeia de caracteres, pára de ler no primeiro <i>Whitespace</i> . Coloca no final o carácter de terminação.                                                                   |
| c          | Se width omissão ou igual a 1 | lê um carácter                                                                                                                                                                |
|            | Se width diferente de 1       | lê exactamente width caracteres, e guarda-os no local apontado (sem carácter de terminação)                                                                                   |

# Entrada/Saída no estilo C

## Exemplo

```
#include <cstdio>
int main ()
{
 char apelido [80];
 int i;
 double peso;
 // scanf e %s só lê uma palavra
 printf ("Apelido: "); // scanf tem espaço, tab ou \n como separador
 scanf ("%79s", apelido); // ZZZ não pode ter mais de 79 letras!

 printf ("Idade: ");
 scanf ("%d", &i);

 printf ("Peso: ");
 scanf ("%lf", &peso);

 printf ("%s, pesa %.1lf e tem %d anos.\n", apelido, peso, i);
}
```

```
Apelido: Amaral
Idade: 45
Peso: 78.87
Amaral, pesa 78.9 e tem 45 anos.
```

De notar a **passagem de parâmetros por endereço** na função scanf.

# Entrada/Saída no estilo C

printf/scanf

- ▶ Estas funções não verificam se os tipos dos parâmetros e dos formatos de conversão são adequados!

# Entrada/Saída no estilo C

## printf/scanf

- ▶ Estas funções não verificam se os tipos dos parâmetros e dos formatos de conversão são adequados!
- ▶ O seguinte pedaço de código:

```
float x = 7.5;
printf("Metade (inteira) de %d vale %.1f.\n", x, x/2);
```

Se compilado ignorando os **aviso**s do compilador, irá resultar numa saída (para ecrã) em que o número apresentado (inteiro) é imprevisível. Isto acontece porque `x` e `x/2` são **float** e o formato de conversão utilizado apenas é válido para inteiros.

# Entrada/Saída no estilo C

## printf/scanf

- A função `scanf` permite ler uma string contendo espaços (formato `%[^\\n]`) e permite ler vários argumentos:

```
#include <cstdio>
using namespace std;
int main ()
{
 char texto[1000];
 printf("De-me texto:\n");
 // %[^\\n] para ler uma frase (c/ espaços) \\n fica no buffer
 scanf("%999[^\\n]*c","", texto); // *c remove \\n do buffer
 printf("Deu-me:\\n%s\\n\\n", texto);

 float x;
 int i;
 printf("De-me um inteiro, seguido de um float:\\n");
 scanf("%d %f", &i, &x); // deixa \\n no buffer
 printf("Deu-me\\ni= %d\\nx= %.2f\\n", i, x);
}
```

De notar a chamada por endereço ou apontador.

# Entrada/Saída no estilo C

## printf/scanf

- ▶ Um exemplo da execução do programa anterior:

```
De-me texto:
```

```
A sabedoria é filha da experiência. Leonardo da Vinci
```

```
Deu-me:
```

```
A sabedoria é filha da experiência. Leonardo da Vinci
```

```
De-me um inteiro, seguido de um float:
```

```
6 7.25525
```

```
Deu-me
```

```
i= 6
```

```
x= 7.26
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- ▶ Um ponteiro é declarado colocando um asterisco “\*” antes do nome da variável no momento da declaração:

```
struct complexo {
 float re; // parte real
 float im; // parte imaginaria
} num = {2.3, 5.4};
struct complexo * num_ptr = #
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- ▶ Um ponteiro é declarado colocando um asterisco “\*” antes do nome da variável no momento da declaração:

```
struct complexo {
 float re; // parte real
 float im; // parte imaginaria
} num = {2.3, 5.4};
struct complexo * num_ptr = #
```

- ▶ Já vimos que para acedermos ao membro `re` e `im` de uma variável do tipo `struct complexo` temos de escrever: `num.re` e `num.im`, respectivamente.

# Ponteiros e estruturas

Sintaxe: o operador `->`

- ▶ Um ponteiro é declarado colocando um asterisco “`*`” antes do nome da variável no momento da declaração:

```
struct complexo {
 float re; // parte real
 float im; // parte imaginaria
} num = {2.3, 5.4};
struct complexo * num_ptr = #
```

- ▶ Já vimos que para acedermos ao membro `re` e `im` de uma variável do tipo `struct complexo` temos de escrever: `num.re` e `num.im`, respectivamente.
- ▶ Dado um ponteiro para uma estrutura, no exemplo `num_ptr`, o acesso aos membros da estrutura apontada é feito usando “`->`” em vez do “`.`”:

```
num_ptr->re = 3.7; // <=> num.re = 3.7
num_ptr->im = 1.0; // <=> num.im = 1.0
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- Retomando a declaração:

```
struct complexo {
 float re; // parte real
 float im; // parte imaginaria
};
// Dois complexos com valores
a ={2.2, 3.3}, b={5.7, 7.7};
// dois ponteiros para complexo
struct complexo * tab[2];
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- Retomando a declaração:

```
struct complexo {
 float re; // parte real
 float im; // parte imaginaria
};
// Dois complexos com valores
a ={2.2, 3.3}, b={5.7, 7.7};
// dois ponteiros para complexo
struct complexo * tab[2];
```

- Apenas para ilustrar a sintaxe:

```
tab[0] = &a; // tab[0] aponta para a
tab[1] = &b; // tab[1] aponta para b
// mostrando valores apontados por tab[i], i=0,1
for(int i = 0; i < 2; i++)
 cout << tab[i]->re << " + i"
 << tab[i]->im << '\n';
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- ▶ Um ponteiro é declarado colocando um asterisco “\*” antes do nome da variável no momento da declaração:

```
struct produto{
 string nome;
 float custo;
};
strut produto ovo = {"Ovos", 0.99};
struct produto *ovo_ptr = & ovo;
```

# Ponteiros e estruturas

Sintaxe: o operador ->

- ▶ Um ponteiro é declarado colocando um asterisco “\*” antes do nome da variável no momento da declaração:

```
struct produto{
 string nome;
 float custo;
};
struct produto ovo = {"Ovos", 0.99};
struct produto *ovo_ptr = & ovo;
```

- ▶ Para alterar o custo de ovo temos (neste caso) três possibilidades:

```
ovo.custo = 0.79; // atraves de ovo
(*ovo_ptr).custo = 0.67; // atraves de *ovo_ptr
ovo_ptr->custo = 0.75; // atraves de ovo_ptr
```

# Ponteiros e estruturas

Ponteiros e estruturas – sintaxe. Um exemplo.

```
#include <iostream>
using namespace std;

struct produto {
 std::string nome;
 double custo;
};

void mostra(const struct produto * ptr) {
 std::cout << "\nO produto " << ptr->nome << " custa " << (*ptr).custo;
}

void mostra(const struct produto um) {
 std::cout << "\nO produto " << um.nome << " custa " << um.custo;
}

int main(){
 struct produto ovo = {"Ovos", 0.99}; // inicializa valor
 std::cout << "\nInicialmente..." ;
 mostra(ovo); // por endereço
 ovo.custo = 0.79; // baixa preço
 std::cout << "\n\nDepois baixar o custo:";
 mostra(ovo); // por valor
 cout << endl;
 return(0);
}
```

# Ponteiros e estruturas

Ponteiros e estruturas – sintaxe. Um exemplo.



```
Inicialmente...
O produto Ovos custa 0.99

Depois baixar o custo:
O produto Ovos custa 0.79
```

# Ponteiros e estruturas

Ponteiros e estruturas – sintaxe. Um exemplo.



```
Inicialmente...
O produto Ovos custa 0.99

Depois baixar o custo:
O produto Ovos custa 0.79
```

- ▶ De notar que os parêntesis em  
`(*ptr).custo` são indispensáveis!  
O operador `.` tem precedência sobre o operador de desreferenciação `*`.

# Ponteiros e estruturas

Ponteiros e estruturas – sintaxe. Um exemplo.



```
Inicialmente...
O produto Ovos custa 0.99

Depois baixar o custo:
O produto Ovos custa 0.79
```

- ▶ De notar que os parêntesis em  
`(*ptr).custo` são indispensáveis!  
O operador `.` tem precedência sobre o operador de desreferenciação `*`.
- ▶ A instrução `*ptr.custo`  
**daria erro de compilação** pois equivale a  
`* (ptr.custo)`

# Ponteiros e Estruturas

## Tabela de ponteiros

- ▶ Considere a seguinte estrutura:

```
struct correio {
 std::string nome; // Apelido, primeiro nome
 std::string endl; // Duas linhas para o endereço
 std::string end2; // Duas linhas para o endereço
 std::string cidade; // Nome da cidade
 std::string cpostal; // Código postal
} list[MAX_ENTRADAS]; // Tabela com muitas moradas.
```

# Ponteiros e Estruturas

## Tabela de ponteiros

- ▶ Considere a seguinte estrutura:

```
struct correio {
 std::string nome; // Apelido, primeiro nome
 std::string endl; // Duas linhas para o endereço
 std::string end2; // Duas linhas para o endereço
 std::string cidade; // Nome da cidade
 std::string cpostal; // Código postal
} list[MAX_ENTRADAS]; // Tabela com muitas moradas.
```

- ▶ Esta estrutura é pesada! Defina-se então um vector de ponteiros para essa estrutura:

```
struct correio * list_ptrs[MAX_ENTRADAS];
e inicialize-se:

int entradas_info; // num. de entradas com info.
.
.
.
for (int i = 0 ; i < entradas_info; i++)
 list_ptrs[i] = & list[i];

// Agora para basta ordenar lista_ptr, para aceder
// de forma ordenada aos elementos de lista
.
.
```

# Ponteiros e Estruturas

## Tabela de ponteiros

- ▶ Considere a seguinte estrutura:

```
struct correio {
 std::string nome; // Apelido, primeiro nome
 std::string endl; // Duas linhas para o endereço
 std::string end2; // Duas linhas para o endereço
 std::string cidade; // Nome da cidade
 std::string cpostal; // Código postal
} list[MAX_ENTRADAS]; // Tabela com muitas moradas.
```

- ▶ Esta estrutura é pesada! Defina-se então um vector de ponteiros para essa estrutura:

```
struct correio * list_ptrs[MAX_ENTRADAS];
e inicialize-se:

int entradas_info; // num. de entradas com info.
.
.
for (int i = 0 ; i < entradas_info; i++)
 list_ptrs[i] = & list[i];

// Agora para basta ordenar lista_ptr, para aceder
// de forma ordenada aos elementos de lista
.
.
```

- ▶ O livro (Ouaille 2003), tem várias gralhas na página 235:

- ▶ faz `++list_ptr`, o que é ilegal em C++, pois o valor associado ao nome de um array é constante e não pode ser alterado.
- ▶ `current = number_of_entries` em vez de `current < number_of_entries`

# Os operadores new e delete

## Exemplo 1

```
#include <iostream>
using namespace std;
int main(){
 int i = 5; // um inteiro
 int * ptr = &i; // ptr aponta para i;

 cout << "&i == ptr : " << &i << " == " << ptr;
 cout << "\n i == *ptr : " << i << " == " << *ptr;
 ptr = NULL; // ptr não aponta pra nada

 ptr = new int ; // ptr aponta para zona onde cabe um inteiro
 *ptr = 7; // coloca o valor 7 nessa zona de memória
 cout << "\n&i != ptr : " << &i << " != " << ptr;
 cout << "\n i != *ptr : " << i << " != " << *ptr << endl;
 delete ptr; // Liberta espaço
 return(0);
}
```

```
&i == ptr : 0xbff90c8d4 == 0xbff90c8d4
i == *ptr : 5 == 5
&i != ptr : 0xbff90c8d4 != 0x804a008
i != *ptr : 5 != 7
```

# Os operadores new e delete

## Exemplo 3

- Exemplo criando uma tabela com o tamanho estritamente necessário:

```
#include <iostream>
using namespace std;
int main(){
 int n;
 do {
 cout << "Quantos numeros que ler [1-10]? "; cin >> n;
 } while (n <= 0 || n > 10);

 // Cria tabela de dimensão n
 int * tab = new int[n]; // pede 'a heap um bloco com n*sizeof(int)

 cout << "De-me os " << n << " numeros\n";
 for (int i = 0; i < n; i++) cin >> tab[i];
 cout << "Mostra os " << n << " numeros\n";
 for (int i = 0; i < n-1; i++) cout << tab[i] << ", ";
 cout << tab[n-1] << endl;
 delete[] tab; // Liberta a memória reservada
 return(0);
}
```

```
Quantos numeros que ler [1-10]? 3
De-me os 3 numeros
23 56 78
Mostra os 3 numeros
23, 56, 78
```

# Os operadores new e delete

## Exemplo 1

- Na linguagem norma C++11 **NÃO É PERMITIDO FAZER:**

```
#include <iostream>
using namespace std;
int main(){
 int n;
 do {
 cout << "Quantos numeros que ler [1-10]? ";
 cin >> n;
 } while (n <= 0 || n > 10);

 // Cria tabela de dimensão n - funciona mas não em C++ ansi estrito
 int tab[n];

 cout << "De-me os " << n << " numeros\n";
 for (int i = 0; i < n; i++) cin >> tab[i];
 cout << "Mostra os " << n << " numeros\n";
 for (int i = 0; i < n-1; i++) cout << tab[i] << ", ";
 cout << tab[n-1] << endl;
 //delete[] tab; return(0);
}
```

A definição de uma tabela: `int tab[n];` em que

`n` **não é uma constante** **não é permitido na norma C++11**  
e **não é permitido** **nesta disciplina.**

# Regras de Precedência

Ver Apêndice C de Oualline 2003.

| Precedência | Operadores        |                   |     |      |        |
|-------------|-------------------|-------------------|-----|------|--------|
| 1           | ()                | []                | ->  | .    |        |
|             | ::                | ::*               | ->* | .*   |        |
| 2           | !                 | ~                 | ++  | --   | (tipo) |
|             | - (unário)        | * (desreferência) |     |      |        |
|             | & (endereço de)   | sizeof            |     |      |        |
| 3           | * (multiplicação) | /                 | %   |      |        |
| 4           | +                 | -                 |     |      |        |
| 5           | <<                | >>                |     |      |        |
| 6           | <                 | <=                | >   | >=   |        |
| 7           | <=                | !=                |     |      |        |
| 8           | & (E bit a bit)   |                   |     |      |        |
| 9           | ^                 |                   |     |      |        |
| 10          |                   |                   |     |      |        |
| 11          | &&                |                   |     |      |        |
| 12          |                   |                   |     |      |        |
| 13          | ?:                |                   |     |      |        |
| 14          | =                 | +=                | -=  | etc. |        |
| 15          | ,                 |                   |     |      |        |

Regras práticas de precedência (como sugerido no vosso livro):

| Precedência | Operadores        |   |   |
|-------------|-------------------|---|---|
| 3           | * (multiplicação) | / | % |
| 4           | +                 | - |   |

E nos restantes casos, utilize parentêsis!

# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C

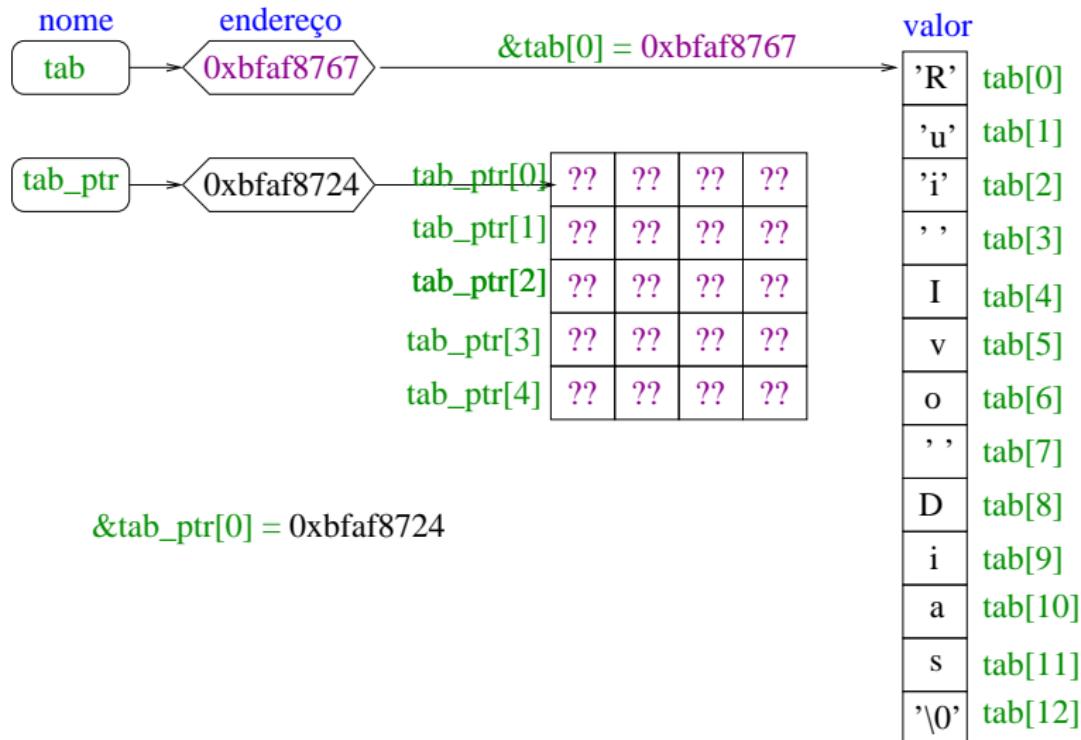
- ▶ No livro está como partir um string em duas. Aqui vamos partir em palavras uma string no estilo C.

```
#include <iostream>
#include <cstring>
using namespace std;
const int MAX_WORD = 5; // Numero maximo de palavras a separar
const char ESPACO = ' '; // Para nao escrever '
int main(){
 char tab[] = "Rui Ivo Dias";
 char * tab_ptr[MAX_WORD]; // Tabela de ponteiros s/ inicializar
 int dim = strlen(tab); // Numero de caracteres antes de '\0'
 int plv = 0; // contador de palavras

 // Inicio da primeira palavra - preve tab[0] == '\0'
 if (tab[0] != '\0'){
 tab_ptr[plv] = tab; // Ponteiro para a primeira palavra
 plv++; // Mais uma palavra
 }
 // Termina palavra anterior. Ponteiro para palavra seguinte
 for (int i=1; i < dim && plv < MAX_WORD; i++)
 if (tab[i-1] == ESPACO && tab[i] != ESPACO) { // Mais um palavra
 tab[i-1] = '\0'; // Termina a palavra anterior
 tab_ptr[plv] = &tab[i]; // Ponteiro para palavra seguinte
 plv++; // Mais uma palavra
 }
 // Mostra as palavras encontradas
 for (int i=0; i < plv; i++)
 cout << tab_ptr[i] << endl;
 return(0);
}
```

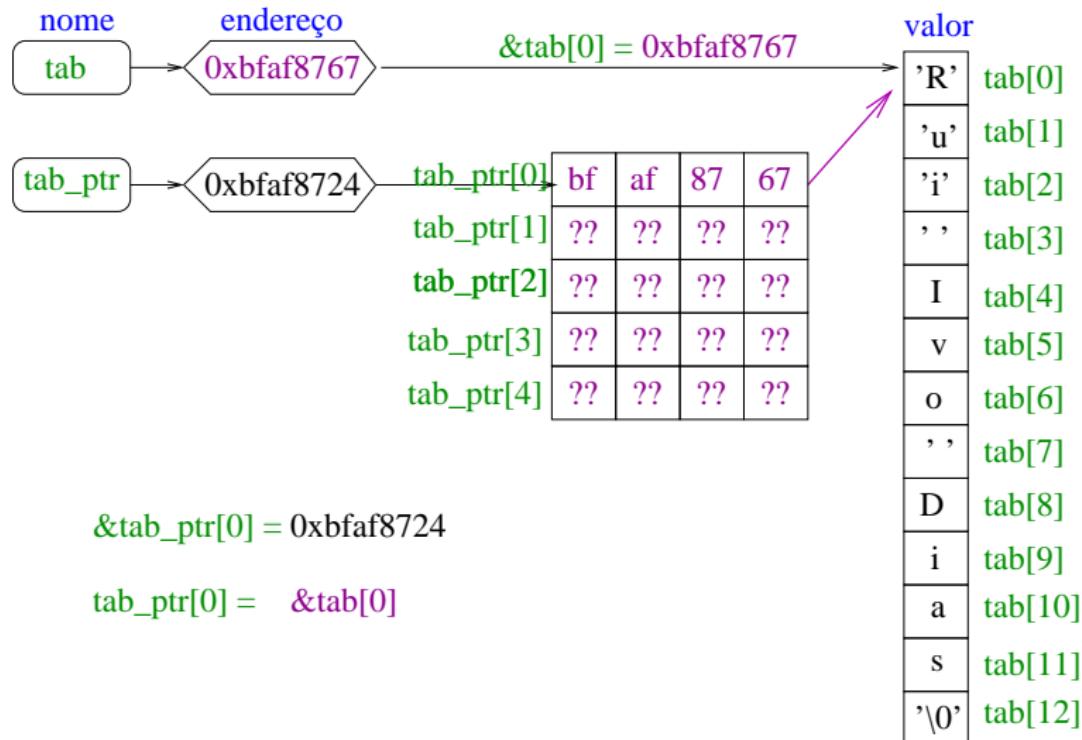
# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



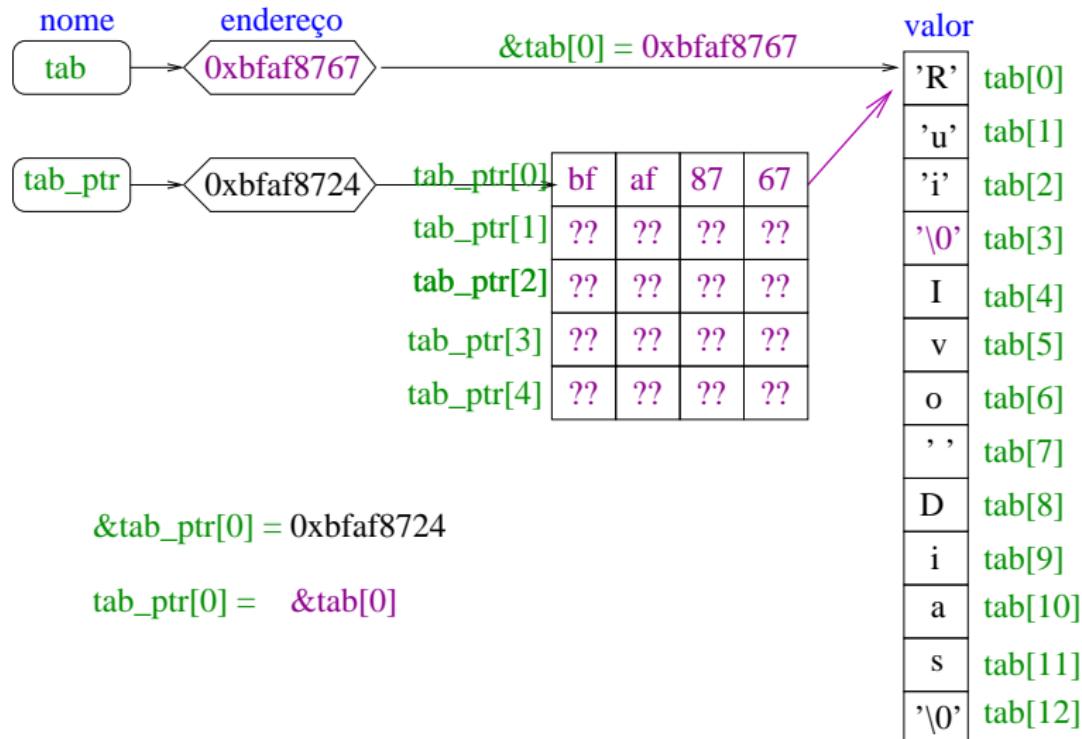
# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



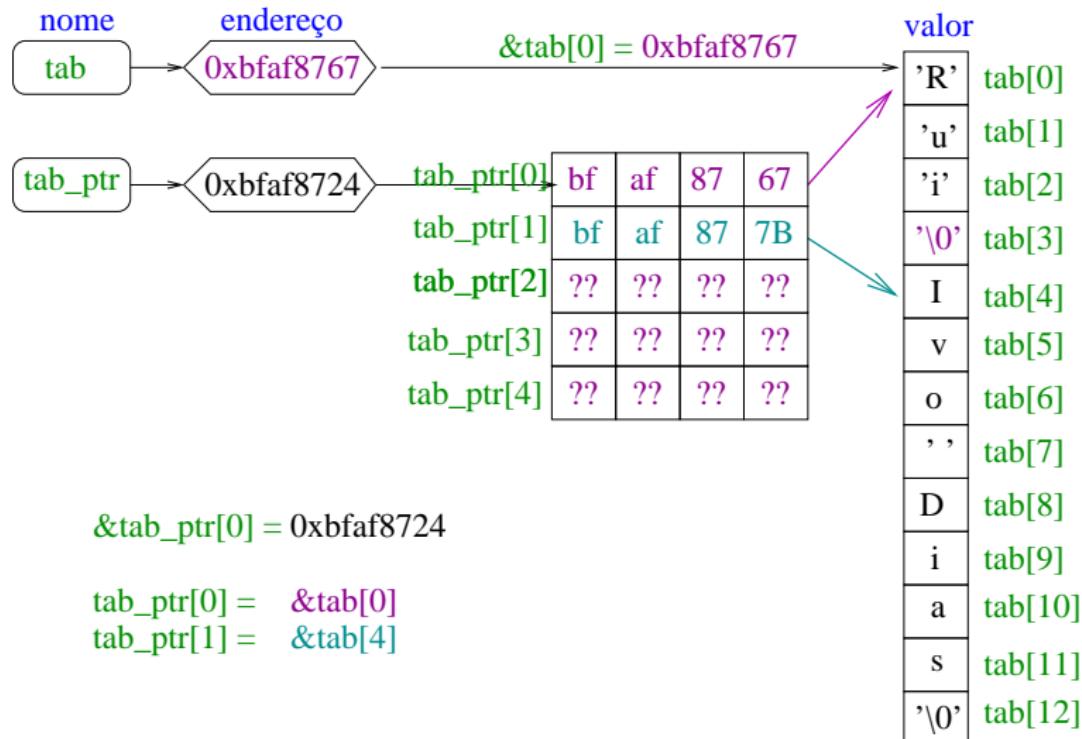
# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



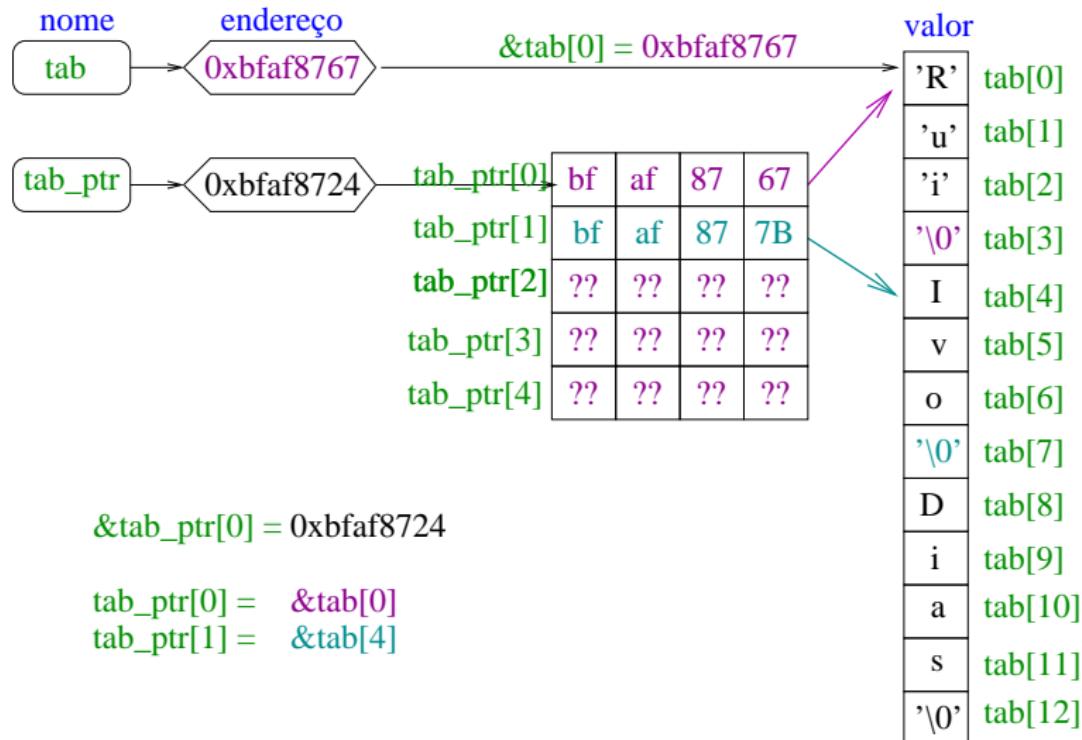
# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



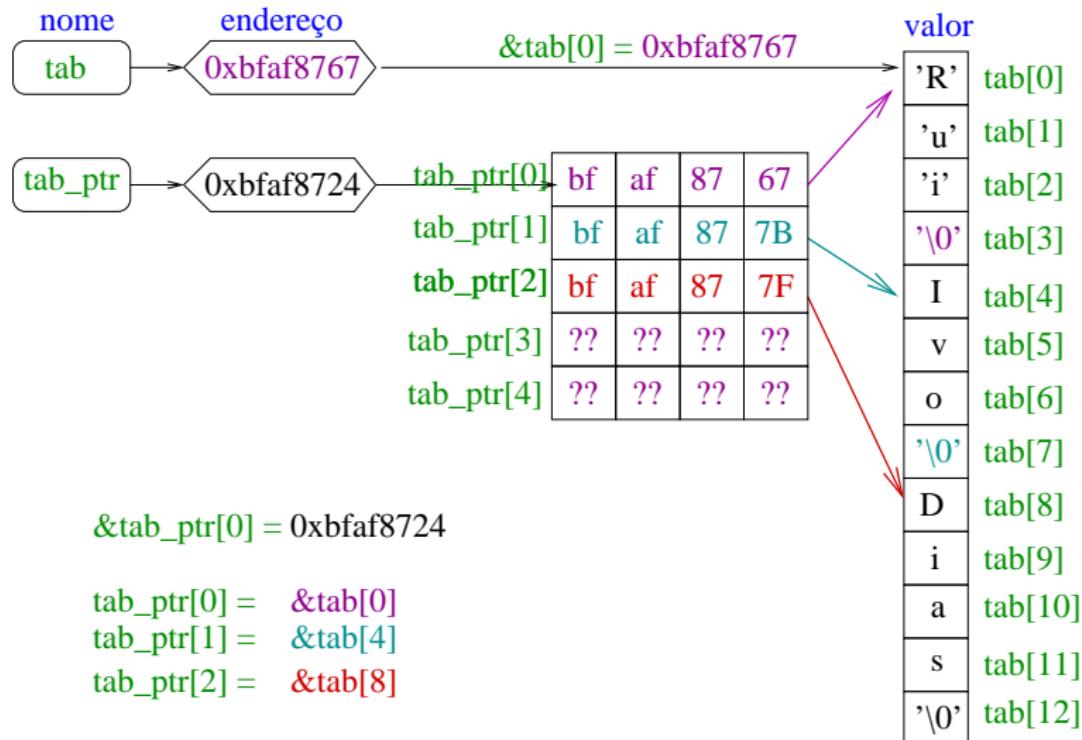
# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C - graficamente



# Ponteiros simples

Exemplo: Partir em palavras uma string no estilo C



Rui  
Ivo  
Dias

# Argumentos na linha de comando

- A função `main()` pode ter dois argumentos, os quais costumam ser designados por `argc` e `argv`.

```
int main(int argc, char * argv[]) { ... }
```

**Dica:** usando estas designações a ordem dos parâmetros é dada pela sua ordem alfabética. Além disso a letra **c** em `argc` lembra-nos que o primeiro é um **contador** e a letra **v** em `argv` lembra-nos que o comando pode ser **verboroso**.

# Argumentos na linha de comando

- ▶ A função `main()` pode ter dois argumentos, os quais costumam ser designados por `argc` e `argv`.

```
int main(int argc, char * argv[]) { ... }
```

**Dica:** usando estas designações a ordem dos parâmetros é dada pela sua ordem alfabética. Além disso a letra **c** em `argc` lembra-nos que o primeiro é um **contador** e a letra **v** em `argv` lembra-nos que o comando pode ser **verboroso**.

- ▶ `argc` diz-nos o número de argumentos na linha de comando, incluindo o nome do próprio programa (logo `argc` é sempre igual ou superior a 1).

# Argumentos na linha de comando

- ▶ A função `main()` pode ter dois argumentos, os quais costumam ser designados por `argc` e `argv`.

```
int main(int argc, char * argv[]) { ... }
```

**Dica:** usando estas designações a ordem dos parâmetros é dada pela sua ordem alfabética. Além disso a letra **c** em `argc` lembra-nos que o primeiro é um **contador** e a letra **v** em `argv` lembra-nos que o comando pode ser **verboroso**.

- ▶ `argc` diz-nos o número de argumentos na linha de comando, incluindo o nome do próprio programa (logo `argc` é sempre igual ou superior a 1).
- ▶ `argv` contém a lista de argumentos, incluindo o nome do próprio programa, apontado por `argv[0]`.

# Argumentos na linha de comando

## Exemplo 1

```
► #include <iostream>
using namespace std;

// Experimentando a linha de comando
int main(int argc, char * argv[]){

 cout << "O comando digitado foi: \n";
 for(int i =0; i < argc; i++)
 cout << argv[i] << ' ';
 cout << endl;
 return(0);
}
```

# Argumentos na linha de comando

## Exemplo 1

►

```
#include <iostream>
using namespace std;

// Experimentando a linha de comando
int main(int argc, char * argv[]){

 cout << "O comando digitado foi: \n";
 for(int i =0; i < argc; i++)
 cout << argv[i] << ' ';
 cout << endl;
 return(0);
}
```

►

```
./argcv.exe<enter>
O comando digitado foi:
./argcv.exe
```

```
./argcv.exe Isto e -45 um teste<enter>
O comando digitado foi:
./argcv.exe Isto e -45 um teste
```

# Argumentos na linha de comando

## Exemplo 1

```
► #include <iostream>
using namespace std;

// Experimentando a linha de comando
int main(int argc, char * argv[]){

 for(int i =0; i < argc; i++)
 cout << argv[i] << '\n';
 return(0);
}
```

# Argumentos na linha de comando

## Exemplo 1

- ▶ 

```
#include <iostream>
using namespace std;

// Experimentando a linha de comando
int main(int argc, char * argv[]){

 for(int i =0; i < argc; i++)
 cout << argv[i] << '\n';
 return(0);
}
```

- ▶ 

```
./argcv.exe Isto e -45 um teste<enter>
./argcv.exe
Isto
e
-45
um
teste
```

# Argumentos na linha de comando

## Exemplo 2 (bibpdc.h)

- ▶ Considere que colocou os seguintes protótipos de funções (definidas no ficheiro “`bibpdc.cpp`”) no ficheiro “`bibpdc.h`”).

# Argumentos na linha de comando

## Exemplo 2 (bibpdc.h)

- ▶ Considere que colocou os seguintes protótipos de funções (definidas no ficheiro “`bibpdc.cpp`”) no ficheiro “`bibpdc.h`”).
- ▶ Em `bibpdc.h` está toda a informação necessária à sua utilização por um programa.

```
#ifndef _BIBPDC_H

// Mostra numeros primos <= n, se n > 0
// Comeca numa nova linha e nao muda de linha no fim
void mostraprimos(int n);

// Devolve true se o numero n > 0 e' primo
// e false caso contrario
// Aborta se n <= 0
bool primo(int n);

// Devolve o factorial de n se n>=0
// Aborta se n < 0
int factorial(int n);

#define _BIBPDC_H
#endif
```

# Argumentos na linha de comando

## Exemplo 3

- ▶ Considere que deseja implementar um [programa que dado um inteiro na linha de comando](#):
  - ▶ informa se este é um número primo,
  - ▶ apresenta o respectivo factorial,
  - ▶ e seguidamente todos os primos menores ou iguais ao número dado.

O inteiro deve ser compreendido entre 1 e 9.

Este programa deverá usar as funções cujos protótipos estão em [bibpdc.h](#)

# Argumentos na linha de comando

## Exemplo 3

- ▶ Considere que deseja implementar um [programa que dado um inteiro na linha de comando](#):
  - ▶ informa se este é um número primo,
  - ▶ apresenta o respectivo factorial,
  - ▶ e seguidamente todos os primos menores ou iguais ao número dado.

O inteiro deve ser compreendido entre 1 e 9.

Este programa deverá usar as funções cujos protótipos estão em [bibpdc.h](#)

- ▶ O objectivo é utilizar a [linha de comando para ler o número](#). Se o parâmetro não for dado ou for inadequado, devem ser produzidas mensagens de aviso adequadas.

# Argumentos na linha de comando

## Exemplo 2 (main())

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include "bibpdc.h"
using namespace std; // Compilando por partes e usando a linha de comando
int main(int argc, char * argv[]){
 switch (argc) {
 case 1 : cout << "Esqueceu-se do parametro no intervalo [1,9]\n";
 return(1);
 case 2 : if (strlen(argv[1]) > 1) {
 cout << "O parametro deve ser apenas um digito no intervalo [1,9]\n";
 return(2);
 } else if (!isdigit(argv[1][0]) || argv[1][0] == '0') {
 cout << "O parametro deve ser um digito no intervalo [1,9]\n";
 return(3);
 }
 break;
 default: cout << "Excesso de parametros\n";
 cout << "O parametro deve ser apenas um unico digito no intervalo [1,9]\n";
 return(4);
 }
 // Ha' um so parametro e e' um digito.
 int n = atoi(&argv[1][0]); // Ou atoi(argv[1]);
 int f = factorial(n);
 cout << "\nO numero " << argv[1][0];
 if (primo(n))
 cout << " e' primo e o seu factorial vale " << f << '.' << endl;
 else
 cout << " nao e' primo e o seu factorial vale " << f << '.' << endl;
 cout << "Os primos menores ou iguais a " << argv[1][0] << ":";
 mostraprimos(n); cout << endl; return(0);
}
```

# Argumentos na linha de comando

## Exemplo 2 (compilando por partes)

- ▶ Compila-se o ficheiro com a função main():

```
g++ -c principal.cpp
criando principal.o
```

# Argumentos na linha de comando

## Exemplo 2 (compilando por partes)

- ▶ Compila-se o ficheiro com a função main():

```
g++ -c principal.cpp
criando principal.o
```

- ▶ Seja `bibpdc.cpp` o ficheiro com a definição das funções cujos protótipos estão em `bibpdc.h`.

# Argumentos na linha de comando

## Exemplo 2 (compilando por partes)

- ▶ Compila-se o ficheiro com a função main():

```
g++ -c principal.cpp
criando principal.o
```

- ▶ Seja `bibpdc.cpp` o ficheiro com a definição das funções cujos protótipos estão em `bibpdc.h`.

- ▶ Compila-se o ficheiro `bibpdc.h`:

```
g++ -c bibpdc.cpp
criando bibpdc.o
```

# Argumentos na linha de comando

## Exemplo 2 (compilando por partes)

- ▶ Compila-se o ficheiro com a função main():

```
g++ -c principal.cpp
criando principal.o
```

- ▶ Seja bibpdc.cpp o ficheiro com a definição das funções cujos protótipos estão em bibpdc.h.

- ▶ Compila-se o ficheiro bibpdc.h:

```
g++ -c bibpdc.cpp
criando bibpdc.o
```

- ▶ Finalmente podemos criar o executável:

```
g++ -o principal principal.o bibpdc.o
criando principal
```

# Argumentos na linha de comando

## Exemplo 2 (compilando por partes)

- ▶ Compila-se o ficheiro com a função main():

```
g++ -c principal.cpp
criando principal.o
```

- ▶ Seja bibpdc.cpp o ficheiro com a definição das funções cujos protótipos estão em bibpdc.h.

- ▶ Compila-se o ficheiro bibpdc.h:

```
g++ -c bibpdc.cpp
criando bibpdc.o
```

- ▶ Finalmente podemos criar o executável:

```
g++ -o principal principal.o bibpdc.o
criando principal
```

- ▶ Em alternativa podíamos fazer a compilação e a ligação num só comando:

```
g++ -o principal principal.cpp bibpdc.cpp
```

# Argumentos na linha de comando

## Exemplo 2 (bibpdc.cpp)

```
#include "bibpdc.h"
#include <iostream>
#include <cassert>
#include <cmath>
// Mostra numeros primos <= n, se n > 0.
// Comeca numa nova linha e nao muda de linha no fim.
void mostraprimos(int n) {
 std::cout << "\n";
 for (int i = 1; i <= n; i++)
 if (primo(i))
 std::cout << i << ' ';
}
// Devolve true se o numero n > 0 e' primo e false caso contrario.
// Aborta se n <= 0
bool primo(int n) {
 assert(n > 0);
 if(n == 1) return false;
 int m = (int) sqrt(float (n));
 for (int i = 2; i <= m; i++)
 if (n % i == 0)
 return false;
 return true;
}
// Devolve o factorial de n se n>=0. Aborta se n < 0
int factorial(int n) {
 assert(n >= 0);
 int f = 1;
 for(int i = 1; i <= n; i++)
 f *= i;
 return (f);
}
```

# Argumentos na linha de comando

## Exemplo 2 (executando)



```
./principal<enter>
E queceu-se do parametro no intervalo [1,9]
```

# Argumentos na linha de comando

## Exemplo 2 (executando)

- ▶ 

```
./principal<enter>
E queceu-se do parametro no intervalo [1,9]
```
  
- ▶ 

```
./principal 123 fafa<enter>
Excesso de parametros
O parametro deve ser apenas um unico digito no intervalo [1,9]
```

# Argumentos na linha de comando

## Exemplo 2 (executando)

- ▶ 

```
./principal<enter>
E queceu-se do parametro no intervalo [1,9]
```
- ▶ 

```
./principal 123 fafa<enter>
Excesso de parametros
O parametro deve ser apenas um unico digito no intervalo [1,9]
```
- ▶ 

```
./principal 123<enter>
O parametro deve ser apenas um digito no intervalo [1,9]
```

# Argumentos na linha de comando

## Exemplo 2 (executando)

- ▶ 

```
./principal<enter>
E queceu-se do parametro no intervalo [1,9]
```
- ▶ 

```
./principal 123 fafa<enter>
Excesso de parametros
O parametro deve ser apenas um unico digito no intervalo [1,9]
```
- ▶ 

```
./principal 123<enter>
O parametro deve ser apenas um digito no intervalo [1,9]
```
- ▶ 

```
./principal 9<enter>

O numero 9 nao e' primo e o seu factorial vale 362880.
Os primos menores ou iguais a 9:
2 3 5 7
```