Linguagem de Programação C

Verónica Vasconcelos & Luís Marques

Departamento de Engenharia Electrotécnica
Instituto Superior de engenharia de Coimbra

1. Introdução

De modo a estudar a estrutura de um programa em Linguagem C vamos começar por analisar um pequeno exemplo:

Exemplo 1.1: Um programa simples em linguagem **C.**

```
#include <stdio.h>
int main(void)
{
  printf("O meu primeiro programa em C\n\n");
}
```

Se executarmos este programa obtemos no monitor (**_** é o cursor a piscar)

```
O meu primeiro programa em C
_
```

Algumas características de um programa em linguagem **C:**

1. um programa em C tem de possuir uma função **main**. A execução do programa começa por esta função
2. esta linguagem é *case sensitive*. Assim **main** é diferente de **Main**
3. todas as instruções são terminadas por **;**
4. os blocos são definidos pelos caracteres **{** e **}**
5. os comentários são colocado entre **/*** e ***/** , podendo ocupar várias linhas
6. na primeira linha de código está: **#include <stdio.h>** onde **#include** é uma directiva de compilação que indica ao compilador para incluir o ficheiro **stdio.h** quando da compilação do programa (por exemplo, é neste ficheiro que está definida a função **printf()**).

1.1 Variáveis, Constantes e Tipos de Dados Pré-Definidos

1.1.1 Variáveis

Uma variável consiste num identificador que é atribuído a uma zona de memória, que irá conter um valor que pode mudar ao longo do programa. A declaração de variáveis tem de ser efectuada antes de qualquer instrução. Quando uma variável é declarada está a ser solicitado ao compilador espaço de memória, correspondente ao tipo de dados da variável. A sintaxe consiste em:

```
identif_tipo identif_var;

ou
identif_tipo identif_var1, identif_var2, ... , identif_varn;
```

1.1.2 Constantes

Constantes são identificadores que correspondem a valores fixos, que não podem ser modificados ao longo da execução do programa. Uma constante pode ser declarada através da palavra reservada **const** ou através da directiva de pré-processamento #**define**.

Constante formal

```
const identif_tipo identif_const = valor;
```

Utilizando a directiva **#define**

```
#define identif_const = valor_ou_texto
```

No caso da constante ser definida através da palavra **const**, é reservado espaço de memória que lhe fica associado e a constante tem um tipo de dados. No caso da utilização da directiva #**define** o pré-processador vai substituir todas as ocorrências correspondentes à constante pelo respectivo valor, antes de ocorrer a compilação do programa.

O recurso a constantes é aconselhável uma vez que torna o programa mais legível e simplifica a actualização do valor que foi atribuído à constante. Imagine que pretende alterar o valor de uma constante, se procedeu à sua declaração apenas precisa de alterar o seu código uma vez, se não, terá pesquisar o seu programa e actualizar todas as ocorrências do valor constante.

Por convenção, o identificador atribuído a uma constante é em letras maiúsculas. É aconselhável.

1.1.3 Tipos de Dados Pré-Definidos

Os tipos de dados pré-definidos existentes em C são:

int	Inteiro
char	Carácter
float, double	Reais (em vírgula flutuante)
void	Nulo

1.1.3.1 O Tipo Inteiro

Para representar números inteiros com e sem sinal dispomos do tipo **int**. A este tipo é possível aplicar os modificadores:

- signed (por omissão)
- unsigned
- short
- long

A aplicação de um, ou mais, modificadores permite alterar a gama de variação do inteiro. Para um sistema de 32 bits temos:

	nº bits ocupados	**Gama de variação**
int	32	-2 147 483 648 .. 2 147 483 647
short int	16	-32768 .. 32767
unsigned short int	16	0 .. 65535
unsigned int	32	0 .. 4 294 967 295
long	32	-2 147 483 648 .. 2 147 483 647
unsigned long	32	0 .. 4 294 967 295

Exemplos de declarações de constantes e variáveis inteiras:

```
const unsigned long o_maior = 4294967295;
const int mais_negativo = -2147483648;
const short int pequeno = 1024;
int x1, x2;
unsigned int positivo;
long grande, maior, x, X;
unsigned short int pequeno_positivo;
```

Em **C** quando do uso de modificadores de tipo podemos eliminar a palavra int, como podemos ver na tabela:

Tipo	Pode ser abreviado por
short int	short
unsigned short int	unsigned short
unsigned int	unsigned
long int	long
unsigned long int	unsigned long

Operadores disponíveis para inteiros

Operador	C
Adição	+
Subtracção	-
Multiplicação	*
Divisão inteira	/
Resto divisão inteira	%

Clarificação acerca da divisão inteira e do resto da divisão inteira:

```
13 | 4
 1 | 3
```

Assim 13/4 tem como resultado 3

e 13%4 tem como resultado 1

Funções pré-definidas para inteiros

Matemática	C
| x |	abs(x)

A função **abs()** está definida na biblioteca **stdlib.h**, pelo que é necessário colocar a linha seguinte, para a poder utilizar:

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

1.1.3.2 O Tipo Real

Permite armazenar números reais no formato de vírgula flutuante. Em C temos:

	nº bits ocupados	Gama de variação
float	32	$-3.4x10^{38}$ .. $-3.4x10^{-38}$, 0, $3.4x10^{-38}$ .. $3.4x10^{38}$
double	64	$-1.7x10^{308}$ .. $-1.7x10^{-308}$, 0, $1.7x10^{-308}$ .. $1.7x10^{308}$
long double	80	$-1.1x10^{4932}$ .. $-3.4x10^{-4932}$, 0, $3.4x10^{-4932}$ .. $1.1x10^{4932}$

Notações Para Representar Números Reais

Decimal - nesta notação o número é representado por uma parte inteira e fraccionária separadas por um ponto decimal. A parte inteira pode ser precedida pelo sinal do número. O ponto decimal tem de ser antecedido e seguido de, pelo menos, um digito.

Exemplos:

0.035 -2532.3 -0.4

Científica - esta possui a seguinte definição formal:

<mantissa>E<expoente> (= mantissax$10^{expoente}$; o E também pode ser e)

Formato corrente	C (notação científica)
$524.4x10^{3}$	524.4e3
33.01×10^{-2}	33.01E-2
-12×10^{6}	-12E6

Exemplos de declarações de constantes e variáveis:

```
const float pi = 3.1415;
const double pi_precisao = 3.141592654;
const long double nepper = 2.718281828;
const double velocidade_luz = 2.99792458E8;
float x1, x2, x3, x, X ;
long double mto_grande, maior;
```

Operadores disponíveis para reais

Operador	C
Adição	+
Subtracção	-
Multiplicação	*
Divisão	/

Funções pré-definidas para reais

Estas funções encontram-se definidas na biblioteca **math.h**

Matemática	C	
| x |	fabs(x)	
$\sqrt{x}$	sqrt(x)	
e^x	exp(x)	
ln (x)	log(x)	
$\log_{10}(x)$	log10(x)	
sin(x)	sin(x)	
cos(x)	cos(x)	
tg(x)	tan(x)	
arctg(x)	atan(x)	atan(x) e atan(x) devolvem o resultado em radianos, de $-\pi$ e π
	atan2(x,y)	
x^y	pow(x,y)	

1.1.3.3 O Tipo Carácter - *Char*

Se um computador só armazena números, como é que ele consegue armazenar texto (conjunto de caracteres)? - Usando o código **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), que estabelece uma correspondência unívoca entre caracteres e números (código alfanumérico).

Código ASCII *Standard*

- Código alfanumérico (+ caracteres de controlo);
- Utiliza combinações de 7 bits pelo que pode representar 128 símbolos (caracteres) distintos.

Código ASCII Estendido

- Código alfanumérico (+ caracteres de controlo);
- Utiliza combinações de 8 bits pelo que pode representar 256 símbolos (caracteres) distintos;
- Os primeiros 128 caracteres são os mesmos do código ASCII *standard*; os outros 128 caracteres correspondem a caracteres especiais (p. ex. 'ç', 'Ç', vogais acentuadas, '≥', '└', '↑', '©', ...);
- os últimos 128 caracteres dependem da língua (*code page* no MS-DOS);

Nas primeiras 32 posições da tabela encontram-se os caracteres de controlo. Ver tabela ASCII no Apêndice TAB_ASCII.

O tipo **char** pode ser **signed** (por omissão) ou **unsigned**. Uma variável ou constante do tipo **char** permite o armazenamento de apenas um carácter.

	nº bits	**Gama de variação**
char	8	Todos os caracteres da tabela ASCII estendida

Representação de valores do tipo carácter

- um valor do tipo carácter é representado pelo carácter entre plicas. Por exemplo: 'a', 'Z', '?', '*'
- em C um carácter também pode ser representado pela sua sequência de escape, sendo este modo de definir o carácter particularmente útil na definição de caracteres de controlo (os 32 primeiros da tabela ASCII).

Exemplos com sequências de escape

Carácter	Posição na tabela	em octal	em hexadecimal
LF	10	’\12’	’\xA’
CR	13	’\15’	’\xD’
ESC	27	’\33’	’\x1B’
1	49	’\61’	’\x31’
A	65	’\101’	’\x41’

A linguagem **C** define ainda os seguintes **caracteres especiais**:

Carácter	C	Carácter	C
alert (beep)	’\a’	carriage return	’\r’
newline	’\n’	vertical tab	’\v’
tab	’\t’	backspace	’\b’
form feed	’\f’	\	’\\’
’	’\’	”	’\’

Na realidade um carácter é armazenado como um inteiro de 8 bits, ou seja, o que é armazenado é a sua posição na tabela ASCII.

Exemplos de declarações de constantes e variáveis do tipo carácter:

```
const char ULTIMA = ’Z’;
const char ENTER = ’\n’;
const char ESC = ’\x1B’;
char letra, inicial;
```

1.2 Operadores Relacionais

É possível comparar expressões utilizando os operadores relacionais, sendo o resultado **0** ou **1**, consoante o resultado da comparação for, respectivamente, **falso** ou **verdadeiro**.

Operador	C
maior que	>
maior ou igual a	>=
menor que	<
menor ou igual a	<=
igual a	==
diferente de	!=

Exemplos:

Expressão	Resultado
’b’ > ’a’	1
123.1 > 10.011	1
101E10 <= 101	0
’b’ == ’a’	0
’a’ != ’A’	1
54.5E2 == 54.5e2	1

1.3 Operadores Lógicos

É possível construir expressões booleanas utilizando os operadores lógicos. O valor **falso** é representado pelo inteiro **0** e **verdadeiro** por um valor inteiro **diferente de zero**.

Operador	C
E	&&
OU	||
NEGAÇÃO	!

Tabela de verdade dos operadores lógicos:

a	b	a && b	a || b	!a
0	0	0	0	1
0	não-zero	0	1	1
não-zero	0	0	1	0
não-zero	não-zero	1	1	0

Nota: não-zero significa um valor diferente de zero, seja ele positivo ou negativo.

1.4 Operadores de Manipulação de Bits (Bitwise Operators)

Estes operadores realizam operações lógicas bit a bit

Operador	C
E bitwise	&
OU bitwise	|
EXOR bitwise	^
NEGAÇÃO bitwise	~
passagem à esquerda	<<
passagem à direita	>>

Exemplo: Suponha que nas variáveis **a** e **b** (1 *byte*) estavam armazenados os valores seguintes, e que se realizavam as operações indicadas:

Variáveis e operação	Valor e resultado
a	1 1 0 0 0 1 1 1
b	1 0 0 1 1 1 1 1
a & b	1 0 0 0 0 1 1 1
a | b	1 1 0 1 1 1 1 1
a ^ b	0 1 0 1 1 0 0 0
~a	0 0 1 1 1 0 0 0
a << 2	0 0 0 1 1 1 0 0
a >> 2	0 0 1 1 0 0 0 1

1.5 Operador de Atribuição

Em **C** o operador de atribuição é o carácter **=** , e deve ler-se **"toma o valor de"**. O operador **=** faz com que à variável que aparece à sua esquerda seja atribuído o resultado da expressão que se encontra à direita. Este é o operador com mais baixa prioridade de todos os operadores da linguagem **C**. O programa seguinte contém instruções de atribuição e encontra-se comentado de modo a tentar explicar este operador.

Exemplo 1.2: Programa que ilustra diversas formas de utilizar o operador atribuição.

```
#include <stdio.h>
void main(void)
{
  const int a = 22;
  int x;

  x = 1; /* x toma o valor 1 */
  x = a; /* x toma o valor da constante a, ou seja 22 */
  x = 3 * a - 99; /* depois de calculado o resultado da expressão,   ou seja 3*22-99, cujo
resultado é 33, é atribuído à variável  x este valor */
  x = a - 20; /* x toma o valor 2 (de 22-20) */
  x = 10; /* x toma o valor 10 */
  x = x + 1; /* x toma o valor de 10+1, ou seja, 11 */
}
```

Em certos casos é possível escrever as instruções de atribuição de um modo abreviado. Suponhamos a linha seguinte:

```
valor_introduzid_utilizador = valor_introduzid_utilizador + 10;
```

podemos substitui-la por uma forma abreviada:

```
valor_introduzid_utilizador += 10;
```

Muitas operações de atribuição permitem a sua escrita num formato abreviado:

Formato normal	Formato normal	Formato abreviado	Formato abreviado
x = x + y	x += y	x = x | y	x |= y
x = x - y	x -= y	x = x & y	x &= y
x = x * y	x *= y	x = x ^ y	x ^= y
x = x / y	x /= y	x = x << y	x <<= y
x = x % y	x %= y	x = x >> y	x >>=y

1.6 Operadores de Incrementação e de Decrementação

Os operadores de incrementação e decrementação, respectivamente **++** e **--** permitem incrementar/decrementar uma variável de uma unidade. São operadores concisos e muito utilizados em **C**. Colocando o operador depois da variável, esta só vê o seu valor alterado depois de ter sido utilizado na expressão. No caso do operador aparecer antes da variável então é realizada em primeiro lugar a incrementação/decrementação e só depois é que o valor corrente da variável é utilizado na expressão.

Sintaxe:

ident_var++	pós-incrementação
++ident_var	pré-incrementação
ident_var--	pós-decrementação
--ident_var	pré-decrementação

Exemplos de algumas instruções que utilizam os operadores ++ e --

```
int i = 10, j;
j = i++; /* j é agora 10 e i é 11 */
j = ++i; /* j é agora 12 e i é 12 */
i = 1;
j = 10 * i++; /* j é agora 10 e i é 2 */
i = 1;
j = 10 * ++i; /* j é agora 20 e i é 2 */
```

1.7 Operador sizeof

O operador **sizeof** permite obter a dimensão, ou seja o número de *bytes* que ocupa, um tipo de dados ou de uma variável.

Exemplos de instruções utilizando o operador **sizeof**

```
double k;
int tam1, tam2, tam3;
char letra;
tam1 = sizeof(int);   /* tam1 toma o valor 4 */
tam2 = sizeof(k);     /* tam2 toma o valor 8 */
tam3 = sizeof(letra); /* tam3 toma o valor 1 */
```

1.8 Operador de Conversão de Tipos (*cast*)

O **C** aceita os seguintes modos de conversão de tipos

cast (expressão) ou (cast)expressão

Exemplos de aplicação do operador conversor de tipos:

```
int x = 2;
float y, z;
y = (float)x;  /* o valor de x é convertido num float e de seguida atribuído a y */
z = float (x); /* e a Z */
```

1.9 Expressões

Uma expressão é um conjunto de operandos (valores, constantes, valor corrente de variáveis e ainda o resultado de funções) agrupados por certos operadores constituindo formas algébricas que representam um valor. As regras usadas na avaliação da expressão são as mesmas da matemática:

- se os operadores tiverem a mesma prioridade, a expressão é avaliada da esquerda para a direita;
- se os operadores tiverem prioridades distintas, são efectuados em primeiro lugar as operações que contêm os operadores de maior prioridade;
- a ordem de avaliação da expressão pode ser alterada com a utilização de parênteses.

É então importante conhecer a prioridade dos operadores na linguagem **C**, que se apresenta a seguir, de uma forma abreviada. No Apêndice "Tabela de Precedência dos Operadores em C" pode consultar a lista completa dos operadores em **C** e prioridades respectivas.

Operador	Prioridade
* / %	MAIOR
+ -	
< <= > >=	
== !=	
&& ||	
=	MENOR

1.10 Identificadores - Regras e Convenções

Os identificadores ou nomes que são atribuídos num programa em C devem obedecer às seguintes regras:

- Devem começar por uma letra ou o carácter underscore (_);
- Os caracteres seguintes podem ser letras, underscore ou dígitos;
- Não podem ser escolhidos identificadores que sejam palavras reservadas da linguagem.
- Os identificadores devem ter nomes significativos, de modo a facilitar a compreensão dos programas e reduzir comentários supérfluos.

Quando se escolhe um identificador existem alguns cuidados e convenções que devem ser tidos em atenção:

- Não é aconselhável a utilização de caracteres acentuados em identificadores, pois alguns compiladores não os aceitam;
- Identificadores totalmente em letras maiúsculas estão geralmente associados a constantes, esta convenção deve ser respeitada;
- Quando se pretende um identificador com várias palavras, estas devem ser separadas por *underscore* ou iniciar as palavras por maiúsculas ou minúsculas.

2. Interacção com o Utilizador

A interacção básica com o utilizador implica a escrita no monitor e a leitura a partir do teclado, sendo isto conseguido utilizando as funções **printf()** e **scanf()**, respectivamente. Existem outras funções para interacção com o utilizador, sobre as quais nos debruçaremos mais tarde.

2.1 Escrita no Monitor

A função **printf()** permite escrever "valores" no monitor. Esta encontra-se definida no ficheiro **stdio.h**, pelo que, se pretendermos utilizar esta função é necessário colocar a linha #include <stdio.h>

Sintaxe:

printf(string_de_formatação,expr_1, expr_2, ..., expr_n);

A ***string* de formatação** é composta por um conjunto de caracteres mais especificadores de formato (*conversion specifier*) colocados entre aspas, existindo um especificador para cada expressão da lista.

Exemplo 2.1: A função printf na sua forma mais simples de utilização, sem especificadores de formato.

```
#include <stdio.h>

void main(void)
{
  printf("1 - Certo ou errado.");
  printf("2 - Certo ou errado.\n");
  printf("3 - Certo\n ou\n errado.");
}
```

e o resultado no monitor é (onde _ representa o cursor):

```
1 - Certo ou errado.2 - Certo ou errado.
3 - Certo
 ou
 errado. _
```

Se pretendermos escrever no monitor um valor então teremos de utilizar o especificador de formato. O valor, resultado da expressão, tem de ser convertido num conjunto de caracteres antes de poder ser escrito no monitor, pelo que tem de existir um modo de informar a função **printf()** do formato pretendido. Assim utiliza-se o especificador de formato, sendo este composto por, pelo menos, um carácter precedido de **%**. A cada expressão na lista corresponde obrigatoriamente um especificador de formato.

Especificador de formato (forma genérica)

% [flags][width][.prec][h|l|L] **type_char**

type_char - carácter de conversão de tipo

d	inteiro c/ sinal
i	inteiro c/ sinal
x	inteiro s/ sinal em hexadecimal (com a .. f)
X	inteiro s/ sinal em hexadecimal (com A .. F)
o	inteiro s/ sinal em octal
f	real no formato parte inteira, ponto e parte fraccionária
e	real no formato exponencial, com carácter e
E	real no formato exponencial, com carácter E
g	formato mais compacto que o menor de f ou e; só coloca os dígitos significativos
G	formato mais compacto que o menor de f ou E ; só coloca os dígitos significativos
c	carácter
s	*string* (cadeia de caracteres)
%%	carácter % (caso especial)

width - reserva um determinado número de colunas

n	reserva **n** colunas e o valor é escrito com justificação à direita
0n	reserva **n** colunas (com justificação à direita) e acrescenta zeros à esquerda

.prec - define o número de colunas da parte fraccionária (números reais –float e double)

(nada)	para f, e, E coloca 6 casas decimais; apenas os dígitos significativos em g, G
.n	**n** casas decimais
.0	Não coloca casas decimais (nem o ponto)

flags

-	altera justificação para justificado à esquerda
+	valores positivos são precedidos pelo carácter +
(espaço)	valores positivos são precedidos por um espaço

[h|l|L]

h	short int ou unsigned short int
l	long int ou double
L	long double

Exemplo 2.2: Programa que ilustra a utilização de especificadores de formato.

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

void main(void)
{
  int i = 5;
  long int j = 1024;
  float x = 10.0;
  double y = 758.625;
  long double z = 1048576.025;
  char let = 'A';

  printf("%d\n", i);
  printf("Um inteiro=%d e um inteiro longo =%d\n", i, j);
  printf("Caracter %c na posicao %d\n\n", let, let);

  printf("------------------------------------------------\n");
  printf("Num real em varios formatos %f %e %E\n", x, x, x);
  printf("Outro real %f \n", y);
  printf("Real %g  %G\n", y, y);
  printf("Um inteiro e um real %d  %f\n\n", -32, fabs(-32.0/3));

  printf("------------------------------------------------\n");
  printf("Numero=%10d\n", i);
  printf("Numero=%010d\n", i);
  printf("%10c%12f\n", 'X', y);
  printf("%10c%12.1f\n", 'y', y);
  printf("%12.0f\n%012.0f\n", y, y);
  printf("%12.1e\n\n", y);

  printf("------------------------------------------------\n");
  printf("%Lf\n", z);
  printf("%12.1Lf\n", z);
}
```

O resultado no monitor é:

```
5
Um inteiro=5 e um inteiro longo =1024
Caracter A na posicao 65

-----------------------------------------------
Num real em varios formatos 10.000000 1.000000e+01 1.000000E+01
Outro real 758.625000
Real 758.625  758.625
Um inteiro e um real -32  10.666667

-----------------------------------------------
Numero=         5
Numero=0000000005
        X  758.625000
        y       758.6
        759
000000000759
    7.6e+02

-----------------------------------------------
1048576.025000
  1048576.0
_
```

2.2 Leitura de Valores

A função **scanf()** permite ler valores a partir do teclado, encontrando-se definida no ficheiro **stdio.h**.

Sintaxe:

scanf(string_de_formatação, lista_de_endereços_das_variáveis);

A ***string* de formatação** é composta apenas por especificadores de formato (colocados entre aspas). O endereço de uma variável é indicado colocando o carácter **&** antes do identificador. Assim **&let** é o endereço da variável **let**. É o especificador de formato que informa a função **scanf()** de como interpretar os valores lidos.

Especificador de formato (forma genérica)

% [*][width][h|l|L] **type_char**

width

n	**n** é o número máximo de caracteres a serem lidos relativamente a uma determinada variável da lista

type_char - carácter de conversão de tipo

d	inteiro
i	inteiro
o	inteiro em octal
x	inteiro em hexadecimal
f, e, g	real em vírgula flutuante
c	carácter
s	*string* (cadeia de caracteres)

***** - supressão de leitura, o valor é lido mas não é colocado no endereço especificado

[h|l|L]

h	short int ou unsigned short int
l	long int ou double
L	long double

Exemplo 2.3: Programa que mostra a utilização correcta da função scanf().

```
#include <stdio.h>
void main(void){
  int i;
  float x;
  double y;
  long double z;
  char letra;

  printf("Introduza uma letra");
  scanf("%c", &letra);
  printf("Introduza um inteiro");
  scanf("%d", &i);
  printf("Introduza 3 numeros reais");
  scanf("%f%lf%Lf", &x, &y, &z);
  printf("Inteiro =%d\n", i);
  printf("Reais = %f\t%f\t%Lf\n", x, y, z);
  printf("Letra %c na pos. %d na tabela ASCII\n", letra, letra);
}
```

Vamos testar um programa com a função **printf()** e que permite verificar os conhecimentos adquiridos sobre especificadores de formato. Utilize uma folha de papel quadriculado e escreva nela o resultado do programa que se segue. De seguida escreva o programa do exemplo seguinte no compilador e execute-o. Compare os resultados.

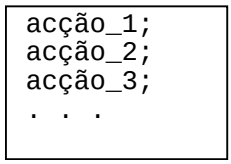
Exemplo 2.3: Programa que ilustra os vários especificadores de formato do **C**.

```
#include <stdio.h>
void main()
{
  float z = 1234.625;
  double zz = 12345678.625;
  int i = 1024;

  printf("%d\n", i);
  printf("%+d\n", i);
  printf("%10d\n", i);
  printf("%+10d\n", i);
  printf("% 10d\n", i);
  printf("%010d\n", i);
  printf("%+010d\n", i);
  printf("% 010d\n", i);

  printf("%f\n", z);
  printf("%e\n", z);
  printf("%E\n", z);
  printf("%g\n", z);
  printf("%G\n", z);
  printf("%20e\n", z);
  printf("%20.1f\n", z);
  printf("%20.1e\n", z);
  printf("%+020.1e\n", z);
  printf("%+020.1f\n", z);

  printf("%Lf\n", zz);
  printf("%Le\n", zz);
  printf("%Lg\n", zz);
  printf("%20Lf\n", zz);
  printf("%20.1Le\n", zz);

  printf("%20.2f\n", z);
  printf("%.2f\n", z);
}
```

3. Estruturas de Controlo

Em qualquer linguagem de programação de alto nível existem três estruturas fundamentais para o controlo de fluxo de um programa. São elas a sequência, a selecção e a repetição, que se podem esquematizar da seguinte forma:

SEQUÊNCIA

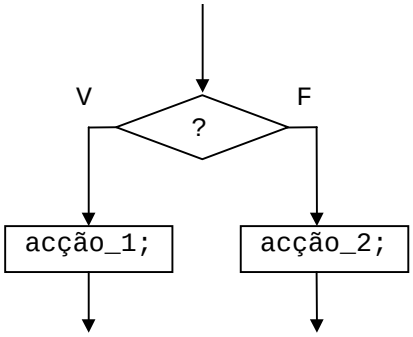


SELECÇÃO ou ESCOLHA

if

if-else

switch

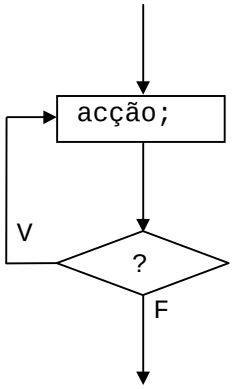


REPETIÇÃO ou CICLO

while

do-while

for



3.1 Sequência

Composto por um conjunto de instruções, umas após as outras, e provoca a execução dessas instruções pela ordem em que se encontram. As instruções são convenientemente separadas por **;**. Vamos ver alguns exemplos:

Exemplo 3.1: Programa para calcular a idade, sendo introduzido o ano de nascimento.

```
/* Calcula a idade sabendo o ano de nascimento */
#include <stdio.h>
#include <math.h>
void  main(void)
{
  int ano_nasc, idade;

  printf("Qual o seu ano de nascimento ?\n");
  scanf("%d", &ano_nasc);

  idade = 2001-ano_nasc;

  printf("Você tem %d anos\n", idade);
}
```

Exemplo 3.2: Programa para calcular a soma de três valores inteiros.

```
/* Calcula a soma de três valores inteiros */
#include <stdio.h>
void  main(void)
{
  int x1, x2, x3, y;

  printf("Indique três números inteiros ?\n");
  scanf("%d%d%d", &x1, &x2, &x3);

  y = x1 + x2 + x3;

  printf("Soma dos três valores , igual a %d\n", y);
}
```

Exemplo 3.3: Se rescrevermos o programa do seguinte modo a resolução estará correcta ?

```
/* Calcula a soma de três valores inteiros */
#include <stdio.h>
void main(void)
{
  int x1, x2, x3;

  printf("Indique três números inteiros ?\n");
  scanf("%d%d%d", &x1, &x2, &x3);

  x1 = x1 + x2 + x3;

  printf("Soma dos três valores , igual a %d\n", x1);
}
```

Exemplo 3.4: Podemos dispensar a variável y se a operação de soma for colocada no printf().

```
/* Calcula a soma de tres valores inteiros */
#include <stdio.h>
void main(void)
{
  int x1, x2, x3;

  printf("Indique três números inteiros ?\n");
  scanf("%d", &x1);
  scanf("%d", &x2);
  scanf("%d", &x3);
  printf("Soma dos três valores , igual a %d\n", x1 + x2 + x3);
}
```

Exemplo 3.5: Programa para fazer a conversão entre Escudos e Euros.

```
/* Faz conversão Escudos -> Euros */
#include <stdio.h>
void main(void)
{
  const float euro_esc = 200.482;
  int escud;
  /* valor original em escudos é um valor inteiro*/
  float  euro;

  printf("Quantos escudos possui ?\n");
  scanf("%d", &escud);

  euro = escud/euro_esc;
  printf("%d escudos correspondem a %.2f Euro\n", escud, euro);
}
```

Exemplo 3.6: Programa para fazer a conversão entre Euros e Escudos.

```
/* Faz conversão Euros -> Escudos */
#include <stdio.h>

void main(void)
{
  const float euro_esc = 200.482;
  float escud, euro;

  printf("Quantos Euros possui ?\n");
  scanf("%f", &euro);

  escud = euro*euro_esc;

  printf("%.2f Euross correspondem a %.2f Escudos\n", euro, escud);
}
```

Exemplo 3.7: Programa para calcular a frequência, conhecido o período do sinal

```
/* Calcula a frequência, conhecido o período do sinal . Relação: f=1/T */
#include <stdio.h>
void main(void)
{
  float freq, T;

  printf("Qual o período do sinal (em s) ?\n");
  scanf("%f", &T);

  freq = 1/T;

  printf("O sinal com período %f s tem frequência %f Hz", T, freq);
  /* observe a diferença entre %f e %e */
  printf("O sinal com período %f s tem frequência %e Hz", T, freq);
}
```

Exemplo 3.8: Programa para calcular a área de um circulo. Relação: $A=\pi r^2$

```
/* Calcula a área de um circulo */
#include <stdio.h>
#include <math.h>
void main(void)
{
  float raio, area;

  printf("Qual o raio do circulo ?\n");
  scanf("%f", &raio);
  area = PI*raio*raio; /* valor de PI encontra-se definido em math.h */
  printf("O circulo tem area = %f\n", area);
}
```

Exemplo 3.9: Programa para calcular a impedância de uma bobine, se forem conhecidas a indutância e a frequência. Relação: $|Z|=2\omega fL$

```
/* Calcula a impedancia de uma bobine */
#include <stdio.h>
#include <math.h>
void main(void)
{
  float freq, L, Z;

  printf("Qual o indutância da bobine (em H) e a frequência ?\n");
  scanf("%f%f", &L, &freq);

  Z = 2*PI*freq*L;
  printf("A bobine de indutância %f Henry, … frequência de %f Hz apresenta uma
impedância de %f Ohm\n", L, freq, Z);
}
```

Exemplo 3.10: Programa para calcular o volume de uma esfera. Relação: $V=4/3\pi r^3$

```
/* Calcula o volume de uma esfera */

#include <stdio.h>
#include <math.h>

void main(void)
{
  float raio, vol;

  printf("Qual o raio da esfera ?\n");
  scanf("%f", &raio);
  vol = 4.0/3*PI*raio*raio*raio;

  /* Vários printf para se observar escrita formatada */
  printf("A esfera de raio %f tem volume = %f\n", raio, vol);
  printf("A esfera de raio %12f tem volume = %f\n", raio, vol);
  printf("A esfera de raio %.2f tem volume = %f\n", raio, vol);
  printf("A esfera de raio %12.2f tem volume = %f\n", raio, vol);
  printf("A esfera de raio %e tem volume = %f\n", raio, vol);
  printf("A esfera de raio %12e tem volume = %f\n", raio, vol);
  printf("A esfera de raio %.2e tem volume = %f\n", raio, vol);
  printf("A esfera de raio %12.2e tem volume = %f\n", raio, vol);
}
```

Exemplo 3.11: Programa para calcular a impedância de um condensador se forem conhecidas a capacidade e a frequência. Relação: $|Z|=1/(2\omega fC)$.

```
/* Calcula a impedância de um condensador */

#include <stdio.h>
#include <math.h>
void main(void)
{
  float freq, C, Z;

  printf("Qual o capacidade do condensador (em F) e a frequência ?\n");
  scanf("%f%f", &C, &freq);

  Z = 1/(2*PI*freq*C);

  printf("O condensador com %f Farad, à frequência de %f Hz apresenta uma impedância
de %f Ohm\n", C, freq, Z);
}
```

Exercício proposto: Altere este programa de modo aos valores reais aparecerem só com duas casas na parte fraccionária. Experimente ainda substituir todos os %f por %e. Qual é a diferença?

3.2 Selecção ou Escolha

Permite alterar a sequência de execução de um programa. Assim, é executado um conjunto ou outro de instruções consoante o resultado de uma expressão de teste (ou condição de teste).

3.2.1 A instrução if de Alternativa Simples

Esta instrução permite que uma ou mais instruções sejam executadas no caso da expressão (ou condição) de teste seja verdadeira. Se o resultado da expressão for falso então passa-se à instrução seguinte.

Sintaxe:

```
if (condição)
    instrução;
```

ou

```
if (condição) {
    instrução_1;
    instrução_2;
    …
    instrução_n;
}
```

Exemplo 3.12: Programa para calcular o seno de um ângulo pertencente ao primeiro quadrante. Se o ângulo não pertencer ao quadrante referido o programa não deve fazer nada.

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

/* necessário para se poder utilizar a função sin */

void main(void)
{
  const float pi = 3.14159;
  float ang, s;

  printf("Introduza um angulo do primeiro quadr. em graus -> ");
  scanf("%f", &ang);
  if (ang >= 0.0 && ang <= 90.0){
    s = sin(pi / 180 * ang);
    /* o argumento da função sin deve ser em radianos */
    printf("Seno de %5.2fº igual a %10.8f\n", ang, s);
  }
}
```

Se pretendermos dar uma indicação no caso de o ângulo não pertencer ao primeiro quadrante, podemos acrescentar outra instrução **if** ao programa anterior. Vejamos o exemplo seguinte.

Exemplo 3.13: Exemplo 3.12 com informação de que o ângulo não se encontra no primeiro quadrante.

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

void main(void)
{
 const float pi = 3.14159;
 float ang, s;

 printf("Introduza um angulo do primeiro quad. em graus -> ");
 scanf("%f", &ang);
 if (ang >= 0.0 && ang <= 90.0){
  s = sin(pi / 180 * ang);
  printf("Seno de %5.2fº igual a %10.8f\n", ang, s);
 }
 if (ang < 0.0 || ang > 90.0)
  printf("Erro! Angulo fora do primeiro quadrante.\n");
}
```

3.2.2 A Instrução if-else de Alternativa Dupla

Na instrução **if-else** o resultado da condição testada determina qual a sequência de instruções a executar. Se o resultado da condição for verdade então é executada a sequência colocada a seguir ao **if** (instrução_1 ou instrução_1_1,…, instrução_1_n), senão é executada a sequência colocada a seguir ao **else** (instrução_2 ou instrução_2_1,…, instrução_2_m).

Sintaxe:

```
if (condição)                ou          if (condição) {
    instrução_1;                             instrução_1_1;
else                                         …
    instrução_2;                             instrução_1_n;
                                         }
                                         else {
                                             instrução_2_1;
                                             …
                                             instrução_1_m;
                                         }
```

O programa anterior pode então ser rescrito de modo a utilizar a instrução **if-else**, o que nos permite poupar o teste de uma condição.

Exemplo 3.14: Exemplo 3.13 rescrito utilizando a instrução if-else.

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

void main(void)
{
 float ang, s;

 printf("Introduza um angulo do primeiro quad. em graus -> ");
 scanf("%f", &ang);
 if (ang >= 0.0 && ang <= 90.0){
   s = sin(PI / 180 * ang);
   printf("Seno de %5.2fº igual a %10.8f\n", ang, s);
 }
 else
   printf("Erro! Angulo fora do primeiro quadrante.\n");
}
```

A instrução_1 ou a instrução_2 que aparecem a seguir à condição de teste do **if** , ou a seguir ao **else**, podem ser instruções quaisquer em **C**, logo também podem ser uma instrução **if-else**. Neste caso dizemos que estamos em presença de instruções **if-else** encadeadas. A importância de um correcto encadeamento da instrução **if-else** é explicitado nos dois exemplos seguintes.

Exemplo 3.15: Suponha que pretende escrever um programa que escreva a mensagem "Voce acertou no numero" se o número fosse positivo e igual a 100 e a mensagem "Número fora da gama !" se o número fosse negativo. Vejamos uma primeira solução:

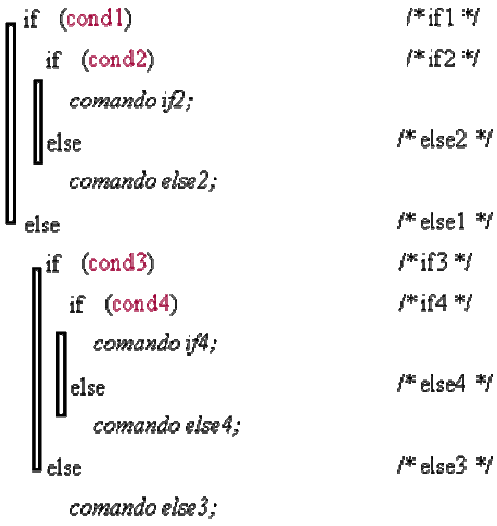
```
#include <stdio.h>
void main(void)
{
 int num;
 printf("Introduza um número");
 scanf("%d", &num);
 if (num > 0)
   if (num == 100) /* operador igual é == não confundir com = */
     printf("Voce acertou no numero\n");
 else
   printf ("Numero fora da gama !\n");
}
```

Escreva este pequeno programa e execute-o várias vezes, respondendo com os valores -5, 1 e 100. **O que aconteceu ?** Com certeza reparou que para o valor 1 também apareceu a mensagem Número fora da gama !. **Porquê ?** O que acontece é que o **else** não está associado ao **if** que nós pretendíamos, pois o compilador vai interpretar o **else** como pertencente ao **if** interior. Isto deve-se ao facto do compilador associar cada **else** com o **if** mais próximo, a contar de fora para dentro. De modo a clarificar a situação temos de utilizar os delimitadores de blocos, que em **C** são as chavetas.

Exemplo 3.16:. O exemplo 3.15 agora correctamente escrito com if-else encadeado.

```
#include <stdio.h>

void main(void)
{
  int num;
  printf("Introduza um número");
  scanf("%d", &num);
  /* agora só quando o número é negativo é que se obtém a mensagem  de erro */
  if (i > 0) {
    if (i == 100)
       printf("Voce acertou no numero\n");
  }
  else
    printf ("Numero fora da gama pretendida !");
}
```

Os Exemplos 3.15 e 3.16 exemplificam que é necessário ter bastante atenção à utilização de instruções **if-else encadeadas** para se obter o resultado pretendido. Sempre que existam instruções de **if-else** encadeadas, o **else** pertence sempre ao **if** mais próximo, que ainda não tenha nenhum **else**. Genericamente:



Exemplo 3.16: Programa que nos indique se o carácter introduzido pelo utilizador é uma letra maiúscula, minúscula, um dígito ou outro carácter.

```
#include <stdio.h>

void main(void){
  char let;

  printf("Introduza um caracter ");
  scanf("%c", &let);

  if (let >= 'a' && let <= 'z')
    printf("Letra minuscula\n");
  else
    if (let >= 'A' && let <= 'Z')
      printf("Letra MAIÚSCULA\n");
    else
      if (let >= '0' && let <= '9')
        printf("Digito\n");
      else
        printf("Outro caracter\n");
}
```

Podemos referi-nos a esta estrutura de **if-else encadeados** como estrutura **if-else** de **alternativa múltipla**. Aqui podemos ver que são executados uma série de testes consecutivos até ocorrer uma das seguintes situações:

- uma das condições da cláusula **if** é verdadeira. Quando isto acontecer as instruções que lhe estão associadas são executadas;
- nenhuma das condições testadas é verdadeira. O programa executa as instruções pertencentes à última cláusula **else** (se existir).

Exemplo 3.17: Este programa permite fazer a conversão de escudos em euros, ou vice-versa, consoante opção do utilizador.

```
#include <stdio.h>

int main(void)
{
 const float euro_esc = 200.482;
 char opc;
 float escudo, euro;

 printf("Conversão monetária \n");
 printf("Euro -> Esc (A)\n");
 printf("Esc -> Euro (B)\n");
 opc=getchar();
/* a função getchar() lê um carácter */

 printf("\n");
 if(opc=='A' || opc=='a'){
  printf("Quantos Euros possui ?\n");
  scanf("%f", &euro);
  escud = euro*euro_esc;
  printf("%.2f Euros correspondem a %.2f Escudos\n", euro, escudo);
 }
 else
  if(opc=='B' || opc=='b'){
   printf("Quantos Escudos possui ?\n");
   scanf("%f", &escud);
   euro = escud/euro_esc;
   printf("%.2f Escudos => a %.2f Euros\n", escudo, euro);
  }
  else
   printf("Opcao incorrecta \n");
}
```

Exemplo 3.18: Programa que calcula a frequência de um sinal periódico, a partir do conhecimento do seu período. Consoante o valor é escolhida a letra adequada para múltiplo (K, M ou G).

```
#include <stdio.h>

int main(void)
{
 float freq, T;
 char mult;

 printf("Qual o período do sinal (em s) ?\n");
 scanf("%f", &T);
 freq = 1/T;
 if(freq<=1000.0)
  mult=' ';
 else
  if(freq<1.0E6){
   mult='K';
   freq = freq/1.0E3;
  }
  else
   if(freq<1.0E9){
    mult='M';
    freq = freq/1.0E6;
   }
   else {
    mult='G';
    freq = freq/1.0E9;
   }
 printf("O sinal com período %.2e s tem frequência %.2f %cHz\n", T, freq, mult);
}
```

3.2.3 A Instrução switch

A instrução **switch** permite implementar a alternativa múltipla. A instrução **switch** é então um modo conveniente de escolher entre várias alternativas. O tipo da expressão_teste terá de ser de um tipo contável (inteiro ou carácter).

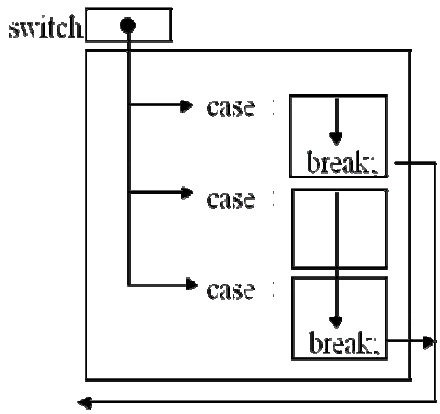
Sintaxe:

```
switch (expressão_teste)
{
   case valor_1:  seq_instrucões_1;
                   [break;]
   case valor_2:  seq_instrucões_2;
                  [break;]
   ...
   case valor_n:  seq_instrucões_n;
                  [break;]

   [default:       seq_instruções_def;]
}
```

onde

- valor_1, ..., valor_n são constantes;
- expressão_teste - tem de possuir resultado compatível com inteiros;
- a instrução **break** é opcional (indicado na sintaxe por [ ]);
- as seq_instruções podem ser vazias; e
- *default* é opcional(indicado na sintaxe por [ ]).

Quando é encontrada uma coincidência entre o resultado da expressão_teste e um dos valores da "lista" (valor_1, ..., valor_n), são executadas as instruções associadas até ser encontrado um **break**, ou até chegar ao fim da instrução **switch** (chaveta final) se não existirem instruções de **break**. Se **default** existir, a seq_instruções_def é executado no caso do resultado da expressão não ser nenhum dos valores da "lista". No caso de **default** não existir, e não se verificar nenhuma coincidência entre expressão_teste e os valores então não é feito nada. Esquematicamente, a instrução **switch** pode ser representada da seguinte forma



Exemplo 3.19: Programa que indica se o carácter escolhido pelo utilizador é uma vogal maiúscula, minúscula, um dígito ou outro carácter, resolvendo isto utilizando a instrução switch.

```
#include <stdio.h>

void main(void)
{
 char car;

 printf("Introduza uma letra ");
 scanf("%c",&car);
 switch (car){
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u': printf("Vogal minuscula\n");
             break;
   case 'A':
   case 'E':
   case 'I':
   case 'O':
   case 'U': printf("Vogal MAIUSCULA\n");
             break;
  case '0':
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9': printf("Digito\n");
            break;
  default: printf("Outro carácter\n");
 }
 printf("Programa terminou\n");
}
```

A instrução **break** faz com que a execução do programa termine a instrução **switch**, e passe à que estiver imediatamente a seguir no código. Nas instruções associadas a **default** não é necessário **break**. De modo a verificar a necessidade da instrução **break** no **switch** experimente executar este programa retirando-lhe o **break** que está a seguir a case 'u':. Na execução introduza uma vogal minúscula. Repare que foi executada a instrução associada a

minúsculas, mas também a que estava a seguir. Isto deve-se ao facto do **switch** não ter sido "quebrado" (falta **break** no sítio certo).

A instrução **switch** é muito utilizada no processamento de menus. Vejamos o exemplo seguinte.

Exemplo 3.20: Programa que para calcular a adição, subtracção, multiplicação ou divisão entre dois números reais. A operação é escolhida pelo utilizador introduzindo um dos caracteres '+', '-', '*' ou '/' consoante a operação a realizar, e de seguida os valores.

```
#include <conio.h>

void main(void)
{
 float num1, num2;
 char op;

 printf("Operaçao a realizar\n");
 printf("+ para Adiçao\n- para Subtracçao\n");
 printf("* para Multiplicaçao\n/ para Divisao\n");
 scanf("%c", &op);
 printf("Introduza dois numeros reais\n");
 scanf("%f%f", &num1, &num2);
 printf("Resultado = ");
 switch (op){
   case '+': printf("%d\n", num1 + num2);
             break;
   case '-': printf("%d\n", num1 - num2);
             break;
   case '*': printf("%d\n", num1 * num2);
             break;
   case '/': if (num2 != 0)
               printf("%f\n", float(num1)/num2);
             else
               printf("Impossivel dividir por zero !");
             break;
     default:  printf("Operacao desconhecida\n");
  }
}
```

Exemplo 3.21: Programa que programa permite efectuar a conversão de escudos em euros, ou vice-versa, consoante opção do utilizador. A escolha é implementada com a instrução switch.

```
#include <stdio.h>

void main(void)
{
 const float euro_esc = 200.482;
 char opc;
 float escudo, euro;

 printf("Conversão monetária \n");
 printf("Euro -> Esc (A)\n");
 printf("Esc -> Euro (B)\n");
 opc=getchar();
 printf("\n");

 switch(opc){
  case 'a':
  case 'A': printf("Quantos Euros possui ?\n");
            scanf("%f", &euro);
            escudo = euro*euro_esc;
            printf("%.2f Euros correspondem a %.2f Escudos\n", euro, escudo);
            break;
  case 'b':
  case 'B':printf("Quantos Escudos possui ?\n");
           scanf("%f", &escudo);
           euro = escudo/euro_esc;
           printf("%.2f Escudos correspondem a %.2f Euros\n", escudo, euro);
           break;
  default: printf("Opcao incorrecta \n");
 }
}
```

3.3 Repetição (Ciclos)

Em **C** existem 3 instruções que permitem implementar ciclos. São elas: **while**, **do-while** e **for**.

3.3.1 Ciclo while

É um ciclo condicional que realiza a iteração enquanto uma determinada condição de teste for verdadeira. Por isso, o ciclo **while** pode nunca fazer a iteração, isto se a expressão testada inicialmente for falsa.

Sintaxe:

```
    while (condição)
        instrução;
ou
    while (condição)
    {
        instrucao_1;
        …
        instrucao_n;
    }
```

Exemplo 3.22: Programa para calcular os quadrados dos 10 primeiros inteiros, apresentados devidamente alinhados.

```
#include <stdio.h>

void main(void)
{
  int i, quad;
  i = 1;
  while (i <= 10)
  {
    quad = i * i;
    printf("%10d ao quadrado é %10d\n", i, quad);
    i++;
  }
}
```

Exemplo 3.23: Programa que lê caracteres e de seguida indica a sua posição na tabela ASCII. O carácter '!' termina a introdução. Na implementação vamos utilizar a função getche. Esta função encontra-se definida em conio.h e lê um carácter proveniente de teclado, sem ser necessário carregar na tecla ENTER.

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

void main(void)
{
  char let = '*';
/*qualquer carácter diferente de '!' garante a   entrada no ciclo*/

  while (let != '!')
 {
    printf("Introduza um carácter ");
    let = getche();
    printf("\nCarácter %c na posição %d\n", let, let);
  }
}
```

3.3.2 O ciclo do-while

Este é um ciclo condicional, que executa a iteração enquanto a condição, ou expressão de teste, for verdadeira, sendo a expressão testada no final do ciclo. Por isto, o ciclo **do-while** é executado sempre pelo menos uma vez.

Sintaxe:

```
do
   instrucao;
while (condição);
```

ou

```
do {
    instrucao_1;
    …
    instrucao_n;
} while (condição);
```

Exemplo 3.24: Programa para calcular a raiz quadrada de um número positivo compreendido entre 1 e 100. O programa deve garantir que o número lido se encontra dentro da gama especificada.

```
#include <stdio.h>
void main(void)
{
 int i, n;
 float ri;

 do {
  printf("Introduza um número entre 1 e 100 ");
  scanf("%d", &n);
 } while (n < 1 || n > 100);
 /* se o valor está for a da gama vamos ler novamente */

 ri = sqrt(i);
 printf("Raiz de %2d = %10.4f\n", i, ri);
}
```

3.3.3 O ciclo for

O ciclo **for** permite que um conjunto de instruções seja executado até o teste da condição ser falso. Muito versátil pois aceita um número de iterações fixas ou condicionais.

Sintaxe:

```
for(expressao_1; expressao_2; expressao_3)
    instrucao;
```

ou

```
for(expressao_1; expressao_2; expressao_3)
{
    instrucao_1;
    …
    instrucao_n;
}
```

A instrução for possui três componentes: expressao_1, expressao_2 e expressao_3, todos elas opcionais. Significado dos três componentes:

- expressao_1 - inicialização das variáveis de controlo do ciclo (pode-se utilizar mais do que uma variável de controlo do ciclo). Só é executada uma vez, quando se entra no ciclo;
- expressao_2 - condição que determina se o ciclo deve continuar. Se o resultado desta expressão for verdadeiro o ciclo continua, se for falso o ciclo termina; e
- expressao_3 - actualização das variáveis de controlo do ciclo (muitas vezes uma incrementação ou decrementação).

Exemplo 3.25: Programa para calcular o quadrado dos 20 primeiros inteiros.

```
#include <stdio.h>

void main(void)
{
 int i, quad;
 for (i = 1; i <= 20; i++)
 {
   quad = i * i;
   printf("%2d*%2d=%3d\n", i, i, quad);
 }
}
```

Observação: O último componente do ciclo **for** também podia ser: i += 1; ou i = i + 1;

Exemplo 3.26: Um programa para calcular o factorial de um número entre 0 e 12.

```
#include <stdio.h>

void main(void)
{
 int i, n, fact;

 printf("Introduza um numero entre 1 e 12");
 scanf("%d", &n);

 fact = 1;
 if (n >= 0 && n <= 12)
 {
   for (i = 1; i <= n; i++)
     fact = fact * i; /* ou fact *= i; */
 /* a instrução seguinte não faz parte do ciclo */
   printf("%d!=%d\n", n, fact);
 }
 else
   printf("Nao é possivel calcular !");
}
```

O incremento do ciclo não tem de ser unitário, pode ter outro valor. É também claro que pode ser uma decrementação. Na realidade pode ser uma operação qualquer válida para o tipo de variáveis de controlo utilizadas.

Exemplo 3.27: Programa para apresentar os quadrados das potências inteiras de base 2 inferiores a 1024, por ordem decrescente.

```
#include <stdio.h>

void main(void)
{
  int i;

 /* em cada iteração i passa a metade */
  for (i = 1024; i >=1; i = i/2)
    printf("%d\t%d\n", i, i*i);
}
```

Como foi referido anteriormente, não é necessário que existam os três componentes do ciclo for. No exemplo seguinte mostra-se a implementação do programa do Exemplo 3.27 onde só existe o componente que testa a continuação do ciclo. É então necessário fazer a inicialização antes do ciclo e a incrementação dentro do ciclo.

Exemplo 3.28: Outra hipótese de resolver o programa do Exemplo 3.27.

```
#include <stdio.h>

void main(void)
{
int i, quad;

  i = 1;/* inicialização da variável de controlo */
  for( ; i <= 20; )
 {
    quad = i * i;
    printf("%2d*%2d=%3d\n", i, i, quad);
    i++;/* incrementação da variavel de controlo dentro do ciclo*/
 }
}
```

O mesmo programa pode ainda ser resolvido utilizando a instrução **break** e omitindo as 3 componentes do ciclo **for**.

Exemplo 3.29: Outra hipótese de resolver o programa do Exemplo 3.27 e 3.28.

```
#include <stdio.h>

void main(void)
{
 int i, quad;
 i = 1;/* inicialização da variável de controlo */
 for( ; ; )
 {
   if(i > 20)
     break;/* saída do ciclo, quando condição verdadeira */
   quad = i * i;
   printf("%2d*%2d=%3d\n", i, i, quad);
   i++;/* incrementação da variavel de controlo */
 }
}
```

Neste caso a instrução **break** permite criar aquilo que se designa por ciclo aberto. A instrução **break** é então utilizada para sair do ciclo, logo que se verifique uma determinada condição. A sintaxe da utilização da instrução **break** num ciclo **for** é:

```
for(inicializações; continuação; actualização)
{
    <sequencia_inst_1>
    if(condição_saida_ciclo)
    break;
    <sequencia_inst_2>
}
<outras_inst>
```

Se a condição de saída do ciclo for verdadeira, o ciclo termina (sequencia_inst_2 já não é executada) e a próxima instrução é a primeira que estiver a seguir ao ciclo (outras_inst).

A instrução break é genérica, não estando ligada apenas ao ciclo for, pois pode ser utilizada para "quebrar" qualquer estrutura de controlo. Já vimos anteriormente a sua utilização para sair da instrução **switch**.

Para finalizar podemos ver que é possível reescrever o código que usa a instrução **for** por outro que utiliza o ciclo **while**. Assim,

```
for(expressao_1; expressao_2; expressao_3)
{
   <sequencia_inst>
}
```

é equivalente a

```
expressao_1;
while(expressao_2)
{
   <sequência_inst>
expressao_3;
}
```

3.3.4 Ciclos Encadeados

Se a instrução que está dentro de um ciclo for outro ciclo então estaremos em presença de ciclos encadeados.

Exemplo 3.30: programa para ler 10 valores inteiros compreendidos entre 1 e 100, e que apresente os quadrados respectivos.

```
#include <stdio.h>

void main(void)
{
 int i, num, quad;

 for(i=0; i<10; i++)
 {
  /* ciclo do-while esta dentro do ciclo for */
  do{
   printf("Introduza um numero entre 0 e 100 - ");
   scanf("%d", &num);
  } while(num<0 || num>100);
  /* garantimos que o valor lido esta entre 1 e 100 */
  quad = num*num;
  printf("%d ao quadrado = %d \n", num, quad);
 }
}
```

Exemplo 3.31: Programa para calcular o somatório seguinte: $S = \sum_{i=5}^{25} \left(i + \frac{i}{2} + \frac{i}{3} \right)^2$, utilizando o ciclo **while**.

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

void main(void){
  int i;
  float soma;
  soma = 0.0;

  i=5;
  while(i <= 25){
    soma = soma + pow((i+i/2.0+i/3.0),2);
    i++;
  }
  printf("Soma dos valores = %.3f\n", soma);
}
```

Exemplo 3.32: Programa para calcular o somatório seguinte: $S = \sum_{i=5}^{25} \left(i + \frac{i}{2} + \frac{i}{3} \right)^2$, utilizando o ciclo **do-while**.

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

void main(void){
  int i;
  float soma;

  soma = 0.0;

  i=5;
  do{
    soma = soma + pow((i+i/2.0+i/3.0),2);
    i++;
  } while(i <= 25);

  printf("Soma dos valores = %.3f\n", soma);
}
```

Exemplo 3.33: Programa para calcular o somatório seguinte: $S = \sum_{i=5}^{25} \left(i + \frac{i}{2} + \frac{i}{3} \right)^2$, utilizando o ciclo **for**.

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

void main(void){
  int i;
  float soma;

  soma = 0.0;

  for(i=5; i<=25; i++)
    soma = soma + pow((i+i/2.0+i/3.0),2);
  printf("Soma dos valores = %.3f\n", soma);
}
```

Exemplo 3.34: Programa que leia um número inteiro **n** e apresente o número, o quadrado, o cubo e a quarta potências dos números compreendidos entre **1** e **n**. **Se** n=3, o programa deve mostar:

```
Numero    Quadrado    Cubo    Quarta
-------------------------------------
    1         1        1        1
    2         4        8       16
    3         9       27      243
```

```
#include <stdio.h>

void main(void){
  int i, n, quad, cub, quart;

  printf("Qual o numero ?\n");
  scanf("%d", &n);

  printf("Numero   Quadrado   Cubo   Quarta\n");
  for(i=1; i<37; i++)
    printf("-");
  printf("\n");

  for(i=1; i<=n; i++)
  {
    quad = i * i;     cub  = quad * i;     quart= cub * i;
    printf("%6d%11d%8d%10d\n", i, quad, cub, quart);
  }
}
```

Exemplo 3.35: Altere o programa do exemplo anterior de modo a apresentar as potências inversas. **Se** n=3, o programa deve mostar:

```
Numero    1/Quadrado     1/Cubo    1/Quarta
-------------------------------------------------------
1         1              1         1
2         0.25           0.125     0.0625
3         0.111111       0.037037  0.0123457
```

```
#include <stdio.h>

void main(void){
  int i, n, quad, cub, quart;

  printf("Qual o numero ?\n");
  scanf("%d", &n);

  printf("Numero   1/Quadrado    1/Cubo       1/Quarta\n");
  for(i=1; i<45; i++)
    printf("-");
  printf("\n");

  for(i=1; i<=n; i++)
  {
    quad = i * i;     cub  = quad * i;     quart= cub * i;
    printf("%-9d%-14g%-13g%-14g\n", i, 1.0/quad, 1.0/cub, 1.0/quart);
  }
}
```

Exemplo 3.36: Programa que lê dois números inteiros **m** e **n** e que calcula a soma dos números pares e dos números impares entre **m** e **n**, inclusive. Utilizando o ciclo **for**.

```
#include <stdio.h>

void main(void)
{
 int i, m, n;
 int s_par, s_impar;

 s_par = 0;
 s_impar = 0;
 printf("Introduza dois valores\n");
 scanf("%d%d", &m, &n);

 for(i = m; i <= n; i++){
  if(i %2 == 0)
    s_par = s_par + i;
  else
    s_impar = s_impar + i;
 }
 printf("Soma dos pares é %d\n", s_par);
 printf("Soma dos impares é %d\n", s_impar);
}
```

Exemplo 3.37: Programa que lê dois números inteiros **m** e **n** e que calcula a soma dos números pares e dos números impares entre **m** e **n**, inclusive. Utilizando o ciclo **while**.

```
#include <stdio.h>

void main(void)
{
 int i, m, n;
 int s_par, s_impar;

 s_par = 0;   s_impar = 0;

 printf("Introduza dois valores\n");
 scanf("%d%d", &m, &n);

 i = m;
 while(i <= n)
{
  if(i %2 == 0)
    s_par = s_par + i;
  else
    s_impar = s_impar + i;
  i++;
 }

 printf("Soma dos pares é %d\n", s_par);
 printf("Soma dos impares é %d\n", s_impar);
}
```

Exemplo 3.38: Programa que lê dois números inteiros **m** e **n** e que calcula a soma dos números pares e dos números impares entre **m** e **n**, inclusive. Utilizando um ciclo **do-while** para garantir que n>m e que são ambos positivos.

```
#include <stdio.h>

void main(void)
{
 int i, m, n;
 int s_par, s_impar;

 s_par = 0;
 s_impar = 0;

 do{
  printf("Introduza dois valores\n");
  scanf("%d%d", &m, &n);
 } while( m>=n || n<1 || m<1);

 for(i = m; i <= n; i++){
  if(i %2 == 0)
    s_par = s_par + i;
  else
    s_impar = s_impar + i;
 }

 printf("Soma dos pares é %d\n", s_par);
 printf("Soma dos impares é %d\n", s_impar);
}
```

Um procedimento conhecido como método de Newton-Raphson permite-nos calcular a raiz quadrada de um número real positivo **a**, do seguinte modo (algoritmo):

- Inicializar x com 1.0
- Atribuir a y o resultado de (x+a/x)/2
- Se x e y ainda não forem suficientemente próximos atribua y a x, e recalcule y. Isto é feito até que a diferença entre x e y seja negligenciável (inferior a um determinado erro fixado à partida).

Exemplo 3.39: O programa para ler um número e o erro admissível, p. ex. 0,00 e calcular a raiz quadrada através do Método de Newton-Raphson.

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

int main(void)
{
 double a, x, y, err;

 printf("Digite valor do qual quer calcular a raiz quadrada \n");
 scanf("%lf", &a);
 printf("Qual o erro permitido \n");
 scanf("%lf", &err);  /* leitura de double exige especificador %lf ou %le */


 x = 1;
 y = (x + a / x)/2;
 while( x-y > err || x-y < -err){
   x = y;
   y = (x + a / x)/2;
 }

 printf("Raiz MNR =%lf -> Raiz com sqrt()=%lf, e erro efectivo = %lf\n", y, sqrt(a)
        , y-sqrt(a));
}
```

Exemplo 3.40: Escreva um programa que leia uma sequência de números inteiros de um só dígito, terminando com a introdução do dígito 0, e conte o número de vezes que cada dígito ocorre. O resultado deve ser apresentado na forma de um histograma. Por exemplo, com a entrada:

3 1 4 2 6 9 6 7 1 6 3 1 9 6 7 0	a saída deve ser	6 1 6 1 3 6 7 9 1 2 3 4 6 7 9 ========== 1 2 3 4 5 6 7 8 9

pois o utilizador introduziu três 1's, um 2, dois 3´s , ... Se algum dígito ocorrer mais de 10 vezes, o programa deve ser parado, sendo apresentada uma mensagem de erro. Tal também deve acontecer se for introduzido um número negativo ou de dois dígitos.

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

int main(void)
{
int n1 = 0, n2 = 0, n3 = 0, n4 = 0, n5 = 0, n6 = 0, n7 = 0, n8 = 0, n9 = 0;
int erro = 0;
int i, a;

 do{
   scanf("%d", &a);
   switch( a ){
     case 1: n1++; break;
     case 2: n2++; break;
     case 3: n3++; break;
     case 4: n4++; break;
     case 5: n5++; break;
     case 6: n6++; break;
     case 7: n7++; break;
     case 8: n8++; break;
     case 9: n9++;
   }
 if(n1 > 10 || n2 > 10 || n3 > 10 || n4 > 10 || n5 > 10 || n6 > 10 || n7 > 10 || n8 > 10 ||
    n9 > 10 || a >= 10  || a < 0)
{
    erro = 1;
     break;
 }
 } while(a != 0);
 if(erro == 1)
   printf("Erro na introdução de dados\n");
 else{
   for(i=9; i>=1; i--)
   {
    if(n1>=i)
      printf("1 ");
    else
      printf("  ");
    if(n2>=i)
      printf("2 ");
    else
      printf("  ");
    if(n3>=i)
      printf("3 ");
    else
      printf("  ");
```

```
    if(n4>=i)
        printf("4 ");
    else
        printf("  ");
    if(n5>=i)
        printf("5 ");
    else
        printf("  ");
    if(n6>=i)
        printf("6 ");
    else
        printf("  ");
    if(n7>=i)
        printf("7 ");
    else
        printf("  ");
    if(n8>=i)
        printf("8 ");
    else
        printf("  ");
    if(n9>=i)
        printf("9\n");
    else
        printf("\n");
   }
   for(i = 1; i <= 9; i++)
     printf("==");
   printf("\n");

   for(i = 1; i <= 9; i++)
     printf("%d ", i);
   printf("\n");
  }
}
```

4. Funções

Uma função é uma unidade autónoma de código do programa, projectada para cumprir uma determinada tarefa.

Vantagens:

- divisão de tarefas grandes em várias tarefas pequenas;
- ocultação de detalhes de operação de partes do programa, facilitando a compreensão do programa;
- reutilização de código e consequente economia de trabalho;
- mais fácil de testar e optimizar (número pequeno de linhas de código);
- diminuição do número total de linhas de um programa quando existem tarefas repetidas ao longo do código.

Sintaxe da definição de uma função:

```
tipo_retorno identif_func(lista de parâmetros)
{
   <declarações de constantes, tipos e variáveis>

   <uma ou mais instruções>

   return <expressão>;
}
```

Cabeçalho

Corpo da função

Uma função antes de ser utilizada tem de ser previamente definida ou conhecido o seu protótipo. Vamos começar por ver um pequeno programa que desenha a letra **E** (como se indica a seguir) com asteriscos e de seguida vamos alterá-lo de modo a utilizar funções.

```
*******
*******
*
*
*******
*******
*
*
*******
*******
```

Exemplo 4.1: Programa para desenhar a letra "**E**".

```
#include <stdio.h>

void main(void)
{
  printf("*******\n");
  printf("*******\n");
  printf("*\n");
  printf("*\n");
  printf("*******\n");
  printf("*******\n");
  printf("*\n");
  printf("*\n");
  printf("*******\n");
  printf("*******\n");
}
```

Podemos verificar que há repetição de código, e identificar duas funções possíveis: uma para desenhar as duas linhas de 7 asteriscos na horizontal e outra para desenhar 2 linhas de um asterisco, a que vamos chamar **horizontal()** e **vertical()**.

Exemplo 4.2: Programa para desenhar a letra "**E**". Utilizando duas funções horizontal e vertical.

```
#include <stdio.h>

void horizontal(void)
{
  printf("*******\n");
  printf("*******\n");
  return ; /* esta linha pode ser omitida */
}

void vertical(void)
{
  printf("*\n");
  printf("*\n");
}
void main(void)
{
  horizontal();
  vertical();
  horizontal();
  vertical();
  horizontal();

}
```

Se uma função não tem valor de retorno, então tipo_retorno é **void**. Na chamada de uma função os parênteses servem para distinguir o identificador de uma função do identificador de variáveis. O facto de se definir uma função não significa que ela seja executada. Ela só é executada quando for chamada por uma outra função; no exemplo dentro da função **main()**.

Exemplo 4.3: Programa para desenhar a letra "**L**" e "**C**", com as funções horizontal e vertical.

```
#include <stdio.h>

void horizontal(void)
{
  printf("*******\n");
  printf("*******\n");
}

void vertical(void)
{
  printf("*\n");
  printf("*\n");
}

void main(void)
{
  int i;
  printf("\nLetra L\n");
  for (i = 1; i <= 3; i++)
    vertical();
  horizontal();

  printf("\nLetra C\n")
  horizontal();
  for (i = 1; i <= 3; i++)
    vertical();
  horizontal();
}
```

Vamos redefinir a função **horizontal()** de modo a poder variar o número de asteriscos a desenhar.

Exemplo 4.4: Função horizontal com um número variável de "*".

```
void horizontal(int n_col)
{
  int j;
  for (j = 1; j <= n_col; j++)
    printf("*");
  printf("\n"); /* esta linha está fora do ciclo */
  for (j = 1; j <= n_col; j++)
    printf("*");
  printf("\n"); /* esta linha está fora do ciclo */
  return;
}
```

Exemplo 4.5: O código da função do exemplo 4.3 pode ser optimizado através do encadeamento de ciclos.

```
void horizontal(int n_col)
{
  int i, j;
  for (i = 1; i <= 2; i++){ /* para cada linha faz */
    for (j = 1; j <= n_col; j++)
      printf("*");
    printf("\n");
  }
}
```

Exemplo 4.6: O programa para desenhar a letra "**E**", tal como definida anteriormente, e utilizando a nova definição da função horizontal.

```
#include <stdio.h>
void horizontal(int n_col)
{
  int i, j;

  for (i = 1; i <= 2; i++){
    for (j = 1; j <= n_col; j++)
      printf("*");
  printf("\n");
  }
  return ;
}
void main(void)
{
  horizontal(7);
  horizontal(1);
  horizontal(7);
  horizontal(1);
  horizontal(7);
}
```

Vamos melhorar a função **horizontal()** de modo a aceitar, para além do número de asteriscos numa linha, o número de linhas e o carácter. Aproveitamos para lhe alterar o nome para **linhas**. A nova função tem então três parâmetros: dois inteiros e um carácter.

Exemplo 4.7: Função que permite desenhar um número variável de linhas horizontais, com comprimento variável.

```
void linhas(int n_col, int n_lin, char let)
{
  int i, j;

  for (i = 1; i <= n_lin; i++){  /* para cada linha faz */
    for (j = 1; j <= n_col; j++)
      printf("%c", let);
    printf("\n");
  }
  return ;
}
```

Exemplo 4.8: Programa para desenhar a letra "E" da seguinte forma:

```
EEEEEEE
II
!!!!!!!
L
$$$$$$$$
```

```
#include <stdio.h>

void linhas(int n_col, int n_lin, char let)
{
  int i, j;

  for (i = 1; i <= n_lin; i++){
    for (j = 1; j <= n_col; j++)
      printf("%c", let);
    printf("\n");
  }
  return ;
}
void main(void)
{
  linhas(7, 1, 'E');
  linhas(2, 1, 'I');
  linhas(7, 1, '!');
  linhas(8, 2, '$');
}
```

4.1 Parâmetros e Argumentos

Quando uma função é definida existem duas partes distintas: o cabeçalho e o corpo da função. O cabeçalho é composto pelo **tipo do retorno** da função, **nome** da função e **lista de parâmetros** da função. Na chamada ou invocação de uma função o número e tipo dos **argumentos** deve coincidir com os **parâmetros** que se encontram na definição da função. Deve existir uma relação biunívoca entre **argumentos** e **parâmetros**.

4.2 Utilização do Protótipo da Função

O protótipo de uma função é composto pelo cabeçalho da função, sendo a linha terminada por ponto e vírgula. No protótipo de uma função podem omitir-se os identificadores dos parâmetros, no entanto a sua presença é útil para melhor documentar o programa e facilitam a identificação de mensagens de erro emitidas pelo compilador.

Utilizando o protótipo é possível utilizar uma função antes de ela se encontrar definida. O programa anterior pode ser rescrito de modo a utilizar o protótipo da função **linhas()**.

Exemplo 4.9: Programa com utilização de protótipos.

```
#include <stdio.h>
/* protótipo da função linhas */
void linhas(int n_col, int n_lin, char let);

/* o protótipo desta função também se poderia ter escrito da seguinte forma
void linhas(int , int , char );    no entanto é menos informativo */

void main(void)
{
  linhas(7, 2, 'E');
  linhas(2, 2, 'I');
  linhas(5, 1, '!');
  linhas(1, 3, 'L');
  linhas(8, 2, '$');
}
```

```
/* definição da função */
void linhas(int n_col, int n_lin, char let)
{
 int i, j;

 for (i = 1; i <= n_lin; i++){
   for (j = 1; j <= n_col; j++)
     printf("%c", let);
   printf("\n");
 }
 return ;
}
```

Quando da chamada da função argumentos não têm de ser apenas constantes. Cada argumento é uma expressão cujo resultado deve ser do tipo esperado na definição da função. Por exemplo na função **main** do programa anterior podíamos ter:

Exemplo 4.10: Programa do Exemplo 4.9 com utilização de protótipos.

```
 . . .
void main(void)
{
 const int N = 10;
 int x = 1;
 ...
 linhas(N-2, abs(x-N/2), 'p');
 /* experimente !! .*/
 /* neste caso o 1º, 2º e 3º argumentos são 8 (de 10-2),
 4 ( de abs(1-10/2) ) e 'p' */
 . . .
}
```

Exercício proposto: Escreva uma função cujo protótipo é **void tabuada(int n, int m);** que escreve no monitor tabela como se mostra a seguir (supôs-se a chamada tabuada(3, 5);). **m** e **n** devem poder variar entre 1 e 10 . Se **m** ou **n** estiverem fora desta gama então deve ser apresentada uma mensagem de erro. Escreva ainda uma função **main** que lhe permita testar **tabuada**.

```
X      1     2     3     4     5
  |---------------------------------------
1 |    1     2     3     4     5
2 |    2     4     6     8    10
3 |    3     6     9    12    15
```

4.3 A instrução return

A instrução **return** serve para definir o resultado a devolver por uma função. O valor a devolver é o resultado da expressão colocada à frente da palavra reservada **return**. A instrução return apenas permite retornar uma variável ou expressão.

Se for utilizado **return** mais que uma vez na mesma função, então quando o fluxo do programa atinge essa instrução o retorno da função é efectuado imediatamente (as instruções que se encontram a seguir já não se realizam).

Exemplo 4.11: Função para ler uma letra minúscula. Esta função só termina quando o utilizador introduzir uma letra minúscula. O valor retornado é o carácter lido.

```
char le_minusc(void)
{
  char let;
  do
    let = getche();
  while (let < 'a' || let > 'z');

  return let;
}
```

Exemplo 4.12: Programa que utiliza a função do Exemplo 4.12 e apresente a letra lida assim como a sua posição na tabela ASCII.

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

char le_minusc(void);

void main(void)
{
  char letra;
  letra = le_minusc();
  printf("Carácter %c na posição %d\n", letra, letra)
}

char le_minusc(void)
{
  char let;
  do
    let = getche();
  while (let < 'a' || let > 'z');
  return let;
}
```

Exemplo 4.13: Função para calcular o módulo do cubo da soma de dois valores reais. O valor retornado pela função deve ser do tipo double. Mostram-se a seguir várias implementações possíveis para esta função.

```
double mod_cubo(float x, float y) /* Resolução 1*/
{
 double cubo;
 double xy;

 xy = double(x + y);
 if (xy > 0.0)
   cubo = xy*xy*xy;
 else
   cubo = -xy*xy*xy);
 return cubo;
}
```

```
double mod_cubo(float x, float y) /* Resolução 2*/
{
 double cubo;
 double xy;

 xy = double(x + y);
 cubo = xy*xy*xy;
 if (xy < 0.0)
   cubo = -cubo;
 return cubo;
}
```

```
double mod_cubo(float x, float y) /* Resolução 3*/
{
 double xy;

 xy = double(x + y);
 if (xy > 0.0)
   return xy*xy*xy;
 else
   return -xy*xy*xy;
}
```

```
double mod_cubo(float x, float y) /* Resolução 4*/
{
 double xy;

 xy = double(x + y);
 if (xy < 0.0)
   xy = -xy;
 return xy*xy*xy;
}
```

Exemplo 4.14: Pretendemos um programa que calcule a seguinte expressão matemática $z = |u+v|^3 + |u-v|^6$ onde u e v são valores introduzidos pelo utilizador. O programa deve utilizar a função mod_cubo implementada no Exemplo 4.13.

```
#include <stdio.h>

/* protótipo da função – Resolução 4*/
double mod_cubo(float x, float y);

void main(void)
{
 double final;
 float u, v;
 printf("Introduza 2 valores\n");
 scanf("%f%f", &u, &v);

 final = mod_cubo(u, v) + mod_cubo(u, -v)*mod_cubo(u, -v);

 printf("Resultado = %lf\n", final);
}
double mod_cubo(float x, float y)
{
 double xy;

 xy = double(x + y);
 if (xy < 0.0)
   xy = -xy;

return xy*xy*xy;
}
```

Se o tipo de retorno de uma função for int, então este pode ser omitido.

Por exemplo a definição da função func :

```
int func(int x, int y)
{
 return -x*y;
}
```

pode ser escrita

```
func(int x, int y)
{
 return -x*y;
}
```

Exemplo 4.15: Escreva uma função para desenhar uma figura do género da seguinte, supôs-se que a chamada da função foi quad(7,8,1,2,'f');

```
fghijkl
mnopqrs
t     u
v     w
x     y
z     a
bcdefgh
ijklmno
```

```
#include <stdio.h>
/* passa para a letra seguinte; se ultrapassar 'z' volta a 'a' */
char devolve(char l)
{
 l++;
 if(l>'z')
   l='a';
 return l;
}
/* larg é a largura da figura; alt a altura; gross_h é a espessura na horizontal e gross_v a
espessura vertical; let é o primeira letra */
void quad(int larg, int alt, int gross_h, int gross_v, char let)
{
 int i, j;
 char letra = let;

 for(i=1; i<=gross_v; i++){
   for(j=1;j<=larg;j++){
     printf("%c",letra);
     letra=devolve(letra);
   }
   printf("\n") ;
 }
 for(i=1; i<=alt-gross_v*2; i++){
   for(j=1; j<=gross_h; j++){
     printf("%c",letra);
     letra=devolve(letra);
   }
   for(j=1; j<=larg-gross_h*2; j++)
     printf(" ");
   for(j=1; j<=gross_h; j++){
     printf("%c",letra);
     letra=devolve(letra);
   }
   printf("\n") ;
 }
```

```
  for(i=1; i<=gross_v; i++){
    for(j=1; j<=larg; j++){
      printf("%c",letra);
      letra=devolve(letra);
    }
    printf("\n") ;
  }
}
```

Exemplo 4.16: Escreva as seguintes funções: **a)** Testar se um carácter é ou não uma vogal; **b)**Testar se um carácter é ou não uma letra do alfabeto; **c)** Testar se um carácter é ou não um dígito; e **d)** Escreva a função main que teste as funções anteriores.

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

int testaVogal(char let) /*Função a)*/
{
  int t;
  switch(let){
    case 'a':
    case 'A':
    case 'e':
    case 'E':
    case 'i':
    case 'I':
    case 'o':
    case 'O':
    case 'u':
    case 'U': t = 1;
              break;
    default:  t = 0;
  }
  return t;
}

int testaLetra(char let) /*Função b)*/
{
  int t;
  if(let>='a' && let<='z' || let>='A' && let<='Z')
    t = 1;
  else
    t = 0;
  return t;
}
```

```
int testaDigito(char let) /*Função c)*/
{
  int t = 0;
  if(let>='0' && let<='9')
    t = 1;
  return t;
}

int main(void) /*Função d)*/
{
  char car;

  printf("Introduza um caracter\n");
  car = getche();
  if(testaVogal(car)==1) /*  if(testaVogal(car))  */
    printf("\nÉ vogal\n");
  else
    printf("\nNão é vogal\n");

  if(testaLetra(car)==1)
    printf("É uma letra do alfabeto\n");
  else
    printf("Não é uma letra do alfabeto\n");

  if(testaDigito(car)==1)
    printf("É um digito\n");
  else
    printf("Não é um digito\n");
}
```

Exemplo 4.17: Escreva funções para: **a)** Determinar qual o maior de dois números reais; **b)** Qual o maior de três números reais; e **c)** Escreva a função anterior, mas agora recorra à função para determinar o maior de dois valores anteriormente escrita. Apresente duas implementações alternativas.

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

float maior2(float a, float b) /*Função a)*/
{
  float m;
  m = a;
  if(b>m)
    m = b;
  return m;
}
```

```
float maior3(float a, float b, float c) /*Função b)*/
{
 float m;
 m = a;
 if(b>m)
   m = b;
 if(c>m)
   m = c;
 return m;
}

float maior3a(float a, float b, float c) /*Função c)*/
{
 float m;
 m = maior2(a,b);
 m = maior2(m,c);
 return m;
}


float maior3b(float a, float b, float c)  /*Função c)*/
{
 return maior2(a, maior2(b,c));
}

/* Para verificar o correcto funcionamento das funções*/
void main(void)
{
 float x=1.0, y=5.0, z=20.0;

 printf("O maior de %f e %f , %f\n", x, y,maior2(x,y));
 printf("O maior de %f , %f e %f , %f\n", x, y, z,maior3(x,y,z));
 printf("O maior de %f , %f e %f , %f\n", x, y, z,maior3a(x,y,z));
 printf("O maior de %f , %f e %f , %f\n", x, y, z,maior3b(x,y,z));
 printf("O maior de %f , %f e %f , %f\n", x, y, z,maior2(maior2(x,y),z));
}
```

4.4 Variáveis Globais versus Locais

Todos as constantes, tipos e variáveis declaradas fora dos blocos das funções são globais, ou seja, são "conhecidas" em todo o programa. As constantes, tipos e variáveis declaradas nos blocos das funções são "conhecidas" apenas dentro do bloco onde são declaradas, logo são constantes, tipos ou variáveis locais. Sempre que uma constante, um tipo ou uma

variável seja redeclarada noutro bloco, então é essa declaração que passa a ser válida.

Vamos exemplificar os conceitos de global e local através de um pequeno programa em **C**, que deve ser compilado e executado, observando com atenção o que é escrito no monitor.

Exemplo 4.18: Global versus local.

```
#include <stdio.h>

const int MAX = 100; /* MAX é uma constante global*/

float x = 1300.125; /* x é uma variável global, inicializada com o valor 1300.125 */

void func(void)
{
  int n, x = 1;
  /* n e x são variáveis locais de func()*/
  /* x global não é válida neste bloco   */

  n = 10;
  printf("\n\nDentro de func %d e %d\n", n, x);
  /* a constante global MAX também é conhecida neste bloco */
  printf("Dentro de func MAX%d\n\n", MAX);
}

void main(void)
{
  int n; /* n é uma variável local de main() */
  float MAX = 1.1;
  /* MAX é agora uma variável local de main()        */
  /* a constante global MAX não é válida neste bloco */

  n = 1;
  printf("em main valor de n = %d\n", n);
  printf("em main valor de MAX = %f\n", MAX);
  /* a variável global x também é conhecida neste bloco */
  printf("em main  x = %d\n\n", x);

  func();
}
```

4.4.1 Efeitos Laterais

A alteração do valor de variáveis globais dentro de funções pode originar erros na execução de um programa. VAMOS VER UM EXEMPLO DE MÁ PRÁTICA DE PROGRAMAÇÃO.

Exemplo 4.19: Efeitos Laterais.

```
#include <stdio.h>
float num;
/* num é uma variável global */

float funct(int x)
{
  num = num + 1.0;
  return x*x - num/2;
}

void main(void)
{
  float x;
  num = 1.0;

  x = funct(2) + funct(2);
}
```

Analisando o Exemplo 4.19 verificamos que a primeira vez que a função funct é chamada retorna o valor **3** e na segunda chamada da função o valor **2.5**, e se fosse chamada novamente iria devolver um valor diferente. **Isto está errado**, pois uma função com os mesmos argumentos deve comportar-se sempre da mesma maneira, o que não é o caso pois o valor devolvido depende do número de vezes que a função já foi utilizada.

Assim, sempre que o valor de variáveis globais é alterado dentro de funções temos efeitos laterais. Estes efeitos tornam o programa difícil de entender e podem originar, e muitas vezes originam, erros. De modo a tornar o programa claro e seguro, e de acordo com as boas regras da programação, a utilização de variáveis globais não é permitida. Ou melhor é apenas permitida em situações muito excepcionais. A utilização de constantes e tipos globais são desejáveis em algumas situações.

5. Tabelas (*Arrays*)

Para entendermos a necessidade da utilização de **tabelas**[1] **(*arrays*)** vamos começar por ver um exemplo. Suponha que se pretendia fazer um estudo estatístico das notas de uma cadeira, numa turma com 10 alunos. Pretende-se calcular a média e se esta for inferior a 12 valores apresentar todas as notas. Como as notas devem ser apresentadas no caso da média ser inferior a 12 valores, é necessário utilizar 10 variáveis do tipo **int** para as armazenar. Um programa em **C** para resolver este problema é apresentado no Exemplo 5.1.

Exemplo 5.1: Programa para calcular a média das notas de uma cadeira, numa turma com 10 alunos e calcular a média. Se esta for inferior a 12 valores apresentar todas as notas.

```
#include <stdio.h>

void main(void)
{
int nota1, nota2, nota3, nota4, nota5, nota6, nota7, nota8, nota9, nota10;
float media;

/* ler 10 notas */
  printf("Introduza valor ");
  scanf("%d",&nota1);
  printf("Introduza valor ");
  scanf("%d",&nota2);
  printf("Introduza valor ");
  scanf("%d",&nota3);
  printf("Introduza valor ");
  scanf("%d",&nota4);
  printf("Introduza valor ");
  scanf("%d",&nota5);
  printf("Introduza valor ");
  scanf("%d",&nota6);
  printf("Introduza valor ");
  scanf("%d",&nota7);
  printf("Introduza valor ");
  scanf("%d",&nota8);
  printf("Introduza valor ");
  scanf("%d",&nota9);
  printf("Introduza valor ");
```

[1] A designação de vector é também muito usual. No caso do vector ter apenas uma dimensão pode designar-se por vector unidimensional ou apenas vector, no caso de duas dimensões pode designar-se por matriz, nos outros casos é designado por vector multidimensional.

```
  scanf("%d",&nota10);
/* calcula media – o modificador de formato tem prioridade superior à divisão */
  media = float(nota1+nota2+nota3+nota4+nota5+nota6+ nota7+ nota8+nota9+ nota10)/10;
  printf("media = %6.2f\n\n", media);

/* se média inferior a 12 valores então apresenta notas */
  if (media < 12.0){
   printf("Notas dos alunos\n");
   printf("%d\n",nota1);
   printf("%d\n",nota2);
   printf("%d\n",nota3);
   printf("%d\n",nota4);
   printf("%d\n",nota5);
   printf("%d\n",nota6);
   printf("%d\n",nota7);
   printf("%d\n",nota8);
   printf("%d\n",nota9);
   printf("%d\n",nota10);
  }
}
```

E se a turma tiver **100** alunos ? Tenho de declarar 100 variáveis do tipo int !!

```
int nota1, nota2, nota3, nota4, nota5, nota6, nota7, nota8, nota9, nota10, nota11, nota12,
nota13, nota14, nota15, nota16, nota17, nota18, nota19, nota20, nota21, nota22, nota23,
nota24, nota25, nota26, nota27, nota28, nota29, nota30, nota31, nota32, nota33, nota34,
nota35, nota36, nota37, nota38, nota39, nota40, nota41, nota42, nota43, nota44, nota45,
nota46, nota47, nota48, nota49, nota50, nota51, nota52, nota53, nota54, nota55, nota56,
nota57, nota58, nota59, nota60, nota61, nota62, nota63, nota64, nota65, nota66, nota67,
nota68, nota69, nota70, nota71, nota72, nota73, nota74, nota75, nota76, nota77, nota78,
nota79, nota80, nota81, nota82, nota83, nota84, nota85, nota86, nota87, nota88, nota89,
nota90, nota91, nota92, nota93, nota94, nota95, nota96, nota97, nota98, nota99,
nota100;
```

Para lermos as 100 notas temos de utilizar 100 pares de instruções do género:

```
printf("Introduza valor ");
scanf("%d",&nota1);
```

Para calcular a média temos a expressão:

```
media = float(nota1 + nota2 + nota3 + nota4 + nota5 + nota6 + nota7 + nota8 + nota9 +
nota10 + nota11 + nota12 + nota13 + nota14 + nota15 + nota16 + nota17 + nota18 +
nota19 + nota20 + nota21 + nota22 + nota23 + nota24 + nota25 + nota26 + nota27 +
nota28 + nota29 + nota30 + nota31 + nota32 + nota33 + nota34 + nota35 + nota36 +
nota37 + nota38 + nota39 + nota40 + nota41 + nota42 + nota43 + nota44 + nota45 +
nota46 + nota47 + nota48 + nota49 + nota50 + nota51 + nota52 + nota53 + nota54 +
nota55 + nota56 + nota57 + nota58 + nota59 + nota60 + nota61 + nota62 + nota63 +
nota64 + nota65 + nota66 + nota67 + nota68 + nota69 + nota70 + nota71 + nota72 +
nota73 + nota74 + nota75 + nota76 + nota77 + nota78 + nota79 + nota80 + nota81 +
nota82 + nota83 + nota84 + nota85 + nota86 + nota87 + nota88 + nota89 + nota90 +
nota91 + nota92 + nota93 + nota94 + nota95 + nota96 + nota97 + nota98 + nota99 +
nota100)/100;
```

E para apresentar as notas temos 100 instruções do género:

```
printf("%d\n",nota1);
```

Como facilmente se pode perceber um programa escrito desta forma torna-se impraticável. E já agora outra questão: que alterações teria de fazer a este programa de modo a ser possível ter um número variável de alunos?

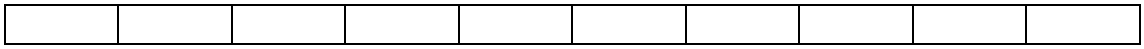
Pensemos então o seguinte: as 10 notas são valores inteiros e o conjunto das notas formam um todo – as notas da cadeira. Será que não há uma forma de considerá-las como uma só variável, com 10 componentes do mesmo tipo ? Há. São as **tabelas**.

Em **C** a declaração de uma variável do tipo tabela tem a seguinte sintaxe:

```
tipo_dos_elementos identif_tabela[número_elementos];
```

No Exemplo 5.1 precisamos então de declarar uma tabela de 10 inteiros **int notas[10];** Esta instrução corresponde à criação de uma tabela cujo identificador é **notas**, que possui **10** elementos e onde cada elemento é do tipo int. Podemos visualizar esta variável da seguinte forma:

notas



0 1 2 3 4 5 6 7 8 9

O primeiro elemento de uma tabela em **C** tem sempre índice **0**. Para se aceder a um determinado elemento da tabela, tem de se colocar o índice entre parênteses rectos, à frente do nome da tabela. Por exemplo **notas[0]** é o primeiro elemento da tabela e **notas[9]** o último.

Utilizando o conceito de tabela podemos então rescrever o programa anterior. Observe que a utilização de ciclos é um aspecto **essencial** na manipulação de tabelas.

Exemplo 5.2: Programa que rescreve o Exemplo 5.1 utilizando tabelas.

```
#include <stdio.h>

void main(void)
{
  int i, soma;
/* declaração da tabela */
  int notas[10];
  float media;

/* vamos utilizar um ciclo para ler as 10 notas */
  for(i = 0; i < 10; i++){
    printf("Introduza nota ");
    scanf("%d", &notas[i]);
/* notas[i] é o elemento com índice i da tabela, e &notas[i] é o seu endereço */
  }

  soma = 0;
  for(i = 0; i < 10; i++)
    soma = soma + notas[i];
  media = (float)soma/10;
  printf("Media das notas =%6.2f\n", media);

  if (media < 12.0){
    printf("Notas dos alunos\n");
    for(i = 0; i < 10; i++)
      printf("%d\n", notas[i]);
  }
}
```

Este programa pode facilmente ser alterado de modo a permitir um número variável de alunos. Tenha atenção aos comentários de modo a perceber as alterações introduzidas.

Exemplo 5.3: Programa que rescreve o Exemplo 5.2 de forma a flexibilizar o número de alunos a frequentar a cadeira.

```
#include <stdio.h>

void main(void)
{
/* numero máximo de alunos */
const int NMAX = 100;
int n_alunos;
int i, soma;

/* O tamanho de uma tabela não pode variar na execução de um programa, logo NMAX
tem de ser uma constante */
int notas[NMAX];
float media;

/* garante que o número de elementos é superior a zero e inferior ao número máximo
permitido, que é NMAX */

do {
  printf("Quantos alunos ? (1..100) ");
  scanf("%d", &n_alunos);
} while (n_alunos < 0 || n_alunos > NMAX);

/* repare que o índice varia entre 0 e n_alunos-1 */
for(i = 0; i < n_alunos; i++){
  printf("Introduza nota ");
  scanf("%d", &notas[i]);
}
soma = 0;
for(i = 0; i < n_alunos; i++)
  soma = soma + notas[i];

/* a soma das notas tem de ser dividida pelo número de alunos */
media = (float)soma / n_alunos;
printf("Media das notas =%6.2f\n", media);

if (media < 12.0){
  printf("Notas dos alunos\n");
  for(i = 0; i < n_alunos; i++)
  printf("%d\n", notas[i]);
}
}
```

Em alguns compiladores obtém-se um erro de compilação com o modo como foi declarada a constante NMAX. Um modo de contornar este problema é declarar NMAX utilizando a directiva **#define**, do seguinte modo:

```
#define NMAX 100
```

Até aqui vimos apenas tabelas cujos elementos são do tipo **int**, mas os elementos de tabelas podem ser de qualquer tipo anteriormente definido.

Exemplos de tabelas cujos elementos são de diferentes tipos de dados:

```
/* tabela de 100 números reais em vírgula flutuante */
float linha_real[100];

/* tabela de 25 caracteres */
char palavras[25];

/* tabela de 10 inteiros longos sem sinal */
unsigned long int positivos[10];
```

As declarações que vimos anteriormente correspondem a **tabelas unidimensionais** ou **vectores**, que têm a sintaxe:

```
tipo_dos_elementos identif_tabela[numero_elementos];
```

Se pretendermos definir **tabelas bidimensionais** ou **matrizes** então a sintaxe é:

```
tipo_dos_elementos identif_tabela[num_elem_dim1][num_elem_dim2];
```

Exemplo:

```
float matrizA[3][4];
```

Visualização da variável matrizA:

matrizA	0	1	2	3
0				
1				
2				

O índice de cada dimensão começa sempre em **0**. Os elementos são acedidos utilizando o identificador da variável e os dois índices.

Para além das tabelas com uma e duas dimensões também podemos declarar tabelas com três ou mais dimensões – **tabelas multidimensionais**. De modo a exemplificar as utilização tabelas bidimensionais o Exemplo 5.4 mostra parte de um programa para somar matrizes, com dimensões 10x5.

Exemplo 5.4: Parte de um programa para somar duas tabelas de dimensão 10x5.

```
#include <stdio.h>

void main(void)
{
float matA[10][5], matB[10][5], soma[10][5];
int i, j;

/*Inserir aqui as instruções para leitura das matrizes*/

/* somar as matrizes; é necessário um ciclo dentro doutro para aceder a todos os
elementos das matrizes */
for(i = 0; i < 10; i++)
  for(j = 0; i < 5; j++)
    soma[i][j] = matA[i][j] + matB[i][j];

/*Inserir aqui as instruções para apresentação do resultado*/
}
```

5.1.1 Inicialização de Tabelas

Quando da declaração de uma variável do tipo tabela, é possível inicializar os componentes dessa tabela com determinados valores, utilizando para isso uma lista de valores separados por vírgulas e delimitadas por chavetas.

Sintaxe:

```
tipo_elementos identif_tabela[número_elementos] =
                                {lista_de_valores_separados_por_virgulas};
```

Exemplo:

```
int pot2[10] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512};
int bases[] = {2, 8, 10, 16};
```

Visualização das variáveis:

pot2

1	2	4	8	16	32	64	128	256	512
0	1	2	3	4	5	6	7	8	9

bases

2	8	10	16
0	1	2	3

Como se viu, a dimensão de uma tabela é facultativo quando uma tabela é inicializada logo na sua declaração. Quando não se define o número de elementos, o compilador reserva espaço para armazenar todos os valores presentes na lista, logo a tabela **bases** vai ser uma tabela de 4 inteiros. Quando se define o número de elementos da tabela e só se inicializam alguns elementos da tabela, então os elementos não inicializados na lista de valores são zero, como se vê no exemplo:

```
int pot[10] = {1, 2, 4, 8};
```

pot

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

No caso de inicialização de tabelas de dimensão superior a 1 a atribuição é feita por linhas.

Exemplo:

```
int tabx[2][3]={1, 2, 3, 4, 5, 6};
```

tabx	0	1	2
0	1	2	3
1	4	5	6

Uma indexação diferente permitiria uma melhor visualização :

```
int tab[2][3]={
                1, 2, 3,
                4, 5, 6  };
```

Numa tabela de 2 dimensões é possível omitir a primeira dimensão, mas não a segunda. A segunda dimensão é necessária, pois o número de colunas só pode ser determinado deste modo. Assim,

```
int tabel[][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

é uma tabela de 3 linhas por 4 colunas

tabel	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Numa tabela de 2 ou mais dimensões também é possível inicializar apenas alguns elementos, sendo os elementos não especificados inicializados com zero. Vamos ver um exemplo:

```
int tab[3][4]={{1, 2}, {3, 4}, {5, 6}};
```

tab	0	1	2	3
0	1	2	0	0
1	3	4	0	0
2	5	6	0	0

5.1.2 Manipulação de Tabelas

Suponha que possui as seguintes declarações:

```
int linha[5];
int nums[5] = {2, 4, 8, 16, 32};
```

que declaram duas tabelas de 5 elementos do tipo **int**, encontrando-se a segunda inicializada. Se eu pretender que os elementos de linha sejam iguais aos de nums, poderia pensar que a seguinte atribuição me resolveria o problema:

```
linha = nums;
```

Mas não, pois obteria um erro de compilação, visto não ser possível fazer a atribuição deste modo. Em **C** para atribuir um conjunto de valores guardados numa tabela a outra tabela é necessário aceder individualmente a cada elemento. Assim para fazer a cópia de nums para linha vou utilizar um ciclo, como se pode ver a seguir:

```
for(i = 0; i < 5; i++)
    linha[i] = nums[i];
```

5.1.3 Métodos de Ordenamento e Pesquisa

Neste subcapítulo descrevem-se os Métodos de Ordenamento e Pesquisa. A exposição do funcionamento dos métodos é acompanhada do respectivo algoritmo. A escolha dos algoritmos apresentados recaiu sobre o mais explícito em detrimento do mais optimizado, de forma a tornar mais fácil o entendimento do método em questão.

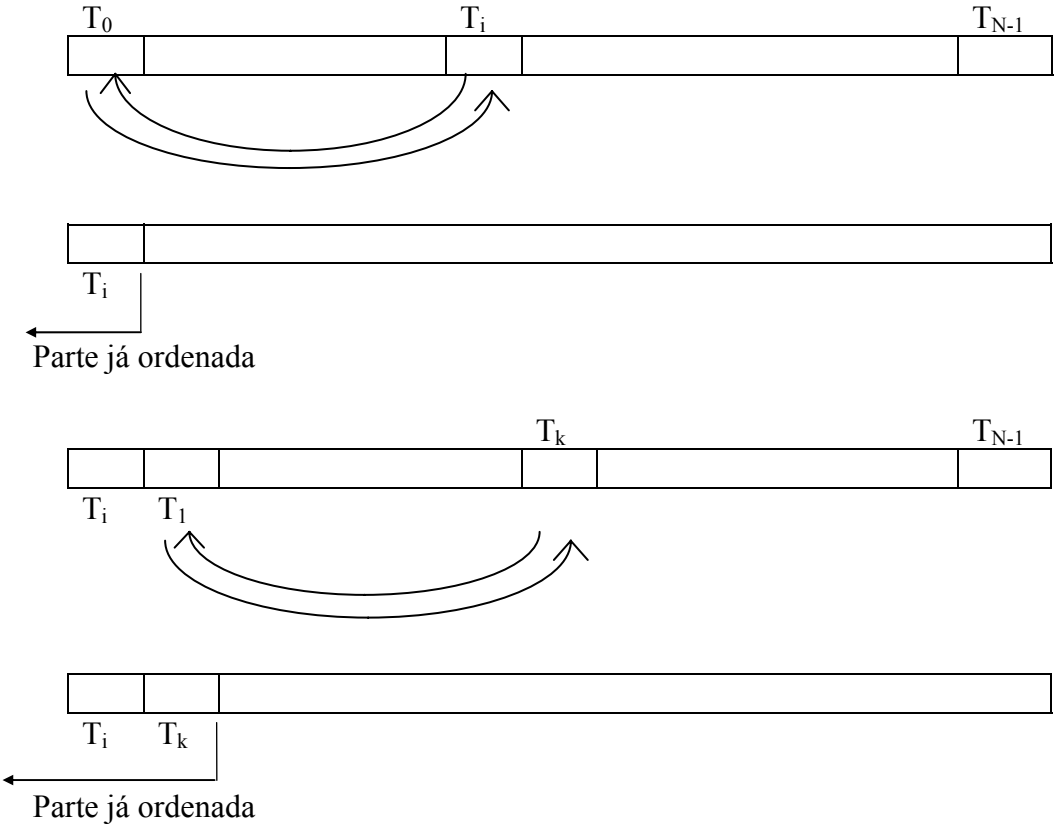
Embora os algoritmos sejam apresentados para ordenar e pesquisar tabelas unidimensionais (vectores) de elementos inteiros eles também são aplicáveis a tabelas em que os elementos são números reais, caracteres ou mesmo estruturas e a ficheiros. No caso dos elementos da tabela ou do ficheiro serem do tipo estrutura a ordenação deve ser feita por um dos campos.

5.1.3.1 Ordenamento em Tabelas

A ordenação consiste em colocar os elementos de um conjunto de acordo com um determinado critério ou ordem. Nestes apontamentos a ordenação vai ser efectuada por ordem crescente.

5.1.3.1.1 Ordenamento por Selecção (*Selection Sort*)

Na tabela a ordenar escolhe-se o elemento mais pequeno T_i e troca-se com o primeiro elemento da tabela. Nesta altura o primeiro elemento já se encontra na posição correcta. A seguir, escolhe-se o elemento menor de entre os elementos nas posições **1** a **N-1** (não esquecer que os índices estão compreendidos entre **0** e **N-1**) e troca-se com o segundo elemento da tabela, e assim sucessivamente até toda a tabela se encontrar ordenada.



Consideremos a tabela **T** preenchida com os elementos {2, 18, 1, 10} e vejamos passo a passo as alterações a que **T** é sujeita quando o algoritmo **Ordenamento por Selecção** é aplicado.

Procurar menor elemento (1) e colocá-lo na 1ª posição	{2, 18, **1**, 10}
O menor elemento (2) da tabela não ordenada troca com a 2ª posição	{1, 18, **2**, 10}
O menor elemento (10) da tabela não ordenada troca com a 3ª posição	{1, 2, **18**, 10}
Termina o processo, tabela já ordenada	{1, 2, 10, 18}

Algoritmo OrdSelec (Algoritmo Genérico)
Objectivo: ordenar uma tabela **T**, com **N** elementos.

Início
$i \leftarrow 0$
Enquanto $(i < N)$ **Faz**
Calcula mínimo (T_{minimo}) *entre* T_i *e* T_{N-1}
Troca T_i *com* T_{minimo}
$i \leftarrow i + 1$
Fimfaz
Fim

Para implementar o algoritmo vamos precisar de uma função para calcular o menor elemento **CalcPosMin** presente numa tabela e de outra para trocar dois elementos de uma tabela **Troca**. Os seus algoritmos são:

Algoritmo CalcPosMin

Objectivo: obter e retornar a posição do menor valor presente na tabela **T**, com índice inicial **i** e final **N-1**.

Início
$temp \leftarrow i$
Enquanto $(i < N-1)$ **Faz**
Se $(T_{i+1} < T_{temp})$ **Então Faz**
$temp \leftarrow i+1$
Fimfaz
$i \leftarrow i+1$
Fimfaz
Retorna *temp*
Fim

Algoritmo Troca

Objectivo: trocar os valores das posições **i** e **pos** da tabela **T.**

Início
$temp \leftarrow T_i$
$T_i \leftarrow T_{pos}$
$T_{pos} \leftarrow temp$
Fim

5.1.3.1.2 Ordenamento por Borbulhamento (*Bubble Sort*)

No método de **Ordenamento por Borbulhamento** começa-se por percorrer a tabela comparando elementos adjacentes (o primeiro com o segundo, o segundo com o terceiro, e assim sucessivamente). Sempre que forem encontrados dois elementos cuja ordem está invertida, trocam-se. No fim de ter sido percorrida a tabela uma vez, temos o último elemento ordenado. Vamos então percorrer a tabela entre o primeiro e o penúltimo elemento do mesmo modo, e no fim desta segunda passagem temos os dois últimos elementos ordenados. De seguida, vamos proceder do mesmo modo entre o primeiro e o antepenúltimo, e assim sucessivamente até toda a tabela se encontrar ordenada. O processo de ordenamento termina quando ao ser efectuada uma passagem completa não ocorreu nenhuma troca.

Consideremos a tabela **T** preenchida com os elementos {3, 1, 9, 6, 7} e vejamos passo a passo as alterações a que **T** é sujeita quando o algoritmo **Ordenamento por Borbulhamento** é aplicado.

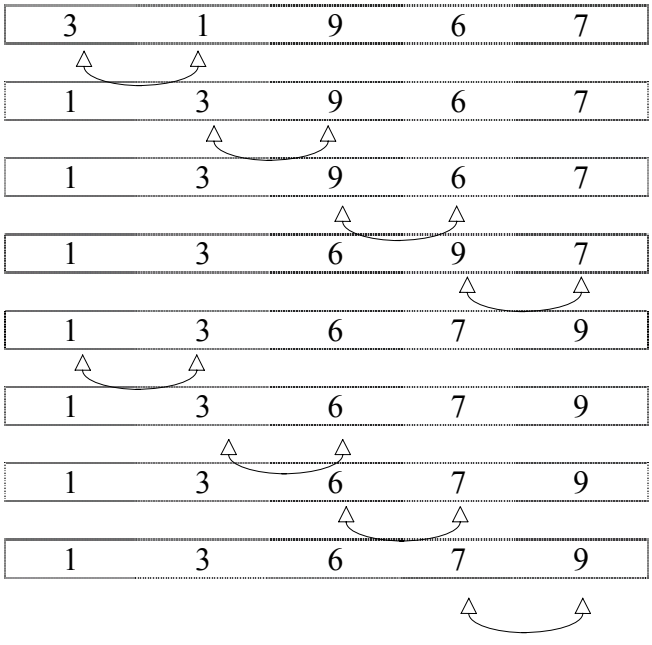
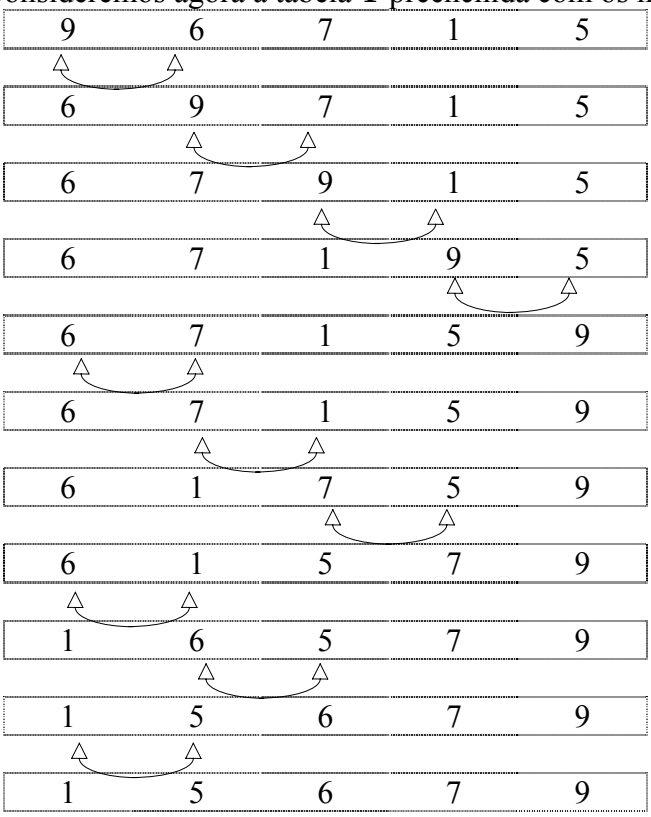


Tabela Desordenada
1ª Passagem

Fim da **1ª** passagem
2ª Passagem

Não houve trocas - **Tabela Ordenada**

Consideremos agora a tabela **T** preenchida com os números {9, 6, 7, 1, 5}



O **Ordenamento por Borbulhamento** pode ser traduzido pelo seguinte algoritmo:

Algoritmo Ord_Bolhas (Algoritmo Genérico)

Objectivo: ordenar a tabela **T** de **N** elementos.

Início

i ← 0

/ Se numa passagem pela tabela não for necessário fazer nenhuma permuta então é porque a tabela já se encontra ordenada, esta situação é indicada por ordenada = 1 */*

ordenada ← 0

Enquanto *(i< N)* **E** *(não ordenada)* **Faz**

Bolhas(N, i, T, ordenada)

i ← i+1

Fimfaz

Fim

Algoritmo Bolhas (correspondente a uma passagem pela tabela)

Objectivo: faz uma passagem pela subtabela de **N-k** elementos e promove as trocas necessárias.

Início

$j \leftarrow 0$

$orden \leftarrow 1$

Enquanto $j < (N-k)$ **Faz**

Se $(T_{j+1} < T_j)$ **Então Faz**

$orden \leftarrow 0$

$temp \leftarrow T_{j+1}$

$T_{j+1} \leftarrow T_j$

$T_j \leftarrow temp$

Podia-se utilizar a função `Troca(j,j+1,T)`

Fimfaz

$j \leftarrow j+1$

Fimfaz

Fim

5.1.3.1.3 Ordenamento por Inserção

Consideremos que temos uma sequência de elementos de uma tabela T_0, …, T_{N-1}, e que $T_0 \leq T_1 \leq \ldots \leq T_{N-1}$. Inserindo um novo elemento **X**, a sequência de elementos deve manter-se ordenada.

Algoritmo InsereOrdem

Objectivo: inserir um elemento **X** na tabela ordenadamente, estando esta já ordenada por ordem crescente, onde **N** é o número de elementos ordenados inicialmente.

Início

$j \leftarrow N-1$

Enquanto $(X < T_j)$ **Faz**

$T_{j+1} \leftarrow T_j$ (move T_j para a posição superior uma casa, **X** terá de ser inserida abaixo de T_j)

$j \leftarrow j-1$

Fimfaz

$T_{j+1} \leftarrow X$

Fim

5.1.3.2 Métodos de Pesquisa em Tabela

A pesquisa de informação é uma tarefa importante e usada frequentemente em todos os sistemas de informação. Assim, apresentam-se dois métodos de pesquisa e efectua-se uma comparação em termos de rapidez dos métodos referidos.

5.1.3.2.1 Pesquisa Sequencial ou Linear

No método de **Pesquisa Sequencial** o elemento **X** a procurar é comparado sequencialmente com todos os elementos da tabela **T**, até ser encontrado o elemento procurado ou termos chegado ao fim da tabela. No melhor dos casos só é necessária uma comparação e no pior dos casos **N** comparações. Em média necessitamos de N**/2** comparações do tipo **X**!=T_i. É de realçar que na realidade são necessárias duas comparações para cada valor de **i** (a segunda é para garantir que estamos a testar elementos válidos), pelo que em média necessitamos de N/2*2=**N comparações** para encontrarmos o valor pretendido.

Algoritmo PesqSeq

Objectivo: procurar um elemento **X,** sequencialmente, na tabela **T** de **N** elementos. É devolvida a posição do elemento se este existir, ou o valor –1 em caso contrário.

Início

```
    i ← 0
    Enquanto (i < N) E (X ≠ Ti) Faz
      i ← i +1
    Fimfaz
    Se (X = Ti ) Então Faz
            Devolve i
    Senão Faz
            Devolve -1
Fim
```

5.1.3.2.2 Pesquisa Linear com Sentinela

Existe uma implementação alternativa ao método de **Pesquisa Sequencial**, que contém uma pequena, mas significativa alteração, em que só são necessárias **N/2** comparações e não **N**. Consiste na colocação de uma "sentinela" na tabela. Para tal, colocamos na posição

de índice **N** o elemento a procurar. Desta forma não é necessária a segunda comparação, e o algoritmo pode ser rescrito da seguinte forma:

Algoritmo PesqSeqComSentinela

Objectivo: procurar um elemento **X,** na tabela **T** de **N** elementos. É devolvido a posição do elemento se este existir, ou o valor –1 em caso contrário.

Início

$i \leftarrow 0$

$T_N \leftarrow X$

Enquanto $(X \neq T_i)$ **Faz**

i ← i +1

Fimfaz

Se $(i = N)$ **Então Faz**

Devolve *i*

Senão Faz

Devolve *-1*

Fim

5.1.3.2.3 Pesquisa Binária

No método de **Pesquisa Binária** a procura do número é efectuada sucessivamente em intervalos cada vez mais pequenos, reduzindo-se em cada passagem o intervalo de pesquisa a metade. Em primeiro lugar a tabela é dividida ao meio. A seguir, verifica-se em qual das metades pode estar o valor procurado. Na metade em que isso for possível - a primeira metade da tabela é limitada por T_i e T_j e a segunda por T_{j+1} e T_k - procura-se na primeira metade se $T_i < X < T_j$ e na segunda no caso contrário - vai-se novamente dividir ao meio e assim sucessivamente até encontrar o valor pretendido ou até não se poder subdividir mais.

Neste método é necessário que a tabela se encontre ordenada, contrariamente ao que sucede no caso da pesquisa sequencial. Uma função que indica se um determinado número existe ou não numa tabela ordenada tem o algoritmo seguinte:

Algoritmo PesquisaBinária

Objectivo: indica se um número **X** existe ou não na tabela **T**, de **N** elementos

Inicio

Inf ← *0*
Sup ← *N-1*
Enquanto *(Inf* ≠ *Sup)* **E** *(T_{Inf}* ≠ *X)* **E** *(T_{Sup}* ≠ *X)* **Faz**

Novo ← *(Inf+Sup) / 2* *Atenção: divisão inteira*

*Se T_{Novo}> **X** então faz*

Sup ← *Novo*

Inf ← *Inf+1*

Fimfaz
Senao Faz
Inf ← *Novo*

Sup ← *Sup-1*

Fimfaz

Fimfaz

Se *(T_{Inf} = X)* **OU** *(T_{Sup} = X)* **Então Faz**

Retorna 1

Fimfaz
Senão Faz
Retorna 0

Fimfaz
Fim

Este método de pesquisa reduz para metade o número de elementos a considerar sempre que efectuamos uma comparação. Assim, a pesquisa binária não exige mais que $(1+\log_2 N)$ passagens. Em cada passagem fazemos 3 comparações é inferior a $3*(1+\log_2 N)$. Podemos então comparar os dois métodos de pesquisa referidos para tabelas de várias dimensões.

Note-se que para valores elevados a diferença entre os dois métodos é abissal, com vantagem clara para o **Método de Pesquisa Binária**. A desvantagem deste método reside no facto de necessitar que a tabela esteja ordenada, o que implica que o processo de inserção seja computacionalmente mais lento. Esta desvantagem esbate-se completamente, pois para além dos resultados apresentados no quadro anterior, a frequência com que se efectuam pesquisas é muito superior à da inserção de novos elementos.

Número Elementos da Tabela	Número Médio de Comparações Pesquisa sequencial com Sentinela (N/2)	Número Máximo de Comparações Pesquisa binária $3*(1+\log_2 N)$
64	32	21
128	64	24
512	256	30
1024	512	33
1048576	524288	63

5.2 Strings (Cadeias de Caracteres)

Em **C** uma *string* é uma tabela de caracteres convenientemente terminada pelo carácter '\0', o carácter nulo (NULL *character*). Note-se que o carácter nulo '\0' (posição **0** na tabela ASCII) é diferente de '0' (carácter zero, posição 48 na tabela ASCII).

Sintaxe:

```
char ident_cadeia[num_elementos];
```

Exemplo:

```
char frase[100];
char morada[50], nome[10], let;
```

É possível declarar na mesma linha *strings* e variáveis do tipo carácter (veja-se por exemplo a segunda linha do exemplo anterior). Se na variável **nome** estivesse armazenado "Certo", então em **nome** está:

nome

'C'	'e'	'r'	't'	'o'	'\0'				
0	1	2	3	4	5	6	7	8	9

Assim, para armazenar uma *string* de 6 caracteres são necessários 7 posições (7 *bytes*), pois o **C** utiliza o carácter '\0' para indicar o fim de uma string. Genericamente, para

armazenar uma *string* de **n** caracteres é necessária uma tabela de **n+1** elementos (pois é necessário adicionar o carácter nulo).

5.2.1 Inicialização de *String*

Como uma *string* não é mais do que uma tabela de caracteres (convenientemente terminada), a sua inicialização pode ser feita de um modo idêntico à inicialização de tabelas, ou seja, utilizando uma lista de valores entre chavetas e separados por vírgulas.

```
char vogais1[] = {'a', 'e', 'i', 'o', 'u', '\0'};
```

Neste caso, e como não está definido o número de elementos, estou a declarar um tabela de 6 elementos do tipo carácter, pois na lista de valores encontram-se 5 vogais e o carácter nulo.

Em **C** existe um modo alternativo, e mais simples, de inicializar *strings*, que consiste em colocar o conjunto de caracteres entre aspas, como se pode ver a seguir:

```
char vogais2[] = "aeiou";
```

onde "aeiou" constitui aquilo que se chama uma ***string* constante**. O carácter nulo é colocado automaticamente pelo compilador na fim da *string*. Se numa *string* não forem inicializados todos os elementos, então nos elementos não inicializados é colocado o carácter nulo.
Assim, as seguintes declarações e respectivas inicializações

```
char vogais1[] = {'a', 'e', 'i', 'o', 'u', '\0'};
char vogais2[] = "aeiou";
char vogais3[10] = "aeiou";
```

têm a seguinte representação em memória

vogais1

'a'	'e'	'i'	'o'	'u'	'\0'
0	1	2	3	4	5

vogais2

'a'	'e'	'i'	'o'	'u'	'\0'
0	1	2	3	4	5

vogais3

'C'	'e'	'r'	't'	'o'	'\0'	'\0'	'\0'	'\0'	'\0'
0	1	2	3	4	5	6	7	8	9

Para colocar uma nova *string* em **nome** é necessário utilizar a função **strcpy**. Esta tem dois parâmetros sendo o primeiro a *string* destino e o segundo a *string* origem. Por exemplo,

```
char nome[10];
strcpy(nome, "Aula");
/* neste caso a string origem é uma string constante */
```

nome

'A'	'u'	'l'	'a'	'\0'					
0	1	2	3	4	5	6	7	8	9

É possível, de acordo com o anteriormente explicado, inicializar uma *string*, como por exemplo em

```
char morada[20] = "Rua Y - Coimbra";
```

no entanto, a instrução de **atribuição**

```
morada = "Rua Y - Coimbra";
```

é **inválida,** dando origem a um erro de compilação. A explicação do sucedido tornar-se-á claro quando do estudo de apontadores.

5.2.2 Leitura e Escrita de *Strings*

A função **scanf** utilizada para leitura de cadeias de caracteres, com o especificador de formato "**%s**" só lê até encontrar um espaço, um *tab* ou uma mudança de linha (referidos na literatura inglesa como *white-space characters*), omitindo também todos os *white-spaces* que se encontram no início de uma linha. Assim se tivermos o seguinte fragmento de código,

```
char frase[100];
scanf("%s", frase);
```

e o utilizador escrever a frase "`Certo ou errado ?`" o que fica armazenado na *string* frase é `Certo` e não `Certo ou errado` ? como pretendíamos. Para resolver este problema existe a função **gets()**. Esta lê uma *string* a partir do teclado, sendo a *string* o parâmetro da função. Se escrevermos o código

```
char frase [100];
gets(frase);
```

e dermos entrada ao mesmo texto, então em frase fica armazenado

```
Certo ou errado ?
```

Para colocar uma *string* no monitor utilizamos a função **printf** com o especificador de formato "**%s**". Um modo alternativo de colocar uma *string* no monitor é utilizar a função **puts()**. Esta tem como parâmetro uma string, coloca-a no monitor e muda de linha. Assim a linha de código

puts(frase); é equivalente a printf("%s\n", frase);

Se a *string* nome, declarada como char nome[10], fosse inicializada com a *string* constante "Certo", e depois colocado o carácter nulo na posição 3 temos (ver código seguinte):

```
/* apenas fragmento do código */
char nome[10] = "Certo"; /* A */
printf ("%s\n", nome);

nome[3] = '\0'; /* B */
printf ("%s\n", nome);
```

obtemos

```
Certo
Cer
_
```

O que demonstra a importância do carácter nulo na especificação de uma *string*. Conteúdo de nome

No ponto *A*

nome

'C'	'e'	'r'	't'	'o'	'\0'				
0	1	2	3	4	5	6	7	8	9

No ponto *B*

'C'	'e'	'r'	'\0'	'o'	'\0'				
0	1	2	3	4	5	6	7	8	9

5.2.3 Algumas Funções Definidas para *Strings*

As funções seguintes, expostas de uma forma simples que corresponde à sua utilização mais usual encontram-se definidas na biblioteca restantes em **string.h**, à excepção das funções **gets()** e **puts()** que se encontram em **stdio.h**.

strlen(str)	Devolve o comprimento da string str
strcpy(dest, orig)	copia string orig para dest
strcat(dest, orig)	copia string orig no fim de dest
strcmp(str1, str2)	compara str1 com str2 e retorna: um valor negativo se str1<str2; zero se as strings forem iguais; um valor positivo se str1 > str2
strupr(str)	converte os caracteres de str para maiúsculas
strlwr(str)	converte os caracteres de str para minúsculas
gets(str)	lê string via teclado
puts(str)	escreve string no monitor e muda de linha

Exemplo 5.5: Programa que exemplifica a utilização de algumas das funções referidas.

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

void main(void)
{
char letras[30], nome[100];
char frase[30] = "Linguagem C", letras2[40];

  strcpy(letras, "AbcDEfg");
  strcpy(nome, letras);
  strcat(letras, "***");

  printf("tres strings\n%s\n%s\n%s\n", frase, letras, nome);
  printf("cujos comprimentos sao %d, %d e %d\n", strlen(frase), strlen(letras),
strlen(nome));

  strcat(frase, " - 1º ano");
  printf("Frase aumentada \n%s\nde comprimento %d\n", frase,
      strlen(frase));
  frase[10] = '\0';
  printf("Frase é agora %s e tem comprimento %d\n", frase, strlen(frase));
  strupr(letras);
  strcpy(letras2, letras);
  strlwr(letras2);
  printf("uma string em maiusculas e outra em minusculas \n%s\n%s", letras, letras2);
}
```

Resultado no monitor:

```
tres strings
Linguagem C
AbcDEfg***
AbcDEfg
cujos comprimentos sao 11, 10 e 7
Frase aumentada
Linguagem C - 1º ano
de comprimento 20
Frase é agora Linguagem  e tem comprimento 10
uma string em maiusculas e outra em minusculas
ABCDEFG***
abcdefg***_
```

5.2.4 Conversão entre Tipos Numéricos e *Strings* e Vice-Versa

Por vezes é necessário fazer a conversão entre tipos numéricos (inteiros ou reais em vírgula flutuante) e *strings* e vice-versa. Para isso o **C** define um conjunto de funções que permite fazer este género de conversões.

atoi(str)	*string* para int; retorna um número
atof(str)	*string* para float; retorna um número
itoa(valor, str, base)	int para *string*

Exemplo 5.6: Programa que exemplifica a utilização de algumas funções de conversão de tipo.

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

void main(void)
{
  char numero[10]="30";
  char quad[10];
  int n;
  float x;

  n = atoi(numero);
  x = atof(numero);

  printf("numero ao quadrado = %d\n", n*n);
  printf("numero ao cubo     = %.2f\n", x*x*x);

  itoa(n, quad, 10);
  printf("n = %s\n", quad);
}
```

6. Apontadores

Uma variável apontador ou ponteiro (*pointer*) permite armazenar um endereço de memória. O operador endereço **&** permite obter o endereço de uma variável. Assim, se tivermos o código seguinte:

```
int num; /* uma variável do tipo int ocupa 4 bytes */
char let = 'Z'; /* uma variável do tipo char ocupa 1 byte */
num = 10;
```

em memória pode ser ilustrado da seguinte forma (estamos a supor os endereços das variáveis)

num	3096	10
	3097	
	3098	
	3099	
let	4000	'Z'
	4001	
	4002	

onde

&num é 3096 e &let é 4000.

6.1 Declaração de um Apontador

Para declarar um apontador é necessário indicar o tipo base (tipo para o qual o apontador aponta) e depois coloca-se o identificador do apontador precedido por um asterisco. Genericamente temos:

```
tipo_base *identif_apontador;
```

Exemplos:

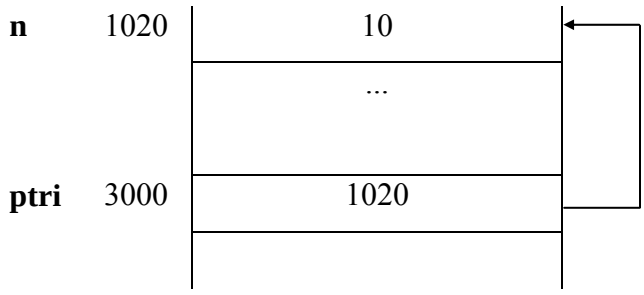
```
int   *ptri;      /* ptri é um apontador para int    */
char  *ptrc;      /* ptrc é um apontador para char */
float *ptrf;      /* ptrf é um apontador para float */
```

6.2 O operador de Indirecção

Podemos aceder ao conteúdo de uma zona de memória apontada por um apontador utilizando o **operador de indirecção *** ou **operador apontado por**. Assim, um apontador permite referenciar indirectamente uma zona de memória. Para tal é necessário que se coloque o apontador a apontar para essa zona, atribuindo-lhe o respectivo endereço.

Exemplo 6.1: Programa que ilustra o acesso a uma variável através de um apontador.

```
#include <stdio.h>

void main (void)
{
 int n;
 int *ptri;
 ptri = &n; /* diz-se que ptri aponta para n */
 *ptri = 10; /* a variável apontada por ptri toma o valor 10 o que equivale a  n=10
 n = *ptri + 20;
 /* n é agora 30*/
}
```



Sempre que uma variável apontador é utilizada numa expressão precedida do operador (unário) ***** então estamos a aceder ao conteúdo da zona apontada por esse apontador. A uma variável é atribuído um endereço fixo que não é possível alterar durante a execução do programa. Consideremos a declaração

```
int num;
```

não podemos fazer

```
&num = 1048; /* ou qualquer outro valor */
```

Se pretendermos apresentar um endereço utilizando a função **printf**, podemos utilizar o especificador de formato **%p** ou então o **%u**. Com **%p** o endereço é indicado no formato hexadecimal e com **%u** em formato decimal (base 10).

Exemplo 6.2: Programa que acede ao conteúdo de uma variável através de um apontador.

```
#include <stdio.h>

void main(void)
{
 int num, *ptr;
 num = 323;

 pt = &num;
 printf("O valor é %d e encontra-se no endereço %u \n", num, &num);
 printf("O valor é %d e encontra-se no endereço %u \n", *ptr, ptr);
}
```

A um apontador para tipo_base só lhe pode ser atribuído o endereço de uma variável do tipo_base. Assim o fragmento de código seguinte não está correcto:

```
int *ptr; /* ptr é um apontador para int */
float num;
ptr = &num; /* errado, pois ptr só deve apontar para variáveis do tipo int */
```

A inicialização de um apontador deve merecer a maior atenção por parte do programador. Um apontador só deve ser utilizado depois de conter um endereço de memória válido, isto é, depois de ser inicializado correctamente.

```
int *ptr; /* ptr é um apontador para int */
int num;
*ptr = 33; /* Errado. Não sei para onde ptr aponta, pelo que posso estar a escrever
sobre outras variáveis ou numa zona de memória protegida */
```

6.3 Aritmética de Apontadores

A variáveis do tipo apontador pode-se:

somar	+
subtrair	-
incrementar	++
decrementar	--

Só se pode somar ou subtrair a um apontador valores inteiros. Podem ainda subtrair-se apontadores, tendo como resultado um inteiro. Vamos considerar as declarações seguintes:

```
int    *ptri;
double *ptrd;
char   *ptrc;
```

e que os valores armazenados em ptri, ptrd e ptrc forem 1000, 2000 e 3000, respectivamente. Se fizermos

```
ptri++;
ptrd++;
ptrc++;
```

então os valores de **ptri**, **ptrd** e **ptrc** passam a ser, respectivamente, 1004, 2008 e 3001. Repare-se que todos os apontadores foram igualmente incrementados, mas o endereço armazenado pelos três apontadores foi incrementado de valores diferentes. Isto deve-se ao facto dos tipos apontados serem diferentes: **int** (4 *bytes*), **double** (8 *bytes*) e **char** (1 byte). Assim a incrementação ou decrementação de um apontador implica passar à posição de memória seguinte, partindo do princípio que o próximo "objecto" apontado é do mesmo tipo. Seguindo esta linha de raciocínio, e no seguimento do que foi dito, então as expressões **ptri+5** e **ptrd-1** são 1020 e 1992 respectivamente, se em **ptrd** e **ptri** estivessem armazenados os endereços 1000 e 2000.

A subtracção de dois apontadores tem como resultado um **int**, que corresponde ao número de elementos entre as duas posições de memória armazenadas nos apontadores.

Exemplo:

```
int *ptrx, *ptry;
int n_elem;
n_elem = ptrx-ptry;
```

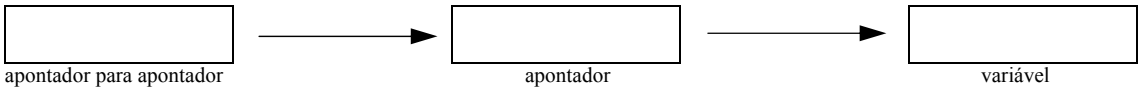
Se em **ptrx** estivesse o endereço 4000 e em **ptry** o endereço 4200 em **n_elem** fica 50, que corresponde a **(ptrx-ptry)/sizeof(int)**.

6.4 Apontador para void

Um apontador para void é um apontador genérico. É o tipo normalmente retornado pelas funções de alocação dinâmica de memória.

6.5 Apontador para Apontador

É possível em C ter um apontador que aponta para outro apontador. A isto chama-se **indirecção múltipla**. Quando um apontador aponta para outro apontador, o primeiro apontador contém o endereço do segundo apontador, o qual por sua vez aponta para a zona de memória da variável.



Declaração:

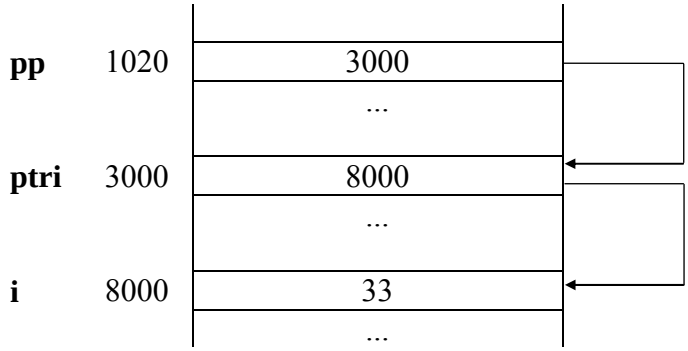
```
int **pp;
```

pp não é um apontador para **int**, mas sim um apontador para um apontador para **int**. Para aceder à variável é necessário utilizar o operador indirecção * duas vezes.

Exemplo:

```
int **pp, *ptri, i;
ptri = &i;
pp   = &ptri;   /* pp contém o endereço de outro apontador */
**pp = 33;      /* estou a aceder à variável i */
```

Podemos ilustrar em termos de memória:



6.6 Alocação Dinâmica de Memória

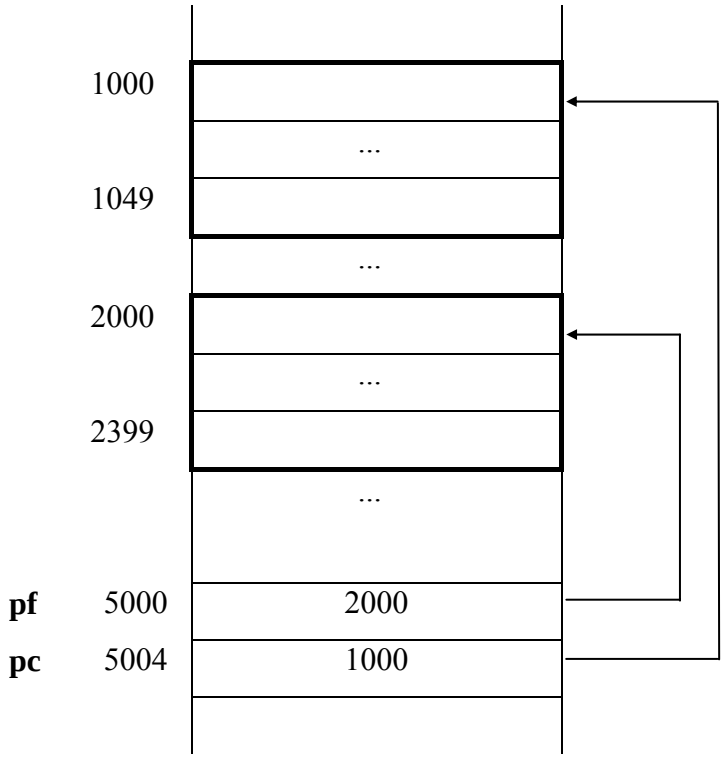
A função **malloc** *(dynamic memory allocation)* e permite alocar (reservar) um bloco memória na fase de execução de um programa, encontrando-se definida na biblioteca **stdlib.h** e tem como protótipo:

void *malloc(unsigned n*bytes*);

malloc devolve um apontador **void** que contém o endereço inicial do bloco de memória alocado. O tamanho em ***bytes*** desse bloco é n*bytes*. Se não houver memória suficiente então **malloc** retorna um apontador com o valor **0** (apontador nulo). É habitual usar em vez do valor **0** a constante simbólica **NULL**, está definida em **stdlib.h** (e em **stdio.h**) como **0**. Quando se utiliza esta função deve-se fazer a conversão de tipos (*cast*) do apontador genérico devolvido pela função para o tipo de dados adequado.

Exemplo:

```
char  *pc; /* pc é um apontador para char */
float *pf; /* pf é um apontador para float */

pc = (char *)malloc(50);/* pc aponta para um bloco de 50 bytes */
pf = (float *)malloc(100*sizeof(float));/* pf aponta para um bloco de 100*4  bytes */
```



Depois de uma chamada de **malloc** é conveniente verificar se o bloco de memória foi alocado com sucesso, bastando para isso testar o valor do apontador. Se o apontador retornado for o apontador **NULL** é comum terminar o programa.

Exemplo:

```
float *pf;

pf = (float *)malloc(100*sizeof(float));
/* pf aponta para um bloco de 400 bytes, o que permite armazenar 100 floats */
if (pf == NULL)
{
  printf("Nao ha memoria suficiente\n");
  exit(1);
}
```

A função **exit()** também se encontra declarada em **stdlib.h**. Esta termina (aborta) a execução de um programa e retorna o seu argumento ao sistema operativo - **0** para normal, **1** para não-normal. É imprudente prosseguir a execução de um programa sem testar o valor devolvido por **malloc.**

A função **calloc()** permite também alocar um bloco de memória na fase de execução de um programa, colocando o valor **0** em todo o bloco alocado. A função **calloc()** tem dois parâmetros: o número de componentes a alocar e o tamanho de cada componente. Esta função encontra-se definida em **stdlib.h**. Tal como na utilização de **malloc()** é conveniente testar o valor do apontador antes de prosseguir. O protótipo da função é:

```
void *calloc(unsigned nitems, unsigned size);
```

Exemplo:

```
char *pc;
float *pf;

pc = (char *)calloc(50, sizeof(char));
/* pc aponta para um bloco de 50 bytes */
if (pc == NULL){
  printf("Nao ha memoria suficiente\n");
  exit(1);
}
pf = (float *)calloc(1000, sizeof(float));
/* pf aponta para um bloco de 1000*4=4000 bytes */
if (pf == NULL){
  printf("Nao ha memoria suficiente\n");
  exit(1);
}
```

A função **realloc()** permite alterar o tamanho do bloco previamente alocado, isto sem alterar os valores que lá estavam guardados. Esta função tem como parâmetros o apontador para o bloco e o novo tamanho, retornando o apontador para o bloco com o novo tamanho. O seu protótipo é:

```
void *realloc(void *block, unsigned size);
```

Exemplo:

```
float *pf;

pf = (float )calloc(100, sizeof(float));
/*pf aponta para um bloco de 100 floats */
if (pf == NULL){
  printf("Não há memória suficiente\n");
  exit(1);
}
…
```

```
   /* e agora precisamos de espaço para 500 floats */
   pf = (float *)realloc(pf, 500 * sizeof(float));
   if (pf == NULL){
     printf("Não há memória suficiente \n");
     exit(1);
   }
…
```

A função **free()** permite libertar a memória anteriormente alocada por **malloc()**, **calloc()** ou ainda por **realloc()**. O parâmetro desta função é o apontador obtido pela chamada de uma função de alocação de memória. O seu protótipo é:

```
void free(void *block);
```

Esta função deve ser utilizada quando o bloco previamente alocado já não for necessário, libertando assim o espaço em memória que já não está a ser utilizado pelo nosso programa.

```
float *pf;
...
pf = (float )malloc(100);
... /* pf aponta para um bloco de 100 bytes, 25 floats*/
free(pf);
...
```

Resumo - protótipos das funções utilizadas para alocação dinâmica de memória em **C**:

```
void *malloc(unsigned nbytes);
void *calloc(unsigned nitems, unsigned size);
void *realloc(void *block, unsigned size);
void  free(void *block);
```

6.7 Apontadores e Tabelas

Na linguagem de programação **C** existe uma relação muito próxima entre tabelas e apontadores. É esta proximidade e a utilização de apontadores no acesso e manipulação de tabelas que estudaremos a seguir.

Consideremos as seguintes declarações:

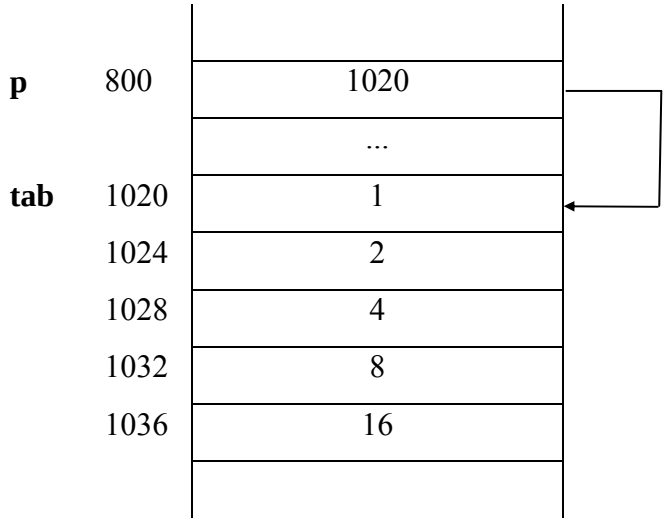
```
int tab[5] = {1, 2, 4, 8, 16};
int *p;
```

posso fazer

```
p = &tab[0];   /* p toma o valor do endereço do primeiro elemento da tabela tab*/
```

e posteriormente aceder à tabela **ta**b utilizando o apontador **p**.

O que eu estou a fazer é atribuir ao apontador **p** o endereço inicial da tabela **tab**. **Não há cópia de elementos.**



A linha de código **p = &tab[0];** também podia ter sido escrita **p = tab;** pois quando o identificador de uma tabela é usado numa expressão, corresponde ao endereço inicial da tabela. Este endereço é um endereço constante.

Há duas notações possíveis para aceder aos elementos de uma tabela utilizando apontadores

notação de *offset* e notação matricial:

***(p+i)** ≡ **p[i]**

Assim o endereço do elemento índice **i** é dado por

(p+i) ≡ **&p[i]**

Para exemplificar o conceito vamos analisar o Exemplo 6.3.

Exemplo 6.3: programa que copia o conteúdo de **pot2** para uma nova tabela alocada dinamicamente, e de seguida apresenta os valores copiados por ordem inversa.

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

void main(void){
  int i, pot2[10]={1, 2, 4, 8, 16, 32, 64, 128, 256, 512};
  int *p;
  p = (int *)calloc(10, sizeof(int));
  if (p==NULL){
    printf("Nao ha memoria suficiente\n");
    exit(1);
}
/***** Ponto A *******/
/* fazer a cópia dos valores para a zona de memória alocada;
 utiliza-se notação matricial */
for(i=0; i<10; i++){
  p[i]=pot2[i];
}
/* apresentar valores do fim para o princípio; utiliza notação de  offset */
for(i=9; i >= 0; i--){
    printf("tab[%d]=%d\n", i, *(p+i));
}
}
```

Repare-se que **não** é possível fazer **tab = p;** pois **tab** é um endereço constante, e como tal não pode ser alterado.

6.8 Apontadores como Argumentos de Funções

Existem duas formas de efectuar passagem de argumentos: por **valor** e por **referência**. As funções exemplificadas até este ponto utilizam passagem por **valor**, ou seja, os valores dos argumentos são calculados, independentemente de se tratar de uma variável ou expressão, e esse **valor é copiado** para a zona de memória reservada para o parâmetro correspondente no cabeçalho da função. Todas as operações efectuadas sobre os parâmetros são efectuadas sobre uma cópia, portanto todas as alterações a que este parâmetro seja sujeito dentro da função invocada **não alteram o valor do respectivo parâmetro**. Na passagem por **referência** o que é enviado à função, quando ela é invocada, é a própria variável ou uma

referência para ela. Portanto as alterações efectuadas nos parâmetros da função invocada correspondem a alterações efectuadas nos próprios argumentos.

Na **linguagem C apenas existe passagem de argumentos por valor**. A questão que se coloca é como pode uma função alterar o valor dos seus argumentos? Não sendo a passagem de parâmetros por referência intrínseca à linguagem **C** temos de recorrer a uma estratégia, que envolve a utilização de apontadores :

- o parâmetro deve ser do tipo apontador;
- o argumento na chamada da função deve ser o endereço da variável cujo conteúdo se pretende alterar;
- na função é utilizada a operação de indirecção (operador *****) para aceder à variável cujo endereço foi passado à função.

Embora seja comum chamar este tipo de passagem de argumentos por referência ou por endereço, na realidade não é. O que é enviado para a função são cópias dos endereços das variáveis, portanto continuamos a lidar com uma passagem por valor. Para ilustrar estes conceitos, vamos analisar o exemplo clássico de um programa que permite trocar o conteúdo de duas variáveis, com recurso a uma função.

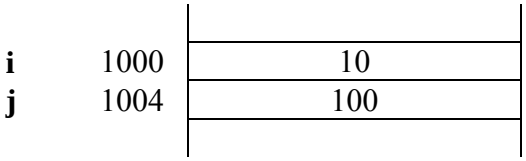
Exemplo 6.4: Programa que troca o conteúdo de duas variáveis, através de uma função.

```
#include <stdio.h>
/* os parâmetros da função são apontadores para int */
void troca(int *x, int *y)
{
int temp;

temp = x;   /* ponto A */
*x = *y;    /* ponto B o acesso é feito utilizando indirecção */
*y = temp;  /* ponto C */
return;
}
```

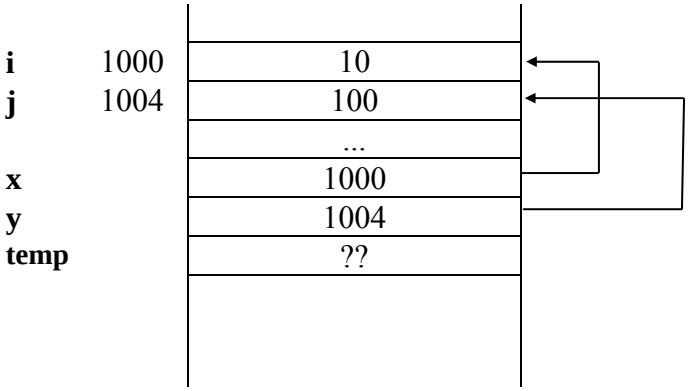
```
void main(void)
{
int i = 10, j = 100;
printf("%8c%8c\n",'i','j');
printf("Valores iniciais%8d%8d\n", i, j);

/* na chamada da função são passados os endereços das variáveis */
troca(&i, &j);
/* ponto D */
printf("Valores trocados%8d%8d\n", i, j);
}
```

Em memória temos:

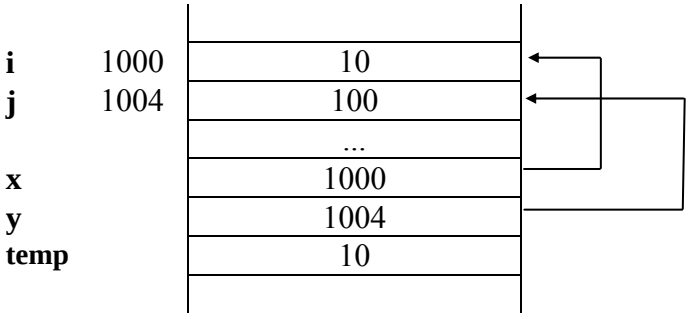
- **Antes de chamar a função**



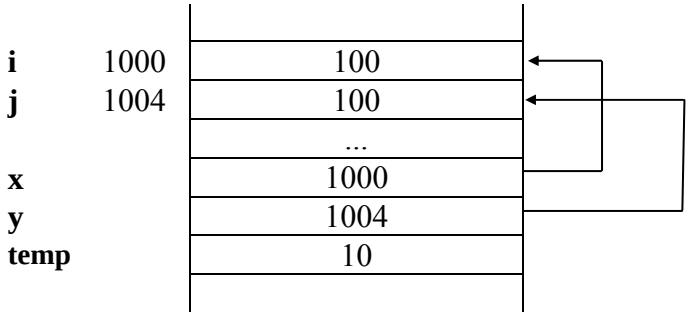
- **Logo após ser invocada a função troca**



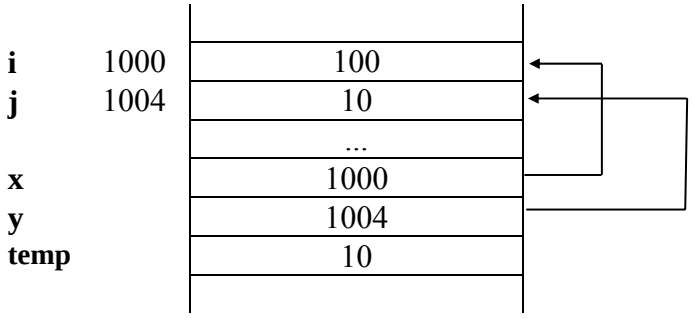
- **No ponto A**



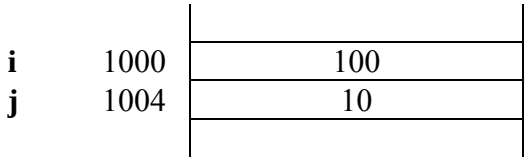
• **No ponto B**



• **no ponto C**



• **no ponto D**



6.9 Passagem de Tabelas para Funções

Como foi referido na secção 6.6, o nome de uma tabela unidimensional corresponde ao endereço do seu primeiro elemento. Como os elementos de uma tabela ocupam posições contíguas de memória, para enviarmos a uma função uma tabela unidimensional é apenas necessário enviar o seu nome e o número de elementos que ela possui, à excepção das tabelas de caracteres que através da função **strlen()** podemos saber o número de caracteres que a tabela contém. O exemplo 6.5 ilustra a passagem de parâmetros no caso de se tratar de tabelas.

Exemplo 6.5: programa que lê, preenche e mostra uma tabela.

```
#include <stdio.h>

/* void leTabela (int p[], int n) */
void leTabela (int *p, int n)
{
        int i;

        printf("Introduza %d valores", n);
        for(i=0; i<n; i++)
                scanf("%d", p+i);
}


/* void apresTabela (int t[], int n) */
void apresTabela (int *t, int n)

{
        int i;
        printf("\n");
        for(i=0; i<n; i++)
                printf("tab[%d]=%d\n", i, t[i]);
}

void main(void)
{
        int tab[10];
        /* na chamada da função utilizamos o endereço inicial da tabela-> tab, podendo
também ser escrito &tab[0] */
        leTabela(tab,10);
        apresTabela(tab,10);
}
```

Repare que o primeiro parâmetro da função **leTabela()** é um apontador para int (**int *p**). Há uma outra forma de declarar este apontador utilizando notação matricial. Esta hipótese foi colocada como comentário.

7. Estruturas (*Struct*)

7.1 Declaração de Estruturas

Apesar de as tabelas serem dados estruturados muito úteis, pois permitem manipular uma grande quantidade de dados de forma indexada, têm a limitação de todos os seus elementos terem de ser do mesmo tipo de dados. Quando se pretende guardar dados relacionados entre si mas que são de tipos diferentes existe a opção do tipo **estrutura** (*struct*, palavra reservada na linguagem C) ou **registo** (*record*). As estruturas permitem agrupar componentes de tipos diferentes designadas por **campos** ou **membros** da estrutura. O tipo de dados dos campos pode ser pré-definido, definido pelo utilizador ou outras estruturas. A única restrição é que o tipo de dados tenha sido previamente definido. Cada campo de uma estrutura possui um identificador, que na literatura inglesa é referido como *tag* (etiqueta) que permite o acesso de forma individualizada a esse campo dentro da estrutura. A sintaxe da declaração em C é:

```
struct identif_estrutura {
      identif_tipo_1 identif_campo_1;
      identif_tipo_2 identif_campo_2;
      ...
      identif_tipo_n identif_campo_n;
};
```

Exemplos:

a) Armazenar informação acerca de objecto numa base de dados

```
struct objecto {
  int codigo;
  char nome[20];
  float peso, comprimento;
};
```

b) Guardar números imaginários

```
struct complexo {
  double real;
  double imag;
};
```

c) Armazenar informação acerca de um aluno

```
struct aluno_isec {
  int num;
  char nome[60];
  char curso[15];
  unsigned short ano_nasc;
  double peso;
  float altura;
};
```

d) Armazenar informação acerca de um aluno; estrutura mais simples que **c)**

```
struct aluno {
    int num;
    char nome[20];
    unsigned short int nasc;
    double peso;
};
```

Para declarar variáveis de um tipo estrutura já declarado, a sintaxe é:

```
struct id_estrutura_definida_antes id_var1, id_var2, ..., id_varn;
```

Exemplo:

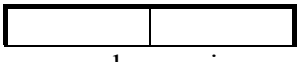
```
struct aluno_isec Maria, Pedro;
struct objecto obj;
struct complexo c1, c2;
```

Quando da declaração do tipo estrutura é possível, **simultaneamente** declarar variáveis desse tipo. Por exemplo, as variáveis **c1** e **c2** podiam ter sido declaradas da seguinte forma:

```
struct complexo {
    double real;
    double imag;
} c1, c2;
```

Para aceder a um determinado campo de uma estrutura temos de utilizar o **operador ponto**. Podemos então visualizar as variáveis **cx**, **al** e **outro**:

cx



cx.real cx.imag

al

al.num al.nome al.nasc al.peso

Exemplo 7.1: programa que lê dois números complexos e apresenta a sua soma nos formatos de parte real/parte imaginária e módulo/fase.

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

struct complexo {
        float real, imag;
};

void main(void)
{
 struct complexo c1, c2, res;
 float modulo, fase;

 printf("Introduza a parte real e imaginária de dois números imaginários.\n");
 scanf("%f%f", &c1.real, &c1.imag);
 scanf("%f%f", &c2.real, &c2.imag);

 res.real = c1.real + c2.real;
 res.imag = c1.imag + c2.imag;

 modulo = sqrt(res.real * res.real + res.imag * res.imag);
/* atan2 retorna ângulo em radianos; temos converter para graus */
 fase = atan2(res.imag, res.real) * 180 / PI;

 printf("Soma dos numeros introduzidos e igual a \n");
 printf("%10.1f + j%-10.1f\n", res.real, res.imag);
 printf("Modulo = %10.4f  fase = j%10.4fº\n", modulo, fase);
}
```

7.2 Inicialização de Estruturas

Podemos inicializar variáveis do tipo **estrutura** em simultâneo com a sua declaração, utilizando para isso listas de valores entre chavetas, de um modo idêntico ao utilizado para inicializar os elementos de uma tabela.

Exemplo: Considerando os tipos **struct complexo** e **struct aluno**

```
struct complexo cx = {2.0, 5.1};
```

A variável **cx** pode ser visualizada como:

cx

2.0	5.1
cx.real	cx.imag

Considerando:

```
struct aluno al = {97200, ”Carlos Ferreira”, 1980, 70.5};
```

A variável **al** pode ser visualizada como:

al

97200	”Carlos Ferreira”	1980	70.5
al.num	al.nome	al.nasc	al.peso

A lista de valores não pode ter mais valores do que o número de campos da estrutura, e se forem menos, os restantes campos serão inicializados com **0**. Se forem colocados mais valores do que o número de campos da estrutura, obtém-se um erro de compilação.

7.3 Estruturas com Membros do Tipo Estrutura

A estrutura **struct aluno_isec** declarada anteriormente para armazenar informação acerca de um aluno, pode ser melhorada de modo a permitir guardar a data de nascimento do aluno e não apenas o seu ano de nascimento. Para tal vamos definir uma estrutura para armazenar uma data, que terá três campos: dia, mês e ano. A nova estrutura **aluno_isec** passará a ter um campo que por sua vez é uma estrutura trata-se de uma **estrutura hierárquica**. Consideremos a declaração dos novos tipos:

```
struct data {
  unsigned short int dia;
  unsigned short int mes;
  unsigned short int ano;
};
struct aluno_isec {
  int num;
  char prim_nome[20];
  char ult_nome[20];
  char curso[15];
  struct data dn;
  double peso;
  float altura;
};
```

Após a declaração da variável **al_elect** podemos guardar a data de nascimento **2/4/1974** acedendo ao campo **dn**. Como este campo por sua vez também é uma estrutura temos de aceder aos seus três campos, usando o **operador ponto (.)**, como nas instruções seguintes:

```
struct aluno_isec al_elect;
al_elect.dn.dia = 2;
al_elect.dn.mes = 4;
al_elect.dn.ano = 1974;
```

Ao contrário das tabelas onde **não** podíamos fazer cópias de toda a tabela sem ser acedendo individualmente aos elementos, com estruturas é possível a atribuição:

```
id_est1 = id_est2;
```

desde que estas estruturas sejam do mesmo tipo. **Todos os campos são integralmente copiados**. O Exemplo 7.2 ilustra o que acabou de ser dito:

Exemplo 7.2: programa que ilustra a cópia de estruturas.

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

struct aluno {
  int num;
  char nome[20];
  unsigned short int nasc;
  double peso;
};

void main(void){
  struct aluno al_x, al_y;

  al_x.num = 97000;
  strcpy(al_x.nome, "Joao");    /* é necessário usar strcpy pois al_x.nome é uma string */

  al_x.nasc = 1977;
  al_x.peso = 80.5;

  al_y = al_x;                  /*copiar as estruturas */

 printf("Dados do aluno x\n");
 printf("%10d%15s%10d%10.2f\n\n", al_x.num, al_x.nome, al_x.nasc,  al_x.peso);
 printf("Dados do aluno y\n");
 printf("%10d%15s%10d%10.2f\n\n", al_y.num, al_y.nome, al_y.nasc, al_y.peso);
}
```

Note-se que o campo **nome** da estrutura **struct aluno** é uma *string* e que esta também foi integralmente copiada. Se um campo for uma tabela de **ints** ou de **floats** vai acontecer o mesmo, ou seja, são copiados todos os elementos da tabela.

7.4 Tabelas de Estruturas

As tabelas estudadas anteriormente têm como elementos tipos pré-definidos. É possível que os componentes de uma tabela sejam estruturas. Esta conjugação revela-se muito útil na prática. Vamos ver um exemplo onde temos uma tabela com 5 elementos, onde cada elemento é uma estrutura com 4 campos:

```
struct aluno {
        int num;
        char nome[20];
        unsigned short int nasc;
        double peso;
};

struct aluno TabAlunos[5];
```

Para compreender melhor a estrutura criada, vamos procurar visualiza-la graficamente:

TabAlunos

0				
1				
2				
3				
4				

Como se efectua o acesso a um campo? Primeiro acedemos ao elemento da tabela que se encontra na segunda posição da tabela (índice 1), para tal utilizamos o operador **[ ]**, e de seguida acedemos ao campo pretendido, utilizando o operador ponto **.**:

```
TabAlunos[1].num
```

A atribuição dos dados: 97001, "António Silva", 1981 e 72.5, ao terceiro elemento da tabela (índice 2) será conseguido através das seguintes instruções:

```
TabAlunos[2].num = 97001;
strcpy(TabAlunos[2].nome,"António Silva");      /*  Cuidado. Este campo é uma string*/
TabAlunos[2].nasc = 1981;
TabAlunos[2].peso = 72.5;
```

Exemplo 7.3: programa que lê os dados de 5 alunos e de seguida indica o número, o nome e a idade de cada um.

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

const int MAX = 5;
struct aluno {
  int num;
  char nome[20];
  unsigned short nasc;
  double peso;
};

void main(void)
{
  struct aluno TabAlunos[MAX];
  int i, idade;

  printf("Introduza dados dos alunos\n");
  for(i = 0; i < MAX; i++){
    printf("Numero, nome, ano de nascimento e peso\n");
    scanf("%d\n", &TabAlunos[i].num);
/* \n é necessário neste scanf; se não existir o campo nome fica  com a string vazia */
    gets(TabAlunos[i].nome);
    scanf("%d%f", &TabAlunos[i].nasc, &TabAlunos[i].peso);
}

  printf("\n\nDados dos alunos\n");
  printf("%-10s%-25s%-10s\n","Numero","Nome","Idade");
  for(i=0; i < MAX; i++){
    idade = 1998 - TabAlunos[i].nasc;
    printf("%-10d%-25s%-10d\n",TabAlunos[i].num,TabAlunos[i].nome, idade);
  }
}
```

7.5 Apontadores para Estruturas

Consideremos a seguinte estrutura:

```
struct stock {
  unsigned int id;
  char nome[20];
  float peso;
};
```

Após conhecido o tipo de dados **struct stock** podemos declarar variáveis do tipo estrutura assim como apontadores para este tipo de dados. Se pretendermos aceder a um campo de uma estrutura através de um apontador temos de, inicializar o apontador com o endereço da variável do tipo estrutura , operador endereço **&** e posteriormente recorrer ao operador indirecção *****. Vamos declarar a variável **obj** do tipo **struct stock** e utilizando o apontador **ptro** vamos guardar nela os seguintes dados: 3300, Boneca, 0.5Kg.

```
struct stock obj, *ptro;
ptro = &obj;          /* ptro aponta para a variável obj */
(*ptro).id = 3300;
strcpy((*ptro).nome, ”Boneca”);
(*ptro). peso = 0.5;
```

A utilização de parênteses nas instruções anteriores é essencial, pois o operador ponto tem prioridade **superior** ao operador indirecção. A utilização dos parênteses e do operador de indirecção em **(*ptro).ident_campo** pode ser substituída pelo operador **->**. Assim as linhas de código anterior podem ser escritas da seguinte forma:

```
ptro->id = 3300;
strcpy(ptro->nome, ”Boneca”);
ptro->peso = 0.5;
```

Exemplo 7.4: programa que guarda os dados introduzidos pelo utilizador numa estrutura através de um apontador e mostra de seguida a informação lida.

```
#include <stdio.h>

struct stock {
   unsigned int id;
   char nome[20];
   float peso;
};

void main(void)
{
  struct stock obj;
  struct stock *ptro;

  ptro = &obj; /* apontador tem de apontar para uma estrutura */
  printf("Introduza nome");
  gets(ptro->nome);
  printf("Introduza numero de identificação e peso");
  scanf("%d%f", &ptro->id, &ptro->peso);
  printf("Os dados introduzidos foram");
  printf("Visualização utilizando o apontador");
  printf("%20d%20s%10.1f\n", ptro->id,ptro->nome, ptro->peso);
  printf("Visualização utilizando o variável obj");
  printf("%20d%20s%10.1f\n", obj.id, obj.nome, obj.peso);
```

7.6 Funções e Estruturas

Uma função pode ter como parâmetros uma, ou mais estruturas. Neste caso, quando da chamada da função, todos os campos do(s) parâmetro(s) concreto(s) (ou argumento(s)) são copiados para o(s) correspondentes parâmetro(s) formais. Suponha-se então a declaração da estrutura seguinte:

```
struct stock {
   unsigned int id;
   char nome[20];
   float peso;
};
```

Uma função que receba uma estrutura do tipo definido e a apresente no monitor será:

```
void apres(struct stock obj)
{
  printf("Dados deste item\n");
  printf("Número de série - %d\n", obj.id);
  printf("Nome - %d\n", obj.nome);
  printf("Peso - %d\n", obj.peso);
}
```

O tipo de retorno de uma função também pode ser do tipo estrutura. Neste caso todos os campos da estrutura são retornados, mesmo que algum ou alguns dos campos sejam do tipo tabela (ou *string*). Vamos então ver uma função que lê os dados para uma estrutura do tipo **struct stock** e retorna os dados lidos:

```
struct stock ler(void)
{
  struct stock obj;

  printf("Introduza dados do item\n");
  printf("Número de série \n"); scanf("%d\n",&obj.id);
  printf("Nome - %d\n"); gets(obj.nome);
  printf("Peso - %d\n";  scanf("%f", &obj.peso);

  return obj;
}
```

Aplicando os conhecimentos apreendidos sobre parâmetros e retorno de funções, podemos rescrever o Exemplo 7.5 de modo a utilizar funções para ler e apresentar valores.

Exemplo 7.5: programa do exemplo 7.3 reescrito através de funções.

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

const int MAX = 5;
struct aluno {
  int num;
  char nome[20];
  unsigned short nasc;
  double peso;
};
```

```
struct aluno le(void)
{
 struct aluno al;

 printf("Numero, nome, ano de nascimento e peso\n");
 scanf("%d\n", &al.num); /* \n para o  campo nome não ficar como uma string vazia */
 gets(al.nome);
 scanf("%d%f", &al.nasc, &al.peso);

 return al;
}

void apresenta(struct aluno ax)
{
 int idade;

 idade = 2001 - ax.nasc;
 printf("%-10d%-25s%-10d\n",ax.num,ax.nome, idade);
}
void main(void)
{
 struct aluno TabAlunos[MAX];
 int i;

 printf("Introduza dados dos alunos\n");
 for(i = 0; i < MAX; i++)
   TabAlunos[i] = le();

 printf("\n\nDados dos alunos\n");
 printf("%-10s%-25s%-10s\n","Numero","Nome","Idade");
 for(i=0; i < MAX; i++)
   apresenta(TabAlunos[i]);
}
```

Como sabemos na linguagem C só existe passagem de parâmetros por valor. Se pretendermos alterar o conteúdo de uma variável do tipo estrutura dentro de uma função teremos de enviar a essa função o seu endereço e posteriormente utilizar o operador indirecção para aceder aos vários campos. No Exemplo 7.6 vamos ilustrar como se processa a passagem de parâmetros de variáveis do tipo estrutura nestas circunstâncias.

Exemplo 7.6: programa que utiliza uma função para actualizar os dados de uma variável do tipo estrutura.

```
#include <stdio.h>

struct stock {
   unsigned int id;
   char nome[20];
   float peso;
};

/* parâmetro obj é um apontador para struct stock */
void altera(struct stock *obj)
{
  printf("Introduza novos dados\n");
  printf("Numero - "); scanf("%d\n", &obj->id);
  printf("Nome - ");
  gets(obj->nome);
  printf("Peso (em Kg) - "); scanf("%f", &obj->peso);
}

void main (void)
{
  struct stock ox={0, "", 0.0};

  printf("Dados \n");
  printf("Numero de identificacao - %d\n", ox.id);
  printf("Nome - %s\n", ox.nome);
  printf("Peso - %8.2f Kg\n", ox.peso);

  altera(&ox); /* temos de passar o endereço de ox como argumento */
  printf("Dados depois de alterados\n");
  printf("Numero de identificacao - %d\n", ox.id);
  printf("Nome - %s\n", ox.nome);
  printf("Peso - %8.2f Kg\n", ox.peso);
}
```

7.7 Bit Fields

Os elementos de uma estrutura podem ser *bit fields*. Um *bit field* é um elemento de uma estrutura composto por um ou mais bits. Utilizando *bit fields* é possível aceder a um determinado bit, ou conjunto de bits, num byte. Os *bit fields* são utilizados quando se pretende armazenar informação no formato mais compacto possível.

Formato geral de um *bit field*

tipo identif_bf : n_bits;

tipo pode ser int ou unsigned, se for int o bit mais significativo é de sinal; **n_bits** é o número de bits ocupados.

Considere que se pretende guardar informação acerca de um aluno, guardando para o efeito o ano em que se encontra, se é aluno do curso diurno ou do nocturno, se possui o estatuto de trabalhador estudante, quantas matrículas já tem e se é repetente no ano que frequenta. Para tal pretende-se utilizar o menor espaço possível para armazenar a informação. Chegámos então ao seguinte tipo de dados:

```
struct inf_aluno {
    unsigned ano   :2;     /*  1, 2 ou 3 */
    unsigned d_m  :1;     /*  0 para diurno, 1 para nocturno */
    unsigned e_te   :1;   /*  0 - estudante, 1 - trabalhador-estudante */
    unsigned n_mat:3;   /*  1..7 */
    unsigned repet:1;     /*  0 - não repetente, 1 - repetente */
};
```

Repare-se que se não fossem utilizados *bit fields*, uma estrutura para armazenar a mesma informação utilizaria 5 *bytes* (se todos os campos fossem do tipo char, por exemplo).

O acesso aos *bit fields* é realizada como se de campos normais se tratassem, ou seja, com o operador ponto. Suponha então que queremos armazenar a seguinte informação: aluno inscrito no 1º ano, no curso diurno, trabalhador-estudante, com 2 matrículas e é repetente. Temos então:

```
struct inf_aluno al;
al.ano   = 1;
al.d_m   = 0;
al.e_te  = 1;
al.n_mat = 2;
al.repet = 1;
```

Podem-se definir *bit fields* que não ocupam todos os bits de um byte, por exemplo:

```
struct produto {
      unsigned codig   :5; /* 0 a 31 */
      unsigned disponiv:1; /* 0 - indisponível, 1 - disponível */
};
```

Não é necessário dar um nome a todos os bits de um *bit field*. No exemplo seguinte podemos aceder ao último e ao primeiro bit de um byte:

```
struct prim_ult {
        unsigned primeiro:1;
        int             :6;
        unsigned ultimo  :1;
};
```

Podemos utilizar *bit fields* e outros tipos na definição de uma estrutura, por exemplo:

```
struct aluno {
        int numero;
        char nome[20];
        unsigned ano  :2;
        unsigned d_m  :1;
        unsigned e_te :1;
        unsigned n_mat:3;
        unsigned repet:1;
float peso;
};
```

Exemplo 7.7: Suponha que se pretendia guardar a data no formato mais compacto possível. A gama de valores a armazenar nos três campos é:

Dia - 1 a 31;
Mes - 1 a 12;
Ano - 1901 a 2020 (vamos apenas representar ano-1900 ou seja 1 a 120).

Vamos ainda escrever uma função para ler a data e outra para a apresentar.

```
#include <stdio.h>

struct data {
  unsigned dia:5; /* 1 .. 31 */
  unsigned mes:4; /* 1 .. 12 */
  unsigned ano:7; /* 1 a 120 */
};

struct data le_data(void)
{
  unsigned short int i, j, k;
  struct data d;

  printf("Introduza a data: dd mm aaaa\n");
/* têm de se utilizar variáveis auxiliares pois não é possível aplicar o operador & a bit fields */
  scanf("%d%d%d", &i, &j, &k);
  d.dia = i;
  d.mes = j;
  d.ano = k - 1900;
  return d;
}

void apresenta_data(struct data dma)
{
printf("Data:%2d-%2d-%2d\n", dma.dia, dma.mes, dma.ano+1900);
}

void main(void)
{
struct data hoje;
hoje = le_data();
apresenta_data(hoje);
}
```

7.8 Uniões

Uma união é uma zona de memória que é partilhada por uma ou mais variáveis.

Sintaxe:

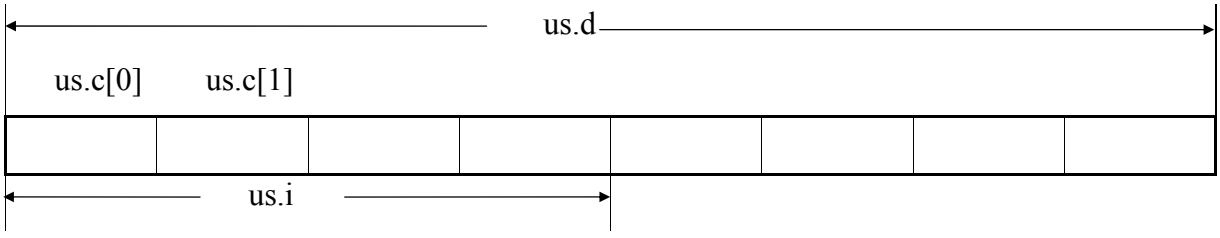
```
union identif_uniao {
        tipo_1  identif_elem_1;
        …
        tipo_n  identif_elem_n;
} [identif_var_1, …, identif_var_m];
```

Embora a sua sintaxe seja idêntica à da definição de uma estrutura, substituindo a palavra reservada **struct** por **union**, resulta num tipo de dados completamente diferente, pois as zonas de memória de cada elemento sobrepõem-se umas às outros. O acesso aos elementos de uma união é feito de um modo idêntico ao acesso aos campos de uma estrutura, utilizando o **operador ponto**. O número de *bytes* ocupados por uma variável do tipo união corresponde ao maior elemento da união.

Exemplo:

```
union varios_tipos {
        int i;
        char c[2];
        double d;
} us;
```

A ilustração da variável **us** na memória ajuda a compreender este tipo de dados. Consideremos que cada rectângulo corresponde a um byte:



Assim posso ter:

```
 us.d = 3.2;
 us.i = 1024;
 us.c[0] = 'a';
 printf("%c", us.c[0]);
//o elemento válido corresponde ao último valor colocado na união
```

Exemplo 7.8: programa para codificar uma variável **short int** através da troca dos seus dois *bytes*, com recurso a uma união.

```
#include <stdio.h>

short int codifica(short int i); /* protótipo da função */

void main(void)
{
        short int x;
        x = codifica(10); /* codifica 10 */
        printf("10 codificado = %d\n", x);
        x = codifica(x); /* descodifica x */
        printf("x descodificado = %d\n", x);
        getch();
}


/* codifica um short int e descodifica um short int codificado */
short int codifica(short int n)
{
        union cripto {
                short int num;
                char c[2];
        } cod;
        char ch;

        cod.num = n;
        ch = cod.c[1];  /* troca os dois bytes do short int */
        cod.c[1] = cod.c[0];
        cod.c[0] = ch;
        return cod.num;
}
```

Resultado no monitor:

```
10 codificado = 2560
x descodificado = 10
```

8. Ficheiros

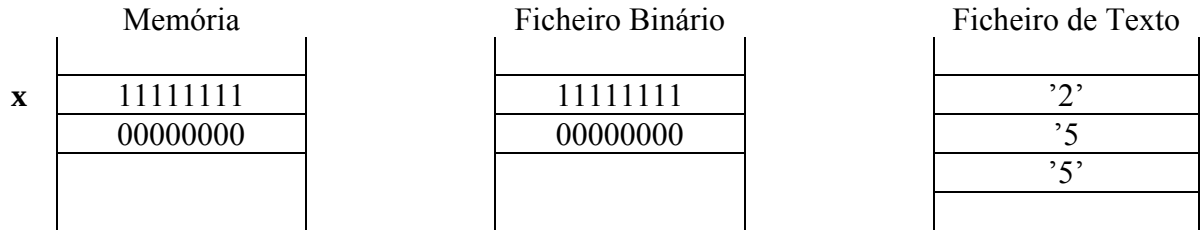
8.1 Ficheiros Texto versus Binário

Num ficheiro de texto toda a informação é escrita na forma de texto, ou seja, utilizando apenas caracteres. Se a informação a escrever é binária então tem de ser transformada num conjunto de caracteres antes de poder ser escrita.

Suponha-se a declaração

```
short int x = 255;
```

Representa-se a seguir o valor na memória, o mesmo valor num ficheiro binário e num ficheiro de texto. Cada rectângulo representa 1 *byte*.



No caso dos ficheiros binários, o valor escrito corresponde aos bits armazenados em memória. Se fosse num ficheiro de texto então seriam escritos os caracteres '2', '5' e '5'.

Os ficheiros são criados em memória secundária, portanto não volátil, o que permite guardar de forma definitiva o seu conteúdo. O tamanho (número de elementos) de um ficheiro pode variar ao longo da execução do programa, só é limitado pelo espaço de memória onde se encontra armazenado.

Na linguagem **C** o acesso a ficheiros é realizado através de *streams*, que correspondem a sequências de *bytes* sem qualquer estrutura interna. Este facto constitui uma vantagem do C pois torna-o *device independent*, a entrada e saída de dados é processada sempre da mesma forma.

8.2 Abertura de um Ficheiro

Para abrir um ficheiro é necessário associar o ficheiro em disco a um *stream*, utilizando para isso a função **fopen().** O protótipo desta função é:

```
FILE *fopen(const char *fname, const char *modo);
```

Esta função, tal como as restantes que lidam com ficheiros, encontra-se definida na biblioteca **stdio.h**. O parâmetro **fname** é uma *string* que contém o nome MS-DOS do ficheiro ao qual queremos aceder e **modo** é o modo de acesso ao ficheiro. Na tabela seguinte encontram-se listados os modos possíveis de acesso a um ficheiro.

Modo	Significado
r	Abre ficheiro de texto só para leitura
w	Cria um ficheiro de texto só para escrita
a	Acrescentar a um ficheiro de texto
rb	Abre ficheiro binário só para leitura
wb	Cria um ficheiro binário só para escrita
ab	Acrescentar a um ficheiro binário
r+	Abre ficheiro de texto para leitura e escrita
w+	Cria um ficheiro de texto para escrita e leitura
a+	Acrescentar ou criar um ficheiro de texto para leitura/escrita
r+b	Abre ficheiro binário para leitura e escrita
w+b	Cria um ficheiro binário para escrita e leitura
a+b	Acrescentar ou criar um ficheiro binário para leitura/escrita

Nota: As letras r, w, a e b vêm do Inglês e são as abreviaturas de *read*, *write*; *append* e *binary*, respectivamente.

A função **fopen()** retorna um apontador para a estrutura **FILE**, estabelecendo um canal de comunicação entre o programa e o ficheiro. Este apontador vai ser utilizados pelas outras funções que lidam com o ficheiro. Se **fopen()** falhar, então é retornado um apontador nulo (**NULL** *pointer*). Quando se tenta abrir um ficheiro deve-se sempre verificar o valor retornado por **fopen()**, e eventualmente, terminar o programa em caso de erro, como se pode ver a seguir:

```
…
FILE *fp;

/* abrir só para leitura; ficheiro de texto */
fp = fopen(”dados.dat”, ”r”);
if(fp == NULL)
{
        printf(”Problemas no acesso ao ficheiro. \n”);
        exit(1);
}
…
```

O tipo **FILE** é uma estrutura e contém a informação necessária para podermos aceder a um ficheiro em disco. Este tipo também se encontra definido em **stdio.h**.

Mais alguma informação acerca dos modos de acesso a ficheiros:

- se abrirmos um ficheiro no modo de leitura (**r** ou **rb**) e o ficheiro não existir então **fopen()** falha (retorna um apontador NULL);
- se abrirmos um ficheiro no modo de acrescento (**a** ou **ab**) e o ficheiro não existir então é criado um ficheiro vazio;
- no modo **a** ou **ab** todos os dados são escritos depois do último elemento do ficheiro; não é possível alterar os elementos já existentes;
- se abrirmos um ficheiro já existente no modo de escrita (**w** ou **wb**) o ficheiro **é apagado** e criado um novo com zero itens;
- a diferença entre os modos **r+** e **w+** é que **r+** não cria o ficheiro se este não existir, ao contrário do modo **w+**.

8.3 Fechar o Ficheiro

Para fechar o ficheiro tem de se utilizar a função **fclose()**, que possui o seguinte protótipo:

```
int fclose(FILE *fp);
```

O apontador passado a **fclose()** deve ser um apontador válido, obtido anteriormente através de uma chamada da função **fopen()**. A função **fclose()** retorna **0** se for bem sucedida e **EOF** se ocorrer algum erro.

8.4 Acesso Byte a Byte

A partir do momento em que o ficheiro estiver aberto, pode-se ler ou escrever um *byte* de informação de cada vez utilizando as funções **fgetc()** e **fputc()**, cujos protótipos são:

int fgetc(FILE *fp); e int fputc(int ch, FILE *fp);

A função **fgetc()** lê o próximo *byte* do ficheiro associado a **fp**, como um **unsigned char** e retorna o valor inteiro correspondente. Se houver erro é retornado **EOF**, que corresponde à constante inteira **-1**, daí o tipo de retorno ser **int** e não **char** ou **unsigned char**. No entanto, pode-se atribuir directamente o valor retornado por **fgetc()** a uma variável do tipo **char**, sendo considerado apenas o *byte* menos significativo. A função **fputc()** escreve um *byte* de informação, armazenado em **ch**, no ficheiro associado a **fp** como um **unsigned char**. Esta função retorna o carácter escrito no caso de ser bem sucedida e **EOF** se houver algum erro.

Exemplo 8.1: programa que escreve uma *string* no ficheiro"letras.txt", depois lê o ficheiro e apresenta o seu conteúdo.

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

void main(void)
{
 char frase[80]="Primeiro teste com ficheiros\nLinha1\nLinha2";
 FILE *ft;
 int i;
 char let;

 /* Abre o ficheiro para escrita*/
 ft = fopen("letras.txt","w");
 if(ft == NULL){
        printf("Impossivel criar o ficheiro\n");
        exit(1);
 }
 i = 0;
 while(frase[i]!='\0'){
        fputc(frase[i],ft);
        i++;
 }
 fclose(ft);

/* Experimente abrir o ficheiro "letras.txt", por exemplo com o Notepad e observe o seu
conteúdo. *//
```

```
/* Abrir o ficheiro para leitura */
        ft = fopen("letras.txt","r");
        if(ft == NULL){
                printf("Impossivel aceder ao ficheiro\n");
                exit(1);
        }
        do{
                let = fgetc(ft);
                printf("%c",let);
        } while(let != EOF);
        fclose(ft);
}
```

8.5 A Função feof

Quando a função **fgetc()** retorna o valor **EOF** pode ter ocorrido uma de duas coisas: chegámos ao fim do ficheiro ou houve um erro no acesso. O problema que pode surgir é se lermos ficheiros binários com **fgetc()**, pois aí qualquer combinação de bits é válida e um dos *bytes* lidos pode corresponder a **EOF** e ainda não estarmos no fim do ficheiro. De modo a eliminar esta ambiguidade podemos utilizar a função **feof()** que nos indica se já estamos ou não no fim do ficheiro. O seu protótipo é:

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

Esta função retorna um valor diferente de zero se estivermos no fim do ficheiro e **0** se não estivermos. Assim para apresentar o conteúdo de um ficheiro utilizando esta nova função temos:

```
FILE *fp;

while(feof(fp) == 0)
        putchar(fgetc(fp));
/* equivalente a printf("%c", car), onde car será uma variável do tipo char */
```

8.6 Escrita/Leitura Formatada em Ficheiros de Texto

No acesso a ficheiros de texto podemos utilizar funções que nos permitam ler e escrever *strings* em vez de carácter em carácter. Duas dessas funções são **fputs()** e **fgets()**, cujos protótipos se indicam a seguir:

```
int fputs(char *str, FILE *fp);

char *fgets(char *str, int num, FILE *fp);
```

A função **fputs()** escreve a *string* apontada por **str** no ficheiro associado a **fp**. Retorna **EOF** se houver erro e um valor não negativo se for bem sucedida. O carácter nulo que termina a *string* não é escrito, assim como não é colocada uma mudança de linha.

A função **fgets()** lê caracteres do ficheiro associado a **fp** para a *string* apontada por **str** até **num-1** caracteres terem sido lidos, ter sido encontrado um carácter de mudança de linha ou ter-se chegado ao fim do ficheiro. O carácter de mudança de linha também é lido (se existir). Esta função retorna **str** em caso de sucesso e **NULL** em caso de haver algum erro.

Há mais duas funções que nos permitem realizar escrita/leitura formatada num ficheiro de texto, são elas **fprintf()** e **fscanf()**. Estas funções trabalham exactamente da mesma maneira que **printf()** e **scanf()**, só que agora o resultado é escrito num ficheiro em vez do monitor e é lido de um ficheiro em vez do teclado, respectivamente. Os seus protótipos são:

```
int fprintf(FILE *fp, char *string_formatação, ...);

int fscanf(FILE *fp, char *string_formatação, ...);
```

Nos exemplos 8.2 e 8.3 são utilizadas estas funções. Utilize um editor de texto e verifique o conteúdo dos ficheiros que criou nestes exemplos.

Exemplo 8.2: programa que lê linhas de texto introduzidas pelo utilizador, terminando por uma linha em branco. De seguida apresenta o texto introduzido e gravado no ficheiro.

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

void main(void){
  FILE *ftexto;
  char linha[80];
  int comp, conta;

  ftexto = fopen("ecrans.txt","w");   /* abrir ficheiro para escrita */
  if (ftexto == NULL){
          printf("Impossivel criar o ficheiro");
          exit(1);
  }
  printf("Introduza varias linhas de texto\n Uma linha em branco termina introdução! \n");
  do{
          printf("?:");
          gets(linha);
          comp = strlen(linha);
          if (comp != 0){
             strcat(linha, "\n");   /* acrescenta mudança de linha */
             fputs(linha, ftexto);
            /*as duas linhas anteriores podem ser substituídas por:  fprintf(ftexto, "%s\n", linha); */
          }
  } while(comp != 0);
  fclose(ftexto);
 /* abrir ficheiro para leitura */
  ftexto = fopen("ecrans.txt","r");
  if (ftexto == NULL){
          printf("Impossivel aceder ao ficheiro");
          exit(1);
  }
  conta = 0;
  printf("O texto introduzido foi:\n");
  do{
          fgets(linha, 79, ftexto);
          if(feof(ftexto)==0){
                  conta++;
                  printf("%2d:%s",cont, linha);
                  /* pára a cada 24 linhas */
                  if(conta%24 == 0){
                          printf("Qualquer tecla para prosseguir…\n");
                          getch();
                  }
          }
  } while(feof(ftexto)==0);
  fclose(ftexto);
  getch();
}
```

Exemplo 8.3: programa que escreve num ficheiro os inteiros entre 1 e 20, as suas raízes e quadrados e a seguir lê o seu conteúdo.

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

void main(void)
{
 FILE *ft;
 char linha[80];
 int i, ii;
 float f;

 /* abrir ficheiro para escrita */
 ft = fopen("tabela.txt","w");
 if (ft == NULL){
        printf("Impossivel criar o ficheiro");
        exit(1);
 }
 fprintf(ft,"%10c%10s%10s\n",'i',"raiz", "quadrado");
 for(i=0; i<30; i++)
        fprintf(ft, "-");
 fprintf(ft, "\n");
 for(i=1; i<=20; i++)
        fprintf(ft, "%10d%10.3f%10d\n", i, sqrt(i), i*i);
 fclose(ft);

 /* abrir ficheiro para leitura */
 ft = fopen("tabela.txt","r");
 if (ft == NULL){
        printf("Impossivel aceder ao ficheiro");
        exit(1);
 }

 printf("Conteúdo do ficheiro\n");
 /*ler e apresentar as 2 primeiras linhas do ficheiro */
 fgets(linha, 79, ft);
 puts(linha);
 fgets(linha, 79, ft);
 puts(linha);

 while(feof(ft)==0){
        fscanf(ft,"%d%f%d\n", &i, &f, &ii);
        printf("%10d%10.3f%10d\n", i, f, ii);
 }

 fclose(ft);
 getch();
}
```

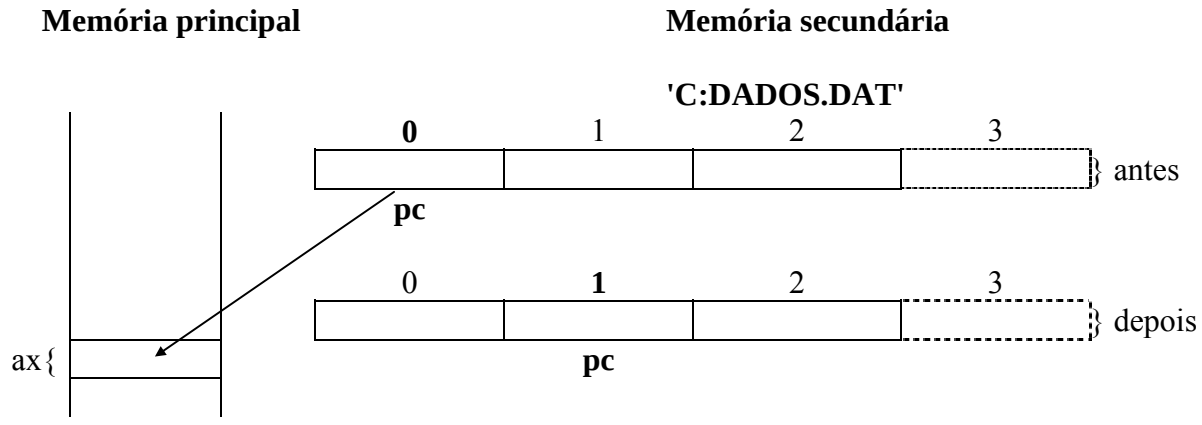
8.7 Acesso Binário (Ficheiros Binários)

Embora **fprintf()** e **fscanf()** sejam duas funções que nos permitem aceder facilmente a um ficheiro, não constituem a maneira mais eficiente de o fazer. Isto deve-se ao facto de quando se pretende escrever/ler um número no ficheiro este tem de ser convertido do seu formato interno num conjunto de caracteres ASCII ou vice-versa no caso da leitura, o que faz perder alguma eficiência ao programa. Para além disso, o ficheiro criado utilizando a função **fprintf()** para escrita é geralmente maior do que um que seja a cópia integral dos bits usados no formato interno (binário).

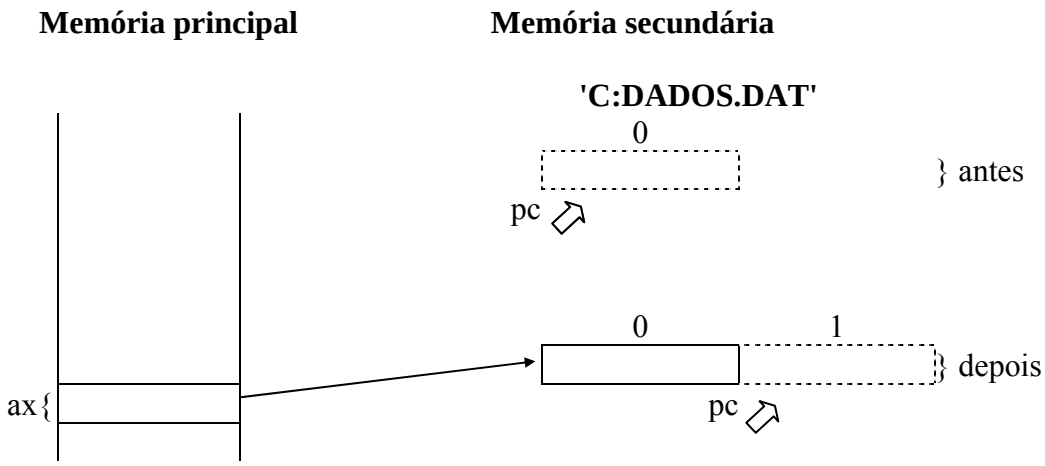
Na linguagem **C** encontram-se definidas as funções **fread()** e **fwrite()** cujos protótipos são:

```
size_t fread(void *buffer, size_t size, size_t num, FILE *fp);

size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
```

onde size_t é equivalente a **long** ou **unsigned long**. A função **fread()** lê a partir do ficheiro associado a **fp**, **num** elementos cada um de tamanho **size_t**, para a zona de memória apontada por **buffer**. Retorna o número de elementos lidos com sucesso. Se o valor retornado for **zero** então não foi lido nenhum elemento, o que significa que chegámos ao fim do ficheiro ou que houve um erro de leitura.



A função **fwrite()** é complementar de **fread()**, escreve no ficheiro associado a **fp num** elementos, onde cada elemento tem **size_t** *bytes*, que se encontram a partir da zona de memória apontado por **buffer**. O valor retornado corresponde ao número de elementos escritos com sucesso.



Repare-se que a **p**osição **c**orrente no ficheiro, **pc** nas figuras anteriores (em inglês é *current position*), avança para a posição seguinte sempre que se faz uma leitura ou escrita no ficheiro. Daí o acesso a ficheiros ser um processo essencialmente sequencial.

Exemplo 8.4: utilização de um ficheiro binário para armazenar as raízes dos 100 primeiros inteiros.

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

void main(void)
{
 FILE *fb;
 int i;
 float f;

 fb = fopen("raizes.dat","wb"); /* é criado ficheiro vazio */
 if (fb == NULL){
        printf("Impossivel criar o ficheiro");
        exit(1);
 }

//escrever no ficheiro as raízes dos números compreendidos entre 0 e 99
 for(i=0; i<100; i++){
        f = sqrt(i);
        fwrite(&f, sizeof(float), 1, fb);
 }
 fclose(fb);
 fb = fopen("raizes.dat","rb");
```

```
/* ficheiro aberto só para leitura; é um ficheiro binário */
  if (fb == NULL){
          printf("Impossivel aceder ao ficheiro");
          exit(1);
  }

  printf("Conteúdo do ficheiro\n");
  while(fread(&f, sizeof(float), 1, fb)!=0)
          printf("%10.2f\n", f);

  fclose(fb);
 getch();
}
```

Se tivéssemos declarado uma tabela e preenchido os seus elementos com as raízes dos números inteiros, então podíamos fazer a escrita, e a leitura, utilizando apenas uma vez a função **fwrite()** na escrita e **fread()** na leitura.

```
float raizes[100];
...
for(i=0; i<100; i++)
      raizes[i] = sqrt(i);
fwrite(raizes, sizeof(float), 100, fb);
/* ou fwrite(raizes, sizeof(raizes), 1, fb)*/
fread(raizes, sizeof(float), 100, fb);
```

8.8 Acesso Aleatório

O tipo de acesso utilizado anteriormente é sequencial, do princípio para o fim para o fim do ficheiro. No entanto, usando outras funções também definidas em **stdio.h**, é possível aceder a qualquer elemento do ficheiro, em qualquer instante. Para isso dispomos da função **fseek()**, cujo protótipo é:

```
int fseek(FILE *fp, long offset, int origem);
```

Aqui **fp** está associado ao ficheiro cujos elementos pretendemos aceder, o valor do **offset** determina o número de *bytes* a partir de **origem**, que passará a ser a nova posição corrente. **origem** deve ser um dos valores da lista seguinte:

Origem	Significado
SEEK_SET	a partir do início do ficheiro
SEEK_CUR	a partir da posição corrente
SEEK_END	a partir do fim do ficheiro

A função **fseek()** retorna o valor **0** se for bem sucedida e um valor diferente de zero de **0** se não o for. Outra função útil que nos permite obter a posição corrente no ficheiro é a função **ftell()**:

```
long ftell(FILE *fp);
```

Esta função retorna o valor da posição corrente no ficheiro associado a **fp**. Se houver erro retorna o valor **-1**.

8.9 Outras Funções

Pode-se eliminar um ficheiro usando a função **remove()**. O seu protótipo é:

```
int remove(char * fname);
```

Esta função apaga o ficheiro cujo nome é apontado por **fname**. Retorna o valor **0** no caso de ser bem sucedida e um valor diferente de zero no caso contrário.

Pode-se alterar o nome de um ficheiro utilizando a função **rename()**, cujo protótipo é:

```
int rename(char *fname, char *newfname);
```

onde **fname** é um apontador para a *string* que contém o nome antigo e **newname** o novo nome do ficheiro. No caso de ser bem sucedida retorna o valor **0** e no caso contrário o valor **-1**. Pode-se colocar a posição corrente de um ficheiro associado a **fp** no início pode ser usada a função **rewind()**. O seu protótipo é:

```
void rewind(FILE *fp);
```

A função **fflush()** obriga o *buffer* de interface a ser esvaziado, escrevendo o seu conteúdo no ficheiro associado a **fp**. Retorna o valor **0** no caso de ser bem sucedida e **EOF** em caso

de falha. O protótipo é:

```
int fflush(FILE *fp);
```

Podemos aceder em "simultâneo" a um ficheiro com dois ou mais *streams,* como mostra o Exemplo 8.5.

Exemplo 8.5: Programa que ilustra o acesso a um ficheiro com vários *streams*.

```
#include <stdio.h>
void main(void)
{
 FILE *f1, *f2;
 int i, val1, val2;

 /* preencher o ficheiro */
 f1 = fopen("teste.dat","wb");
 if(f1==NULL)
        printf("ERRO ...\n");
 /* escreve no ficheiro binário os números de 0 a 19 */
 for(i=0;i<20;i++)
        fwrite(&i, sizeof(int), 1, f1);
 fclose(f1);

/* vamos aceder com f1 do princípio para o fim e com f2 do fim para o inicio */
 f1 = fopen("teste.dat","rb");
 f2 = fopen("teste.dat","rb");
 for(i=0;i<20;i++){
        fseek(f1, i*sizeof(int), SEEK_SET);
        // posição corrente(1) i elementos a partir do início
        fread(&val1,sizeof(int),1,f1);
        fseek(f2, (-i-1)*sizeof(int), SEEK_END);
        // posicao corrente(2) i elementos para trás, a partir do fim do ficheiro
        fread(&val2,sizeof(int),1,f2);
        printf("%12d%12d\n", val1, val2);
 }
 fclose(f1);
 fclose(f2);
}
```

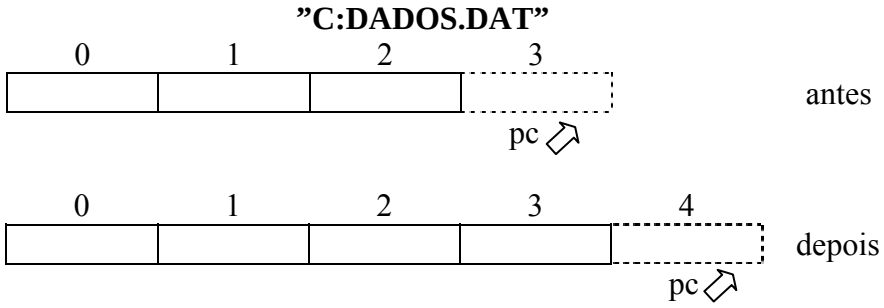
8.10 Inserir, Alterar e Apagar Elementos de um Ficheiro

Utilizando as funções estudadas nas secções anteriores vamos analisar como é possível: inserir novos elementos, alterar um elemento já existente e apagar um elemento em ficheiros. No código que ilustra as várias acções vamos considerar a seguinte estrutura:

```
struct dados {
      int id;
      char nome[20];
      float peso;
};
```

8.10.1 Inserção de Novo Elemento

Temos de colocar o **pc** a apontar para depois da última posição válida e depois escrever o elemento no ficheiro.




```
…
struct dados al;
FILE *f;
/* ler campos de al*/
...
/* abrir ficheiro no modo acrescentar, portanto pc no fim do ficheiro, após último
elemento válido */
f=fopen("dados.dat","ab");
/* e escrever */
fwrite(&al, sizeof(struct dados), 1, f);
…
```

Outra forma será abrir no modo **r+b** e utilizar **fseek()** para colocar **pc** a apontar para depois da última posição válida, como se pode ver a seguir:

```
f=fopen("dados.dat","r+b");
fseek(f, 0, SEEK_END);
fwrite(&al, sizeof(struct dados), 1, f); /* e escrever */
```

8.10.2 Alteração de Elementos

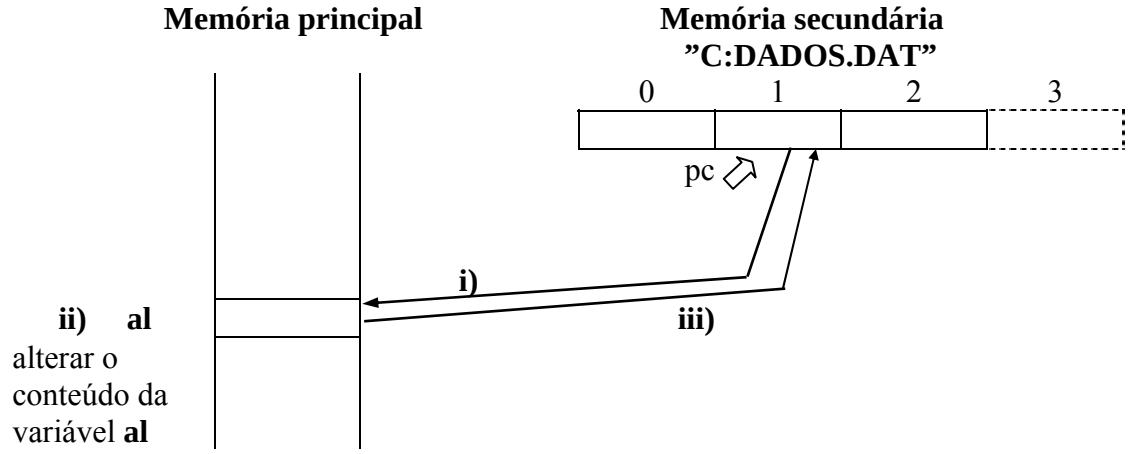
Para alterar um elemento de um ficheiro temos de realizar a seguinte sequência de acções:

i) Ler o elemento a alterar para a memória principal (para uma variável);

ii) Alterar o elemento na memória principal;

iii) Gravar o elemento alterado no ficheiro sobre o elemento antigo.

Podemos representar graficamente a sequência de acções da seguinte forma:



A actualização de um elemento de um ficheiro pode ser implementada da seguinte forma:

```
…
struct dados al;
...
/* ler valor para a memória */
fread(&al, sizeof(struct dados), 1, f);
/*depois da leitura pc passa a apontar para a estrutura seguinte */
/* actualizar em memória */
 . . .
/*colocar pc a apontar uma estrutura atrás */
fseek(f, -1*sizeof(al), SEEK_CUR);
/* ou
fseek((f, ftell(f)-1*sizeof(al), SEEK_SET);
 */

/*e escrevemos por cima */
fwrite(&al, sizeof(struct dados), 1, f);
…
```

8.10.3 Apagar Elemento do Ficheiro

Para apagar um elemento de um ficheiro a solução mais simples consiste em (supondo a existência do ficheiro **"C:DADOS.DAT"**):

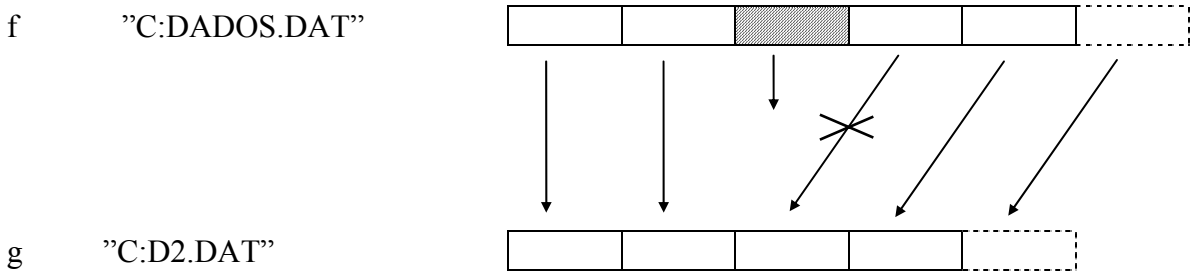
i) Copiar todos os elementos para um ficheiro auxiliar, por exemplo "**C:D2.DAT**", excepto o elemento (ou elementos) que se pretende(m) apagar;

ii) Apagar o ficheiro original "**C:DADOS.DAT**";

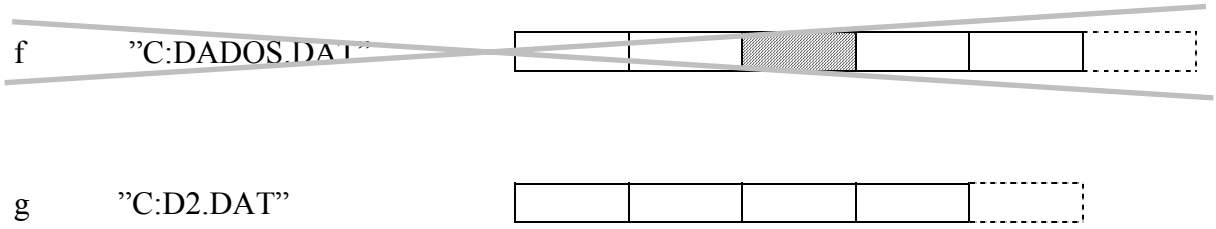
iii) Alterar o nome do ficheiro auxiliar de "**C:D2.DAT**" para "**C:DADOS.DAT**".

A visualização gráfica dos passos anteriores ajuda a perceber melhor o que de facto sucede:

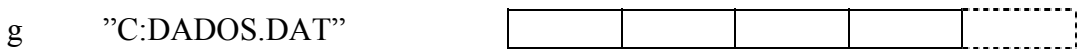
i)



ii)



iii)



A implementação correspondente em C:

```
…
FILE *f, *g;
int n;
struct dados al;
...
/* a eliminação é realizada conhecendo o número do aluno */
printf(”Número do aluno a eliminar”);
scanf(”%d”, &n);
...
f = fopen(”DADOS.DAT”,”rb”); /* vamos ler do ficheiro original, logo modo rb */
g = fopen(”D2.DAT”,”wb”); /* é criado um ficheiro vazio */

while(fread(&al, sizeof(al), 1, f) != 0)
  if(al.id != n)
    fwrite(&al, sizeof(al), 1, g);
fclose(f);
fclose(g); /* depois de tudo copiado, fechar os ficheiros */
remove(”DADOS.DAT”); /* remover ficheiro original */
rename(”D2.DAT”,”DADOS.DAT”); // mudar o nome do ficheiro auxiliar
…
```

Exemplo 8.6: Pretende-se um programa para fazer a gestão de uma base de dados de alunos, onde seja possível realizar as seguintes acções: introduzir novo aluno, mostrar todos alunos, eliminar aluno e alterar dados de aluno. O programa deve ser dividido em várias funções, cada uma delas com uma tarefa perfeitamente definida.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
const char nomefich[20]="alunos.dat";
const char nome2[20]="al2.dat";

struct aluno {
  int id;
  char nome[80];
  int nota;
};

const int tam = sizeof(struct aluno);

/* apresenta dados do aluno no monitor */
void apres_dados(struct aluno a)
{
  printf("\nDados do aluno\n");
  printf("Nome   - %s\n", a.nome);
  printf("Numero - %d\n", a.id);
  printf("Nota   - %d\n", a.nota);
```

```
}

/* le os dados de um aluno */
struct aluno le_dados(void)
{
  struct aluno al;

  printf("\nIntroduza numero do aluno \n");
  scanf("%d\n", &al.id);
  printf("Introduza nome do aluno \n");
  gets(al.nome);
  printf("Introduza nota \n");
  scanf("%d", &al.nota);
  return al;
}

/* cria um ficheiro vazio se este não existir */
void cria_ficheiro(void)
{
  FILE *f;

/* nomefich é uma string constante que contém o nome MS-DOS do ficheiro*/
  f = fopen(nomefich, "rb");
  if (f==NULL) /* se o ficheiro não existir */
          f = fopen(nomefich, "wb");
  fclose(f);
}
/* le e grava novo elemento */
void grava(void)
{
  struct aluno ax;
  FILE *f;
/* a leitura dos dados do aluno é feita utilizando a função le_dados */
  ax = le_dados();
  f=fopen(nomefich, "ab");
  if(f==NULL){
          printf("Impossivel aceder ao ficheiro\n");
          exit(1);
  }

  fwrite(&ax, tam, 1, f);
  fclose(f);
}
```

```
/* mostra todos os alunos */
void mostra(void)
{
 struct aluno al;
 FILE *f;
 f=fopen(nomefich, "rb");
 if(f==NULL){
        printf("Impossivel aceder ao ficheiro\n");
        exit(1);
 }

 while(fread(&al, tam, 1, f) !=0){
  apres_dados(al); /* os dados são apresentados através da  função apres_dados */
  printf("Qualquer tecla para prosseguir\n");
  getch();
 }
 fclose(f);
}
/* elimina aluno com determinado número */
void elimina(void)
{
 struct aluno ax;
 FILE *f, *g;
 int num;

 printf("\nNumero do aluno a eliminar - ");
 scanf("%d", &num);
 f=fopen(nomefich, "rb");
 if(f==NULL){
        printf("Impossivel aceder ao ficheiro\n");
        exit(1);
 }
 g=fopen(nome2, "wb");
 if(g==NULL){
        printf("Impossivel criar ficheiro auxiliar \n");
        exit(1);
 }

 while(fread(&ax, tam, 1, f) !=0){
        if (ax.id != num)
                fwrite(&ax, tam, 1, g);
 }

 fclose(f);
 fclose(g);
 remove(nomefich);
 rename(nome2, nomefich);
```

```
  }
/* --- altera dados de um aluno --- */
void altera(void)
{
  struct aluno ax;
  FILE *f;
  int num;

  printf("\nNumero do aluno ");
  scanf("%d", &num);
  f=fopen(nomefich, "r+b");
  if(f==NULL){
        printf("Impossivel aceder ao ficheiro\n");
        exit(1);
  }

  while(fread(&ax, tam, 1, f) !=0){
        if (ax.id == num){
/* se existir um aluno com esse numero*/
                printf("\nDados antigos\n");
                apres_dados(ax);
                printf("Introduza novos dados\n");
                ax = le_dados();
                fseek(f, -tam, SEEK_CUR);
                fwrite(&ax, tam, 1, f);
                fseek(f, 0, SEEK_CUR);
        }
  }
  fclose(f);
}

/* apresenta opcoes */
void menu(void){
  printf("\n\nOpcoes\n");
  printf("n - novo aluno\n");
  printf("m - mostra todos\n");
  printf("e - elimina aluno\n");
  printf("a - alterad dados de aluno\n");
  printf("s - sair\n");
  printf("A sua opção e':\n");
}
```

```
/* gestao das funcionalidades */
void gere(void){
  char resp;
  do{
          menu();
          resp = tolower(getch());
          switch(resp){
                  case 'n': grava();
                          break;
                  case 'm': mostra();
                          break;
                  case 'e': elimina();
                          break;
                  case 'a': altera();
                          break;
          }
  }while(resp !='s');
}
void main(void){

/* a função cria_ficheiro() verifica se o ficheiro já existe e se não existir cria-o. */
*/ esta função podia ser mais genérica se tivesse como argumento o nome do ficheiro.*/
  cria_ficheiro();

  gere();
}
```

9. Tópicos Vários

9.1 Tipos Enumerados

Na Linguagem C a palavra reservada **enum** permite definir os chamados tipos enumerados, que se destinam a variáveis que possuem um pequeno número de possíveis valores. Um tipo enumerado define uma lista de identificadores únicos e associa um valor a cada um dos identificadores. Os identificadores têm de ser únicos, o mesmo já não se aplica aos valores associados a cada identificador que podem ser repetidos.

Sintaxe:

```
enum tipo_enum {lista_de_identificadores_entre_virgulas};
```

Exemplos do tipo enumerado**:**

```
enum logico {falso, verdadeiro};
enum dias_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};
enum cursos_isec {electrotecnia, mecanica, civil, quimica, informatica};
```

O compilador associa um inteiro a cada identificador, e se nada for especificado, ao primeiro identificador associa o valor 0, ao segundo o valor 1, etc ...
Se for necessário podemos especificar uma ou mais associações entre identificadores e números. Por exemplo:

```
enum cursos_isec {electrotecnia=5, mecanica, civil, quimica, informatica};
```

Aqui a electrotecnia está associado o valor 5, a mecanica o valor 6 e assim sucessivamente. Se for desejável podemos associar um valor a cada identificador, tal como podemos ver no exemplo seguinte:

```
enum cursos_isec {electrotecnia=5, mecanica=10, civil=15, quimica=30,
informatica=100};
```

No conjunto de valores associados aos identificadores podem existir valores repetidos. Por exemplo:

```
enum boolean {indiferente, false=0, true};
```

Aqui **false** está associado o valor 0, e a **true** o valor 1 e a indiferente o valor 0. Repare-se que há valores associados repetidos mas não há, nem pode haver, identificadores repetidos.

Declaração de variáveis do tipo enumerado:

Sintaxe:

```
enum tipo_enum {lista_enumerada} var1, var2, ..., varn;
```

O tipo enumerado e a declaração das variáveis pode ser feito separadamente

```
enum tipo_enum {lista_enumerada};
enum tipo_enum var1, var2, ..., varn;
```

Exemplo:

```
enum cores {preto, azul, vermelho, verde, branco} cor_fundo, cor;
```

o que é equivalente a:

```
enum cores {preto, azul, vermelho, verde, branco};
enum cores cor_fundo, cor;
```

9.2 Operador Condicional ? :

O operador (**? :**) implementa uma expressão condicional. É o único operador ternário na linguagem **C**. A instrução condicional é uma forma abreviada da instrução simples **if-else**

```
(condição) ? expressão_1 : expressão_2;
```

O resultado de uma expressão condicional corresponde à **expressão_1** se a condição for verdadeira e à **expressão_2** no caso contrário. Podemos utilizar a expressão condicional numa instrução de atribuição. Assim

```
ident_variavel = ((condição) ? expressão_1 : expressão_2);
```

onde vemos que a expressão condicional testa a condição; se essa condição for verdadeira atribuirá o valor resultante da **expressão_1** à variável pretendida. Caso contrário atribuirá o

resultado da **expressão_2** à variável. A implementação através da instrução **if-else** é:

```
if (condição)
  ident_variavel = expressão_1;
else
  ident_variavel = expressão_2;
```

Exemplos:

```
int x1 = 2, x2 = 4, max, c;
/* max toma o valor do maior dos dois valores */
max = (x1>x2) ? x1 : x2;
/* também pode ser utilizada numa expressão aritmética */
c = 5 + 2*((x1>x2) ? x1 : x2));
/* c vai tomar o valor 5 + 2 * 4, ou seja, 13*/
```

9.3 Expressões com o Operador Vírgula

Uma expressão com o **operador vírgula** é uma expressão que consiste numa lista de expressões separadas por vírgulas. Todas as expressões são calculadas da esquerda para a direita. O resultado de uma expressão com o operador vírgula corresponde ao valor resultante da última expressão da lista.

Sintaxe:

```
(expressao_1, expressao_2, ..., expressao_n)
```

Exemplos:

```
int k = 0, i = 4, j = 2;
res = (k++, 5*2);  /* é atribuído a res o valor 10 */

/* o resultado de uma expressão com o operador vírgula pode ser utilizada num
expressão aritmética */
y = 3*(i++, j--, j*i);
/* em primeiro lugar i é incrementado, passa a ter o valor 5, j é decrementado, passa a
ter o valor 1, e j*i é então 5. O resultado da expressão colocada entre parênteses é
então 5. y vai tomar o valor 3*5 */
```

```
…

/* pedaço de código para apresentar os números entre 1 e 20 e 20 e 1, em simultâneo */
int i, j;
for (i = 1, j = 20; i <= 20; i++, j--)
      printf("%d\t%d\n", i, j);
…
```

9.4 Definição de Tipos Utilizando typedef

A palavra reservada **typedef** permite associar novos identificadores a tipos pré-definidos na linguagem ou a tipos definidos pelo utilizador. Pode-se posteriormente utilizar o novo identificador na declaração de variáveis.

Sintaxe para tipos de dados pré-definidos:

```
typedef identif_tipo_conhecido novo_identif_tipo;
```

Exemplos:

```
/* definir novos identificadores */
typedef int inteiro;
typedef float real;
typedef unsigned char byte;

/* declaração de variáveis */
inteiro i, j;
real x;
byte on, acesso;
```

Sintaxe para tabelas:

```
typedef tipo_elementos identif_tipo[numero_elementos]
```

Exemplos:

```
typedef int Tabela[10];
typedef float Matriz[5][10];

/* declaração de variáveis */
Tabela tab;
Matriz matA, matB;
```

Sintaxe Para estruturas:

```
typedef struct identif_estrutura {
    identif_tipo_1 identif_campo_1;
    identif_tipo_2 identif_campo_2;
    ...
    identif_tipo_n identif_campo_n;
} novo_identif_tipo;
```

Quando se utiliza o **typedef** a colocação de **identif_estrutura**.

Exemplos:

```
typedef struct aluno_isec {
      char nome[100];
      char curso[15];
      unsigned short ano_nasc;
      unsigned idade;
      double peso;
} ALISEC;

ALISEC alunox, alunoy;
/* posso usar o tipo ALISEC em vez de struct aluno_isec*/
```

Também podemos utilizar typedef para tipos enumerados, por exemplo:

```
typedef enum cores {preto, azul, vermelho, verde, branco} CORES;
/* e declarar variáveis */
CORES cor_fundo, cor;
```

9.5 Macros

As linhas começadas pelo carácter # são processadas (pelo pré-processador) antes da etapa normal de compilação. Só é possível definir uma macro em cada linha de código e essa linha não termina por ponto e vírgula.

```
#define texto_original texto_substituicao
```

Na fase de pré-processamento todas as ocorrências da palavra **texto_original** são substituídas pela palavra **texto_substituicao**. Se no código do nosso programa surgir

```
#define  comp  100
...
c = comp * x;
```

isto será "lido" pelo compilador como

```
c = 100 * x;
```

É possível definir macros que tenham parâmetros, conseguindo-se desta forma escrever funções simples. Um exemplo de uma macro com parâmetros:

```
#define funmac(x,y) x*x + y*y
```

funmac é semelhante a uma função formal, mas esta semelhança não é uma equivalência. Podemos ter o código

```
z = funmac(a, b); /* expressão_1 */
```

que este, na fase de pré-processamento é transformado em

```
z = a*a + b*b;
```

ou seja, o código que o compilador vai compilar é

```
z = a*a + b*b;
```

A escrita da macro anterior falha em casos como o seguinte

```
z = 3*funmac(a+1, b+2);
```

que será pré-processada e dará origem ao código

```
z = 3 * a + 1 * a + 1 + b + 2 * b + 2;
```

o que não corresponde de forma nenhuma ao que nós pretendíamos. Para obter o resultado desejado, e como precaução na correcta definição de macros devemos utilizar:

- parênteses em torno dos parâmetros;
- parênteses em torno do texto a substituir.

Seguindo estas regras a macro é escrita da seguinte forma:

```
#define funmac(x ,y) ((x)*(x) + (y)*(y))
```

e assim a expansão da expressão_1 é:

```
z = 3 * ( (a + 1) * (a +1) + (b + 2) * (b + 2) );
```

o que corresponde ao pretendido.

Se a macro tiver parâmetros não devemos colocar espaços antes da abertura dos parênteses (a seguir ao nome da macro), pois nesse caso o pré-processador considera que a macro não tem parâmetros. Assim, se tivermos,

```
#define quad (x) ((x)*(x))
```

e tivermos o código

```
y = quad(a);
```

este é substituído por

```
y = (x) ((x)*(x))(a)
```

Exemplo:

/ utilizar o operador ?: para definir uma macro que nos permita calcular o maior de dois valores */*

```
#define max(x,y)  (((x) > (y)) ? (x) : (y))
```

9.6 Argumentos de um Programa

É possível passar parâmetros a um programa quando este é invocado, sendo os argumentos a informação que segue o nome do programa quando este é "chamado" na linha de comando. Os parâmetros da função **main()** são chamados **argc** e **argv**. O parâmetro **argc** contém o número de argumentos passados na linha de comando e é um valor inteiro. O parâmetro **argv** é uma tabela de apontadores para *strings*. O modo mais comum de declarar **argv** é: **char *argv[]**. Os parênteses rectos sem um valor entre eles indicam que se trata de uma tabela com número indeterminado de elementos à partida. Para aceder às *strings* uma a uma basta indexar **argv**. Assim **argv[0]** aponta para o nome do programa, **argv[1]** para o primeiro argumento, e assim sucessivamente.

E**xemplo 9.1:** Programa que mostra os argumentos passados em linha de comando.

```
/* programa teste.c; executável teste.exe */
void main(int argc, char *argv[]){
  int i;
  printf("Sao %d argumentos\n", argc);
  for(i=0;i<argc;i++)
    printf("%dº -> %s\n", i, argv[i]);
}
```

Se eu escrevesse na linha de comando

```
c:\djgpp\cprog>teste certo errado ontem 12345
```

o resultado do programa seria

```
São 5 argumentos
0º -> c:\tc\teste.exe
1º -> certo
2º -> errado
3º -> ontem
4º -> 12345
```

Se um dos argumentos contiver espaços, então este deve ser especificado entre aspas (por exemplo: teste ”primeiro segundo” certo). Repare-se que os argumentos são sempre interpretados como *strings*.

Exemplo 9.2: Programa para copiar 2 ficheiros *byte*-a-*byte*. Os nomes dos ficheiros origem e destino são passados na linha de comando.

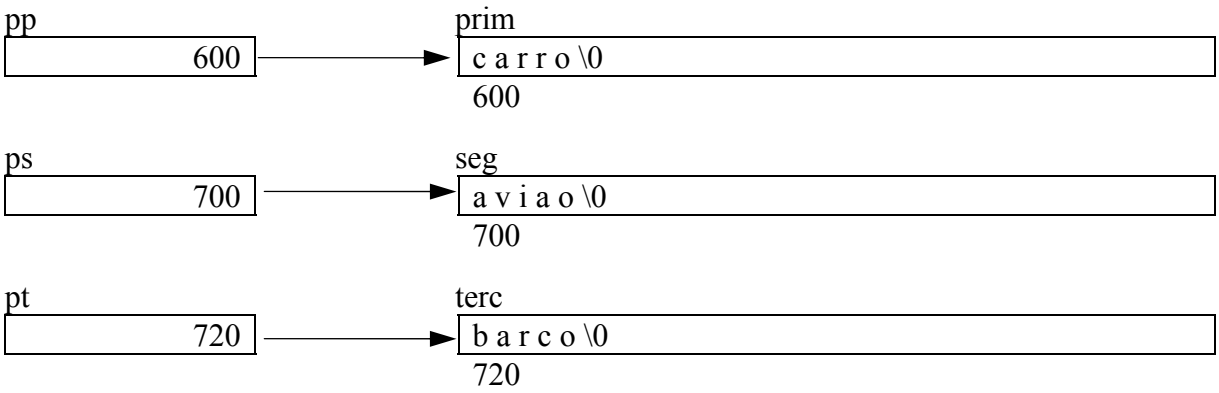
```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
 FILE *f, *g;
 char c;

 if (argc !=3){
        printf("Numero de argumentos errados. \n");
        exit(1);
 }
 f = fopen(argv[1], "r");



 if (f = = 0){
        printf("Ficheiro origem inexistente. \n");
        exit(1);
 }

 g = fopen(argv[2], "w");

 while((c = fgetc(f)) != EOF)
        fputc(c, f);

 fclose(f);
 fclose(g);
}
```

9.7 Tabelas de Apontadores

Suponha que tenho três variáveis do tipo *string* e 3 apontadores para **char**. Eu posso colocar cada um dos apontadores a apontar para uma das *strings* e posteriormente utilizar o identificador da *string* ou o apontador para aceder a essa *string*. Recordemos o que foi referido na Secção 6.5, que o nome de uma tabela corresponde ao endereço do seu primeiro elemento.

Exemplo:

```
…
char prim[10] = ”carro”;
char seg[10] = ”aviao”;
char terc[10] = ”barco”;
char *pp, *ps, *pt;

pp = prim;
ps = seg;
pt = terc;

puts(prim); /* podemos visualizar as strings */
puts(pp);
…
```



Posso declarar uma tabela de apontadores e utilizá-la para apontar para as várias *strings*.

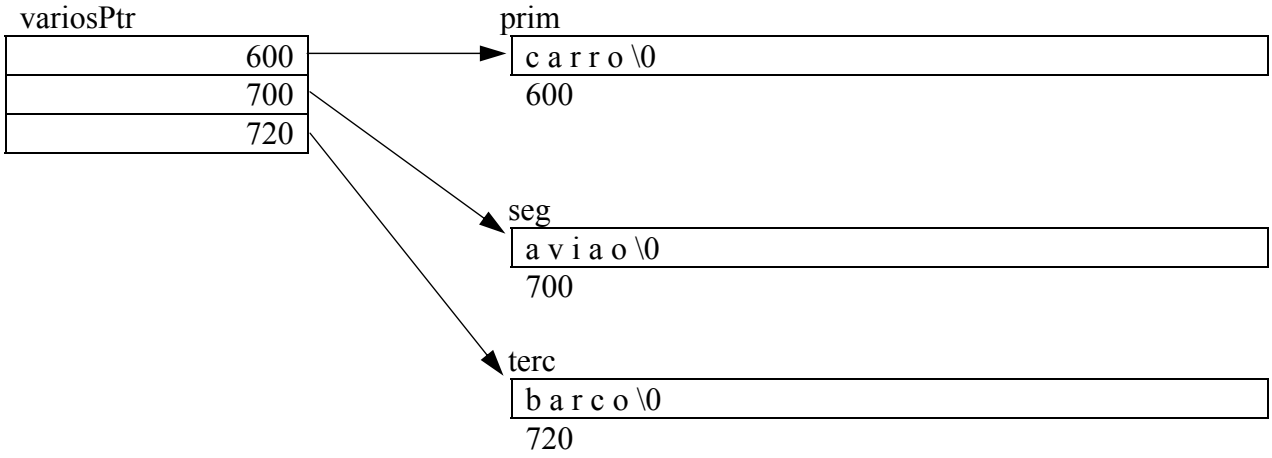
```
char *variosPtr[3];
```

Repare-se que **variosPtr** é uma tabela de 3 elementos onde cada elemento é do tipo apontador para char (ou seja, char *).

Exemplo:

```
…
char prim[10] = ”carro”;
char seg[10]  = ”aviao”;
char terc[10] = ”barco”;
char *variosPtr[3];
variosPtr[0] = prim;  /* prim contém o endereço de prim[0]*/
variosPtr[1] = seg;
variosPtr[2] = terc;
```

```
/* as duas linhas seguintes fazem o mesmo */
puts(prim);
puts(variosPtr[0]);
```



9.8 Alocação Dinâmica de Matrizes

Para alocar memória para uma matriz de **m**x**n** elementos, podemos utilizar vários métodos.

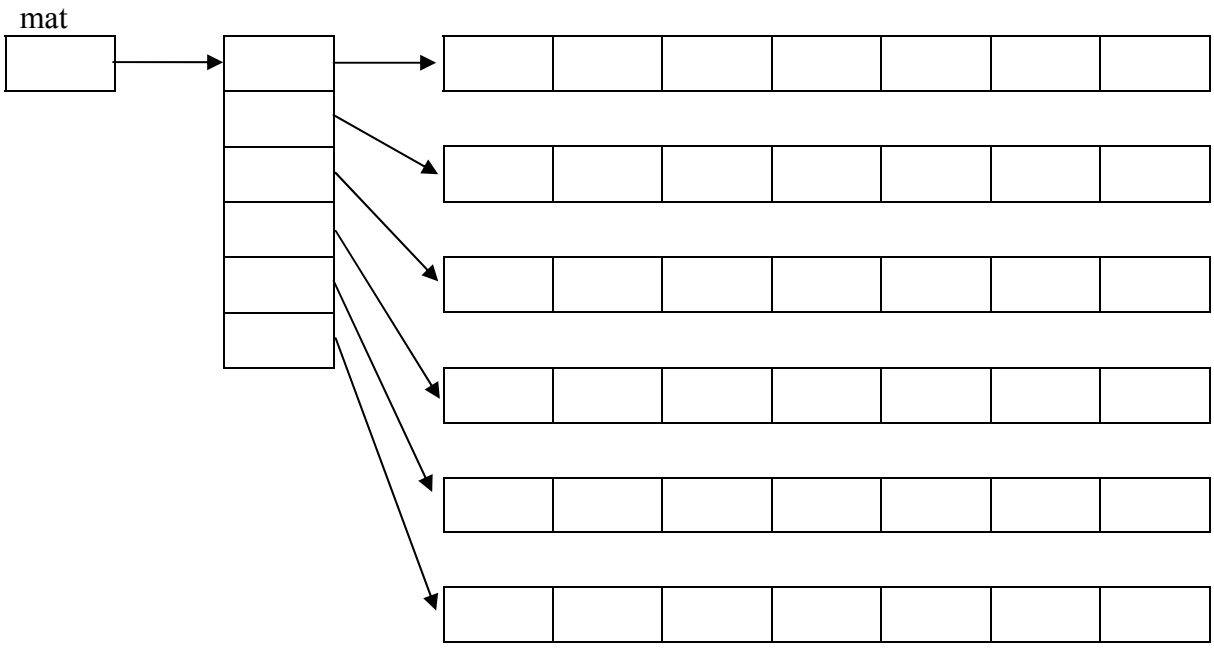
1º Método:

i) declarar uma variável do tipo apontador para apontador (para o tipo dos elementos da tabela);

i) colocar esse apontador a apontar para uma tabela de **m** linhas de apontadores (para tipo dos elementos da tabela), utilizando **malloc()** ou **calloc()**;

ii) colocar cada apontador da tabela alocada em ii) a apontar para uma tabela de **n** colunas do tipo pretendido, utilizando **malloc()** ou **calloc()**;

São necessárias (1+n) chamadas à função de alocação de memória e o mesmo número a **free()**. A matriz pode não ficar numa zona contígua de memória, pois a alocação é efectuada linha a linha. Os passos descritos no **1º método** encontram-se representados na figura seguinte:



E**xemplo 9.3:** Programa que aloca espaço de memória para uma matriz - **1º método**.

```
/* alocação de uma matriz de floats de m linhas e n colunas */

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

void main(void)
{
  float **mat;
  int m, n, i;

/*as dimensões da matriz podiam ser pedidos ao utilizador */
  m = 10;
  n = 20;
// alocar tabela de  apontadores; mat fica a apontar para primeiro elemento dessa tabela
  mat = (float **)calloc(m, sizeof(float *));
  if (mat == NULL){
          printf("Erro na alocaçao da matriz\n");
          exit(1);
  }
// cada um dos  apontadores da tabela é colocado a apontar para uma linha de n colunas
  for(i=0; i<m; i++){
          mat[i] = (float *)calloc(n, sizeof(float));
          if (mat[i] == NULL){
          printf("Erro na alocaçao da matriz\n");
                          exit(1);
          }
  }
```

```
/* aqui pode-se utilizar a matriz e para lhe aceder pode-se utilizar mat[i][j]*/
        /* ... */

/* libertar espaço ocupado */
  for(i=0; i<m; i++)
        free((float *)mat[i]);
  free((float **)mat);
}
```

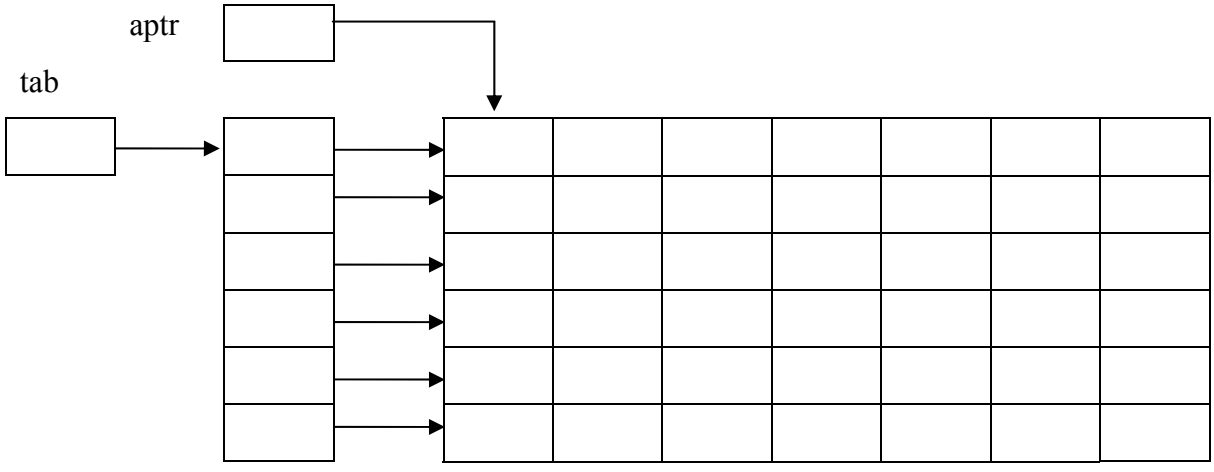
2º Método:

ii) declarar uma variável do tipo apontador para apontador (para tipo dos elementos da tabela) e colocar esse apontador a apontar para uma tabela de **m** linhas de apontadores (para tipo dos elementos da tabela), utilizando **malloc** ou **calloc**;

iii) alocar espaço para guardar toda a matriz, ficando um apontador (consideremos **aptr**) a apontar para o primeiro elemento;

iv) cada um dos apontadores alocados em **i)** é posteriormente colocado a apontar para uma linha da matriz.

Os passos descritos no 2º **método** encontram-se representados na figura seguinte:



Neste **2º método**, em primeiro lugar, aloca-se espaço para guardar toda a matriz, ficando um apontador (**aptr**) a apontar para o primeiro elemento. Depois alocam-se número de linhas apontadores para tipo_elementos e cada um desses apontadores é posteriormente colocado a apontar para uma linha da matriz.

E**xemplo 9.4:** Programa que aloca espaço de memória para uma matriz - **2º método**.

```
#include <stdio.h>
void main(void)
{
  int **tab;
  int *aptr;
/*as dimensões da matriz podiam ser pedidos ao utilizador */
  int m = 5, n = 10;
  int k;

  /* aptr aponta para uma matriz de m linhas  por n colunas */
     aptr = (int *)malloc( m * n * sizeof(int) );

/* tab aponta para uma tabela de m  apontadores para float */
   tab = (int **)malloc( m * sizeof(int *) );

/* cada elemento da tabela apontada por tab é colocado a apontar para uma linha da
matriz */
  for(k = 0; k < m; k++)
    tab[k] = aptr + ( k * n );

//aqui pode-se usar a matriz e para lhe aceder usa-se tab[i][j]
          /* ... */
  free(tab);
  free(aptr);
}
```

10. Apêndices

10.1 Utilização Avançada do Modo de Texto

No ficheiro **conio.h** encontram-se definidas várias funções que permitem utilizar todas as funcionalidades disponíveis em modo de texto: cor, códigos estendidos de tecla (acesso às teclas de funções, cursores e ALT+Tecla) e "janelas".

Definição do modo de texto

Número de linhas e de colunas em que se encontra dividido o monitor; em cada quadrícula é possível escrever um carácter. A definição do modo é feita utilizando a função **textmode()**, cujo protótipo é: **void textmode(int newmode);**

Modos de texto (número de colunas e linhas endereçáveis)

modo	**nºcolunas x nº linhas**	**Placa gráfica**
BW40	40 x 25	16 cores, níveis cinzento
C40	40 x 25	16 cores
MONO	80 x 25	2 cores (preto e branco)
BW80	80 x 25	16 cores, níveis de cinzento
C80	80 x 25	16 cores
C4350	80 x 50	16 cores

Também é possível alterar as cores utilizadas na escrita dos caracteres. As cores disponíveis numa placa gráfica a cores, valor numérico e constante simbólica, são:

0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
15	WHITE
128	*BLINK*

Em termos de cores utilizadas para a escrita dos caracteres no monitor temos a cor de fundo e a cor do carácter, podendo estas ser alteradas pelas funções **textbackground()** e **textcolor()** que possuem os protótipos:
void textcolor(int newcolor);
void textbackground(int newcolor);

Para cor de fundo podemos utilizar as 8 primeiras da lista e para cor do texto todas as da lista. Se pretendermos o texto a piscar devemos somar à cor do carácter a constante BLINK.

É possível alterar a cor de fundo e do carácter utilizando a função **textattr()**. As cores de fundo e do texto encontram-se armazenadas num byte com a seguinte disposição de bits:

7	6	5	4	3	2	1	0
B	b	b	b	f	f	f	f

onde B - blink (piscar)
b - background (fundo)
f - foreground (texto)

O protótipo desta função é: void textattr(int newattr);

Por exemplo, se pretendermos texto com cor amarela, a piscar, e sobre fundo azul temos o código: seguinte:
textattr(BLINK + 16*BLUE + YELLOW);
/ a multiplicação por 16 destina-se a deslocar os 4 da cor azul, 4 bits para a esquerda */*

o que é equivalente ao código seguinte:

textcolor(BLINK+YELLOW);
textbackground(BLUE);

Dentro do ecrã é possível definir uma área ou janela e a partir daí passar a "trabalhar" (em termos de escrita e de leitura) apenas dentro dessa janela. Isto é conseguido utilizando a função **window()**, cujo protótipo é:

void window(int left, int top, int right, int bottom);

Por omissão o modo de texto é 80x25 a cores, logo uma janela (1,1,80,25).
Em modo 80x25 o canto superior esquerdo corresponde à posição (1,1), o direito à (80,1), o canto inferior esquerdo à (1,25) e o direito a (80,25).

Podemos colocar o cursor em qualquer ponto da janela corrente utilizando a função **gotoxy()**, que tem como protótipo:

```
void gotoxy(int x, int y);
```

É possível determinar a posição do cursor utilizando para isso as funções **wherex()** e **wherey()** que determinam, respectivamente, a coluna e a linha onde se encontra o cursor num determinado instante. Protótipos:

```
int wherex(void);
int wherey(void);
```

Limpar monitor, ficando o cursor no canto superior esquerdo
```
void clrscr(void);
```

Apagar totalmente ou parcialmente uma linha

void clreol(void);	Apaga desde o cursor até ao fim da linha.
void delline(void);	Apaga a linha onde está o cursor e puxa todas as linhas abaixo desta para cima

Insere linha
Insere uma linha vazia e puxa todas as linhas a partir da linha onde se encontra o cursor para baixo.
```
void insline(void);
```

Formato do cursor
```
void _setcursortype(int cur_t);
```
onde cur_t pode tomar os seguintes valores (e significado respectivo)

_NOCURSOR	(cursor não se vê)
_SOLIDCURSOR	(rectângulo a cheio a piscar)
_NORMALCURSOR	(cursor normal; carácter de sublinhado a piscar)

Verificar se foi pressionada uma tecla:
```
int kbhit(void);
```
retorna o valor 0 se não tiver sido pressionada nenhuma tecla e uma valor diferente de zero se alguma tecla for pressionada.

Leitura de carácter
```
int getch(void);
int getche(void);
```
Estas funções permitem ler um carácter sem ser necessário carregar na tecla ENTER. g**etche()** ecoa no monitor a tecla pressionada, ao contrário de **getch()**.

No caso da tecla ser especial, a função **getch()** primeiro retorna o carácter nulo (’\0’) e de

seguida (quando novamente chamada) retorna o código estendido da tecla. Apresenta-se uma tabela resumida dos códigos estendidos:

Segundo código	Significado
59-68	F1-F10
72	Up Arrow
73	PgUp
75	Right Arrow
77	Left Arrow
79	End
80	Down Arrow
81	Page Down
83	Del
133-134	F11-F12

Funções alternativas a **printf()**, **scanf()**

```
int cprintf(char *string_formatacao, ...);
int cscanf(char *string_formatacao, ...);
```

Estas funções trabalham do mesmo modo das originais, no entanto, a função **cprintf()** para executar correctamente uma mudança de linha necessita de **\r\n**.
A vantagem da utilização destas funções **cprintf()** e **cscanf()** é a utilização imediata das cores de fundo e de carácter sempre que sejam alteradas, sem necessidade de fazer **clrscr()**.

O programa seguinte permite exemplificar o funcionamento de algumas das funções descritas anteriormente.

Exemplo 10.1: O programa seguinte permite exemplificar o funcionamento de algumas das funções descritas anteriormente. Teste este programa, observe a execução e analise o código.

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

void cores(void)
{
  int car, fundo;
  window(20,3,60,23);
  for(car=0; car<=15;car++){
    textcolor(car);
    cprintf("Cor %d - > Texto Texto Texto Texto\r\n", car);
  }
  printf("Qualquer tecla para prosseguir \n");
  getch();
  clrscr();
  for(car=0; car<=15; car++){
    textcolor(car+BLINK);
    cprintf("Texto a piscar ... ...\r\n");
  }
 printf("Qualquer tecla para prosseguir \n");
 getch();
 clrscr();
 for(car=0; car<=15; car++){
    textbackground(15-car);
    textcolor(car);
    cprintf("Texto texto TEXTO ...\r\n");
  }
 printf("Qualquer tecla para prosseguir\n");
 getch();

}
void janelas(void)
{
  int i, j;

  for(i=0; i<12; i++){
    delay(1000);
    textbackground(15-i);
    textcolor(i);
    window(2+i,2+i,79-i,24-i);
    clrscr();

    for(j=0; j<20; j++){
      cprintf("%d\n", j);
      delay(50);
    }
  }
}
```

```
void cursores(void)
{
 textmode(C80);
 textbackground(BLUE);
 textcolor(YELLOW);
 clrscr();
 cprintf("carregue em ENTER para observar os tres tipos de cursores\r\n");
 _setcursortype(_NORMALCURSOR);
 cprintf("\r\nNormal ");
   getch();
 _setcursortype(_SOLIDCURSOR);
 printf("\r\nQuadrado ");
 getch();
 _setcursortype(_NOCURSOR );
 cprintf("\r\nInvisivel");
 getch();
}

void modos(void)
{
 int i;
 textcolor(YELLOW);
 textbackground(BLUE);
 textmode(MONO);
 clrscr();
 for(i=0; i<10; i++)
   cprintf("Modo MONO 80x25 BW \n");
 printf("\nQualquer tecla para continuar…");
 getch();
 textmode(C40);
 clrscr();
 for(i=0; i<10; i++)
   cprintf("Modo C40 40x25 16 cores \n");
 printf("\nQualquer tecla para continuar…");
 getch();
 textmode(C80);
 clrscr();
 for(i=0; i<10; i++)
   cprintf("Modo C80 80x25 16 cores \n");
 printf("\nQualquer tecla para continuar…");
 getch();
 textmode(C4350);
 clrscr();
 for(i=0; i<10; i++)
   cprintf("Modo C4350 80x55 16 cores \n");
 printf("\nQualquer tecla para continuar.");
 getch();
}
```

```
void especiais(void){
  char tec;
  textbackground(BROWN);
  clrscr();
  textcolor(BLUE);
  cprintf("Carregue numa tecla qualquer…\r\n");
  tec=getch();
  if(tec == '\0'){
    cprintf("TECLA ESPECIAL\r\n");
    tec=getch();
    switch(tec){
      case 59:
      case 60:
      case 61:
      case 62:
      case 63:
      case 64:
      case 65:
      case 66:
      case 67:
      case 68:cprintf("Tecla F%d\r\n",tec-58);
              break;
      case 72:cprintf("Up Arrow\r\n");
              break;
      case 80:cprintf("Down Arrow\r\n");
              break;
      case 77:cprintf("Right Arrow\r\n");
              break;
      case 75:cprintf("Left Arrow\r\n");
              break;
      default:cprintf("Outra tecla especial\r\n");
    }
  }
    else
      cprintf("Voce carregou em %c\r\n", tec);
  cprintf("Qualquer tecla para voltar ao menu.");
  getch();
}
void erro(void){
  window(20,10,55,14);
  textbackground(BROWN);
  clrscr();
  textcolor(YELLOW+BLINK);
  textbackground(CYAN);
  gotoxy(14,2);
  cprintf("Opcao errada\r\n\r\n");
  gotoxy(2,4);
  _setcursortype(_NOCURSOR);
  textcolor(WHITE+BLINK);
  textbackground(CYAN);
  cprintf("Qualquer tecla para voltar ao menu.");
  getch();
}
```

```
void menu(void)
{
  char opc;

  do {
    textmode(C80);
    window(1,1,80,25);
    textbackground(BLUE);
    clrscr();
    textcolor(YELLOW);
    _setcursortype(_NORMALCURSOR);
    clrscr();
    cprintf("Opcoes\r\n");
    cprintf("1  - Ver cores\r\n");
    cprintf("2  - Janelas\r\n");
    cprintf("3  - Cursores\r\n");
    cprintf("4  - Modos de texto\r\n");
    cprintf("5  - Teclas especiais\r\n");
    cprintf("9  - Sair\r\n");
    cprintf("A sua opcao -");

    opc = getch();
    clrscr();
    switch(opc){
      case '1':cores();
               break;
      case '2':janelas();
               break;
      case '3':cursores();
               break;
      case '4':modos();
               break;
      case '5':especiais();
               break;

      case '9':break;
      default:erro();
    }
   } while(opc != '9');
}

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

10.2 Tabela de Precedência dos Operadores em C

Operador	**Precedência**
() [] -> .	**MAIOR**
! ~ ++ -- - (cast) * & sizeof	
* / %	
+ -	
<< >>	
< <= > >=	
== !=	
&	
^	
|	
&&	
||	
? : (*operador ternário*)	
= += -= *= /= etc...	
, (*vírgula*)	**MENOR**

Nota: Na segunda linha temos os operadores unários; assim **&** é o operador endereço e não o operador AND-bitwise.

10.3 Referência da linguagem C

Funções para I/O **stdio.h**

Genéricas sobre ficheiros

```
FILE *fopen(char *fname, char *fmode);
int fclose(FILE *fp);
int remove(char *fname);
int rename(char *fname, char *newfname);
```

Ficheiro de texto (assumido como não formatado)

```
int getc(FILE *fp);
int putc(int ch, FILE *fp);
int getchar(void);
int putchar(int ch);
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
char *fgets(char *s, FILE *fp);
int fputs(char *s, FILE *fp);
```

Ficheiro de texto formatado

```
int fprintf(FILE *fp, char *s, ...);
int fscanf(FILE *fp, char *s, ...);
int printf(char *s, ...);
int scanf(char *s, ...);
```

Ficheiro binário

```
int fread(void *s, size_t size, size_t number, FILE *fp);
int fwrite(void *s, size_t size, size_t number, FILE *fp);
int fseek(FILE *fp, long n, int orig);
long ftell(FILE *fp);
void rewind(FILE *fp);
```

size_t ⇔ long ou unsigned long; *depende do compilador*

Erros no acesso ao ficheiro

```
int feof(FILE *fp);
int ferror(FILE *fp);
```

Strings **string.h**

```
char *strcpy(char *t, char *s);
char *strncpy(char *t, char *s, int n);
char *strcat(char *t, char *s);
char *strncat(char *t, char *s, int n);
char *strcpy(char *t, char *s);
int  strcmp(char *t, char *s);
int  strncmp(char *t, char *s,int n);
```

```
char *strchr(char *t, int c);
char *strrchr(char *t, int c);
char *strstr(char *t, char *s);
size_t strlen(char *s);
char strerror(int n);
char *strtok(char *t, char *s);
void *memcopy(char *t, char *s, int n);
void *memmove(char *t, char *s, int n);
int   memcmp(char *t, char *s, int n);
void *memchr(char *t, char *s, int n);
void *memset(char *t, int c, int n);
```

Genéricas **stdlib.h**

```
char * itoa(int value, char *s, int base);
double atof(char *s);
int atoi(char *s);
long atol(char *s);
int abs(int n);
long labs(long n);
int rand(void);
void srand(unsigned int seed);
void *calloc(int nobj, size_t n);
void *malloc(size_t n);
void *realloc(void *p, size_t n);
void free(void *p);
void abort(void);
void exit(int stat);
int atexit(void(*f)(void));
int system(char *s);
typedef unsigned size_t;
typedef long fpos_t;
```

Teste de Caracteres **ctype.h**

```
int isalnum(char c);
int isalpha(char c);
int iscntrl(char c);
int isdigit(char c);
int isgraph(char c);
int islower(char c);
int isprint(char c);
int ispunct(char c);
int isspace(char c);
int isupper(char c);
int tolower(int c);
int toupper(int c);
```

Operações Matemáticas **math.h**

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double atan2(double x, double y);
double sin(double x);
double sinh(double x);
double cosh(double x);
double tanh(double x);
double exp(double x);
double log(double x);
double pow(double x, double y);
double pow10(int p);
double sqrt(double x);
double ceil(double x);
double floor(double x);
double fabs(double x);
double ldexp(double x, int e);
double frexp(double x, int *fr);
double modf(double x, double *fr);
double fmod(double x, double y);
```

11. Bibliografia

K. N. King, " C Programming – A Modern Approach", Norton & Companyjavascript:NewSearch(%22DP 2008%22), 2nd Ed., 2008

Luís Damas, "Linguagem C", FCA - Editora de Informática, 1999

António Rocha, Introdução à Programação Usando C", FCA - Editora de Informática, 2006

Pedro Guerreiro, "Elementos de Programação com C", FCA - Editora de Informática, 3ª Ed, 2006

Herbert Schildt, "Teach Yourself C", McGraw-Hill, 3rd Ed, 1998

http://publications.gbdirect.co.uk/c_book/