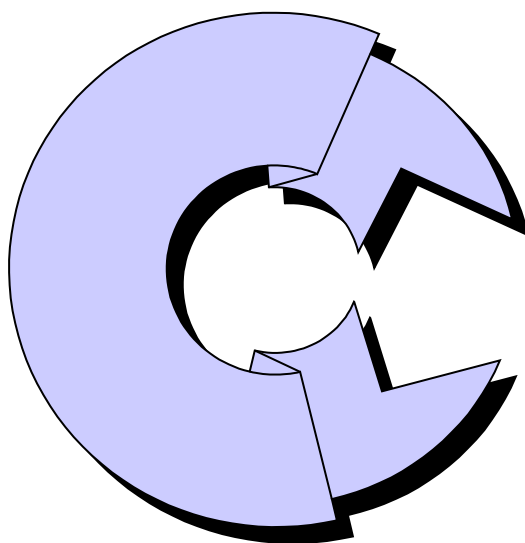


Apontamentos

Disciplina : Algoritmia e Programação (APROG)

Curso : Engenharia Eletrotécnica e de Computadores - Bolonha

Conteúdo : Introdução à Linguagem C



◆ Introdução

A linguagem **C** evoluiu de duas linguagens B e BCPL e foi desenvolvida nos *Bell Telephone Laboratories* em 1972 por Brian W. Kernighan e Dennis M. Ritchie. A linguagem C está associada ao desenvolvimento de sistemas operativos, nomeadamente o sistema UNIX, e é uma linguagem independente do hardware. Com algum cuidado na implementação de programas em C, é possível que os mesmos corram em diversas plataformas.

A rápida expansão da linguagem C levou a que surgisse muitas variantes que eram similares mas muitas vezes incompatíveis. Em 1989 foi aprovado um documento, e atualizado em 1999, que teve como objetivo criar uma versão standard do C (ISO/IEC 9899:1999).

Basicamente, um programa em C está dividido em módulos designados de funções. Todas as funções podem ser programadas por nós mas a linguagem C possui uma grande coleção de funções designadas de "C Standard Library". Assim, para programarmos em C é necessário por um lado, conhecer a linguagem propriamente dita, e por outro lado, aprender a usar as funções das bibliotecas standard do C.

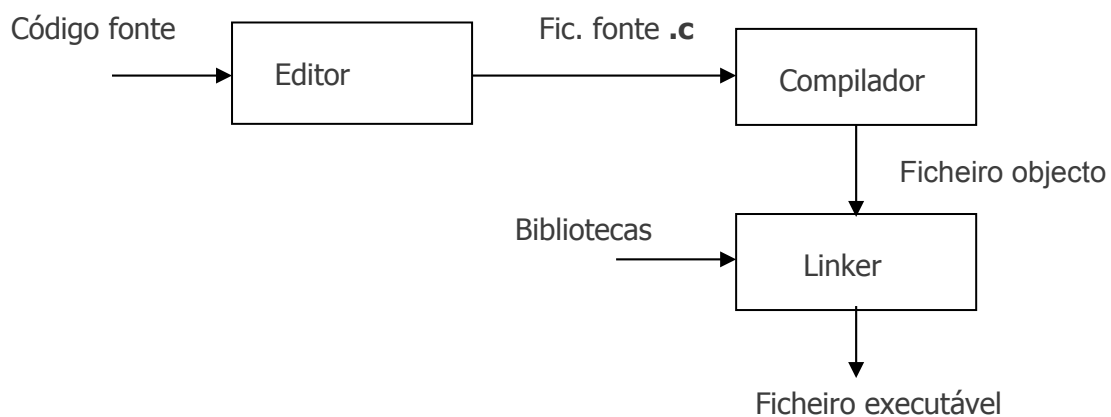
➤ **Passos para a construção de um programa em C, pronto a ser executável:**

- 1º Passo: *Edição* do ficheiro fonte. Existem vários utilitários para editar, escrever, o código fonte, como por exemplo: *vi*, *emacs* e *notepad*. Os ambientes de desenvolvimento integrados de programação são também bastante utilizados uma vez que integram uma série de funcionalidades dentro do próprio ambiente para desenvolvimento de programas. As funcionalidades fazem parte de um ambiente gráfico normalmente bastante fácil de usar. Mais importante é gravar o ficheiro fonte com a extensão **.c**.
- 2º Passo: *Compilação* do ficheiro fonte. O compilador verifica se a sintaxe das instruções de um programa está correta e traduz o programa para linguagem máquina. Se não houver erros de compilação, será então criado um ficheiro objeto, com o nome igual ao nome do programa e extensão **.obj** (em Dos) ou **.o** (em Unix). Antes de criar a linguagem máquina, o compilador invoca o pré-

processador de C para executar possíveis comandos, designadamente diretivas (iniciadas por um cardinal **#**) de pré-processamento. Estas diretivas consistem maioritariamente na inclusão de outros ficheiros no programa.

- 3º Passo: "*Linkagem*" do programa objeto. Os programas em C contêm referências a funções definidas fora do programa, nomeadamente, aquelas existentes em bibliotecas standard do C e bibliotecas privadas. O código obtido na fase de compilação possui "buracos" devido a essas partes que faltam. Nesta fase, é ligado o código produzido anteriormente com o código que falta produzindo o ficheiro executável.

Esquema geral:



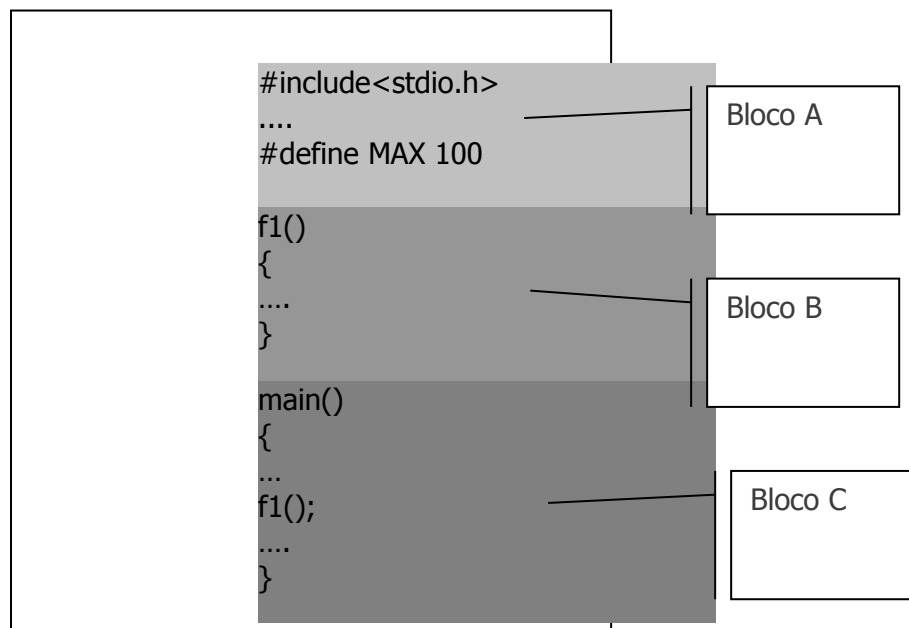
► Estrutura básica de um programa em C

Num programa em C, é possível destacar diferentes blocos de código, nomeadamente:

- Bloco de comandos que não são instruções C, tais como: inclusão de bibliotecas, diretivas de compilação e definição de constantes (na figura: **Bloco A**).
- Bloco de funções do programador (na figura: **Bloco B**). Todo código tem de ser desenvolvido e dividido em funções. O programador cria as suas próprias funções para construir um programa.
- Bloco principal: Caracteriza-se por conter a função principal do programa, a função **main()**. Esta função contém o conjunto de instruções responsáveis por

dar início à execução do programa. O código de qualquer função tem de ser escrito entre **{ }** (na figura: **Blocos B e C**).

O bloco B pode também ser colocado depois da função **main()**. Para isso, é necessário colocar os protótipos das funções, referentes ao bloco B, antes de qualquer chamada às mesmas por parte de outras funções.



Quando for compilado o programa, se o compilador apresentar um aviso com uma mensagem do tipo: *"Function should return a value"*, deve-se ao facto de se estar a usar um compilador de C++. Este aviso pode ser simplesmente ignorado, ou então, para ser eliminado, é suficiente colocar a palavra **void** (que significa *nada*) antes da função **main()**. De notar, que a função **main()** pode ser definida com um cabeçalho mais elaborado, conforme se mostra a seguir:

main(int argc, char *argv[])

A sua utilização justifica-se quando um programa precisa de receber dados, através dos parâmetros da função, que foram passados na linha comando. O parâmetro **argc** indica quantos argumentos foram passados na linha de comando (incluindo o próprio nome do programa), e **argv** é um vetor com todas as cadeias de caracteres (*strings*) passadas na linha de comando.

O programa abaixo apresentado, imprime para o ecrã a mensagem “Boas Vindas” e pretende mostrar a estrutura básica de um programa em C.

```
#include<stdio.h>

main()
{
    printf("Boas Vindas");
}
```

A função padrão **printf()**, tais como outras funções para Input/Output, encontram-se na biblioteca **<stdio.h>**. Sempre que são utilizadas num programa, é necessário adicionar uma diretiva para incluir a biblioteca de funções respetiva.

Na escrita do código fonte, a linguagem C é muito flexível na medida em que permite que qualquer instrução seja codificada numa só linha, ou dividida por diversas linhas, isto é, não tem colunas ou linhas dedicadas para se começar a escrever as instruções, como é o caso das linguagens COBOL e RPG.

No programa anterior a única instrução da função **main()** poderia ser escrita da seguinte maneira:

```
printf("Boas Vindas");
```

Terminada a execução do programa, temos a visualização da mensagem, ficando o cursor colocado a seguir ao último carácter. Se pretendêssemos, o que é muito comum, que aquele fosse posicionado na linha imediatamente a seguir à apresentação da mensagem, tínhamos de adicionar à função **printf()**, um carácter especial para esse fim. Esse carácter é **'\n'** (New Line). O programa ficaria assim:

```
#include<stdio.h>

main()
{
    printf("Boas Vindas\n");
}
```

O compilador quando encontra o símbolo '\ ' espera encontrar imediatamente a seguir a este, outro símbolo com uma função especial. A tabela abaixo, apresenta um conjunto significativo de símbolos usados na linguagem C.

\7	<i>Bell</i> (sinal sonoro do computador)
\a	<i>Bell</i> (sinal sonoro do computador)
\b	BackSpace
\n	New Line (mudança de linha)
\r	Carriage return
\t	Tabulação horizontal
\v	Tabulação vertical
\\	Carácter \
\'	Carácter ' (plica)
\"	Carácter (aspas)
\?	Carácter ? (ponto de interrogação)

Não se pode falar de um programa em C, sem se falar do carácter ponto e vírgula ';'. Este símbolo indica ao compilador o fim de uma instrução e é importante a sua colocação para se evitar erros de compilação.

► Comentários

Os comentários têm como objetivo fornecer notas e/ou esclarecimentos úteis sobre os blocos e/ou linhas de código de um ficheiro fonte. Seguindo esta prática, será certamente mais fácil e rápido, quer para quem faz o programa quer para outros que o venham a alterar, entender o que se pretende em determinado ponto do programa.

O compilador reconhece que determinada linha ou linhas de código são comentários se estas estiverem limitadas pelos símbolos '/*' e '*/'. Podem ser usados os caracteres '//' quando se pretende comentar só uma linha de código do programa.

Os comentários não /* Programa: prog01.c
são alvo de compilação. Data: 99/10/20
*/

Por exemplo:

```
....
/* Método principal da aplicação */
main() {
...
}
```

◆ **Palavras reservadas, tipos de dados básicos, constantes e operadores**

► **Palavras reservadas**

O conjunto de palavras reservadas em C é relativamente pequeno uma vez que muitas das suas capacidades encontram-se nas bibliotecas da linguagem concebidas para realizar todo tipo de tarefas.

No quadro seguinte são apresentadas as palavras reservadas da linguagem C. À medida que forem utilizadas serão dadas as explicações necessárias.

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

► **Tipos de dados**

Os dados são armazenados na memória primária do computador em estruturas primitivas (variáveis) ou em estruturas não primitivas, como por exemplo vetores e matrizes (estruturas multi-dimensionais). Em C, como na maior parte das linguagens, é necessário associar um tipo de dados à estrutura. Esta etapa é feita quando se faz a declaração da estrutura.

Os tipos de dados disponíveis para esta linguagem são cinco:

- **int** (para inteiros),
- **float** (para reais),
- **double** (para reais),
- **char** (para caracteres) e
- **(Apontador)**.

A declaração segue o seguinte formato:

tipo var1 [, var2,...,varn];

Exemplos:

```
int i;  
char c1,c2;  
float salario;
```

Notas gerais sobre as variáveis:

- Os nomes das variáveis devem ser sugestivos, isto é, os nomes devem expressar o tipo de informação ao qual se referem (ex: **soma**, **total**, **valor_mensal**,...).
- A declaração das variáveis deve ser sempre feita antes da sua utilização.
- O tipo que lhe está associado indica o **nº de Bytes** que irão ser precisos para guardar um valor nessa variável.
- A linguagem C é *Case Sensitive* o que quer dizer que diferencia letras minúsculas de maiúsculas.
- O nome de uma variável não deve ser escrito em maiúsculas e o primeiro carácter deve ser uma letra ou o símbolo '_'.

Uma variável pode ser inicializada com um valor através de uma operação de atribuição. Esta operação destrói o conteúdo atual da variável, a qual ficará com o novo valor.

A inicialização de uma variável pode acontecer também quando se faz a sua declaração.

Exemplos:

```
int soma=0;  
char c='s';  
double salario=0;
```

A atribuição pode ser feita a várias variáveis num único passo, sendo essa operação realizada da direita para a esquerda.

Exemplo:

```
soma1=soma2=soma3=0;
```

▶▶ Dados do tipo inteiro

Uma variável do tipo inteiro pode armazenar inteiros positivos e negativos. O tamanho em Bytes de um inteiro varia de arquitetura para arquitetura, sendo o valor mais comum 2 Bytes. No sentido de se poder correr um programa C, independentemente da máquina onde vai ser executado, é necessário adicionar qualificadores ao tipo de dados inteiro. Em C, existem os seguintes qualificadores:

- **short** (inteiro pequeno)
- **long** (inteiro grande)
- **signed** (inteiro com sinal)
- **unsigned** (inteiro sem sinal)

Estes qualificadores permitem “fixar” o tamanho dos valores inteiros, independentemente das características do sistema onde vão funcionar.

O quadro seguinte apresenta as várias categorias para os inteiros, o espaço que ocupam em memória e o âmbito das mesmas:

Tipo	Nº de Bytes	Valor mínimo	Valor máximo
int	2 (ou 4)	-32 768	32 768
short int	2	-32 768	32 768
long int	4	-2 147 483 648	2 147 483 647
unsigned int	2	0	65 535
unsigned short int	2	0	65 535
unsigned long int	4	0	4 294 967 295

▶▶ Escrita e leitura de inteiros

Normalmente é utilizada a função **printf()**, para escrita formatada de informação. O símbolo ‘%’ seguido de um ou mais caracteres indica o tipo de dados que irá ser mostrado. No caso dos inteiros, o carácter é o **d**.

Exemplo:

```
#include<stdio.h>

main()
{
    int n1=12;
    int n2=16;
    printf("Conteúdo de n1 = %d e de n2 = %d\n",n1,n2);
}
```

Output deste programa é: Conteúdo de n1 = 12 e de n2 = 16

A função **printf()** substitui os locais onde aparece o símbolo de formatação pelos valores das variáveis que se encontram a seguir ao primeiro argumento. A ordem destas variáveis é importante uma vez que as substituições são realizadas segundo essa ordem.

No que respeita à leitura de valores, via teclado, é normalmente utilizada a função **scanf()**. Esta função só deve conter o formato das variáveis que se pretende ler.

Exemplo:

```
....
int n3;
scanf("%d",&n3);
....
```

O primeiro argumento é uma *string* (cadeia de caracteres) e deve indicar a formatação do valor ou valores a ler. Cada valor lido deve ter uma correspondência com um símbolo de '%'. Nos argumentos seguintes da função, deve-se colocar as variáveis precedidas do operador **&** (excepto se forem *strings*). A presença deste símbolo, deve-se ao facto desta função precisar de conhecer o **endereço** onde a variável reside para poder alterá-la. Mais tarde será explicado como é realizada tal alteração.

Outro exemplo:

```
#include<stdio.h>
main()
{
    short int x,y;
    printf("Introduza dois números: ");
    scanf("%d%d",&x,&y);
    printf("O resultado de %hd+%hd =%hd\n",x,y,x+y);
}
```

O formato de leitura e escrita de variáveis inteiras **short** e **long** nas funções `scanf()` e `printf()` deve ser precedido dos prefixos **h (short)** e **l (long)**, respectivamente. Para o tipo **unsigned** deve-se substituir **%d** por **%u**.

▶▶ Dados do tipo real

A declaração de variáveis reais pode ser feita recorrendo aos seguintes tipos:

Tipo	Nº de Bytes
float	4
double	8

O formato de escrita e leitura segue as mesmas regras do tipo inteiro sendo agora utilizado o **%f** para reais **float** e **%lf** para reais **double** (long float).

Exemplo:

```
#include<stdio.h>
main()
{
    float raio, perimetro;
    double pi=3.1415927,area;
    printf("Digite o valor do raio da circunferencia:");
    scanf("%f",&raio);
    printf("Area = %lf\n Perimetro = %lf\n",pi*raio*raio,
    2*pi*raio);
}
```

▶▶ Dados do tipo carácter

Uma variável do tipo **char** contém apenas um único carácter o qual é sempre armazenado num único Byte.

Exemplo:

```
char ch='A', ch1='\n';
```

O formato de leitura e escrita para caracteres é **%c**.

Exemplo:

```
#include<stdio.h>
main()
{
    char ch;
    printf("Introduza um carácter: ");
    scanf("%c",&ch);
    printf("O carácter =%c\n",ch);
}
```

Os formatos de leitura e escrita devem estar de acordo com o tipo das variáveis, caso contrário, poderão ocorrer situações indesejáveis.

Muitas outras funções de leitura e escrita estão disponíveis na linguagem C, ficando esse estudo para mais tarde.

Resumo dos Formatos de Leitura e Escrita

Tipo	printf	scanf	Significado
char	%c	%c	Um único carácter
int	%d	%d	Um inteiro (base decimal)
int	%o	%o	Um inteiro (base octal)
int	%x ou %X	%x ou %X	Um inteiro (base hexadecimal)
short int	%hd	%hd	Um inteiro pequeno
long int	%ld	%ld	Um inteiro grande
unsigned int	%u	%u	Um inteiro positivo
unsigned short int	%hu	%hu	Um inteiro pequeno positivo
unsigned long int	%lu	%lu	Um inteiro grande positivo
float	%f ou %e ou %E	%f ou %e ou %E	Um real (e/E-notação científica)
double	%lf ou %e ou %E	%lf ou %e ou %E	Um real (e/E-notação científica)
long double	%Lf	%Lf	Real grande

► **Constantes**

Uma constante corresponde a um valor fixo, o qual não pode ser alterado ao longo da execução do programa.

A vantagem está na rápida substituição do valor da constante, que existe num só local do programa fonte, por outro valor, ficando todo o programa consistente com a nova atualização.

Existem dois tipos de definição de constantes em C:

- Através da palavra reservada **const**.
Sintaxe: **const tipo símbolo = valor;**
- Através da diretiva de pré-processamento **#define** (constante simbólica).
Sintaxe: **#define símbolo valor**

Exemplos:

```
const int n = 100;
#define N 100
```

A principal diferença entre estas duas definições relaciona-se com o seguinte: na fase de pré-processamento, o símbolo existente na diretiva **#define** é substituído, ao longo de todo o programa, pelo valor que lhe foi atribuído, enquanto que na definição **const**, a constante existe fisicamente numa determinada posição de memória durante a execução do programa.

No fim da instrução **#define** não deve ser colocado o símbolo ponto e vírgula. As declarações de constantes devem ser colocadas depois das directivas do tipo **#include**.

► Operadores

►► Operadores para dados numéricos

A tabela seguinte apresenta os operadores básicos usados nas instruções de cálculo aritmético.

Operação	Significado
+	Soma
-	Subtração
*	Multiplicação
/	Divisão inteira (os dois operandos têm de ser inteiros)
/	Divisão real (pelo menos um operando tem de ser real)
%	Módulo – Resto da divisão inteira

Operadores relacionais

A tabela seguinte apresenta os operadores básicos usados nas instruções que envolvem testes condicionais.

Operador	Designação	Exemplo	Significado
==	Igualdade	a==b	Testa se a é igual a b
>	Maior que	a>b	Testa se a é maior do que b
>=	Maior ou Igual que	a>=b	Testa se a é maior ou igual a b
<	Menor que	a<b	Testa se a é menor do que b
<=	Menor ou Igual que	a<=b	Testa se a é menor ou igual a b
!=	Diferente de	a!=b	Testa se a é diferente de b

Uma expressão que contenha um operador relacional, devolve sempre como resultado o valor lógico **Verdadeiro** (*true*) ou **Falso** (*false*).

Operadores lógicos

Esta classe de operadores permite interligar duas ou mais condições tendo como resultado dessas combinações um único valor lógico (Verdadeiro ou Falso).

Quadro resumo:

Operador	Significado	Exemplo
&&	AND (E lógico)	y<2 && y<=6
	OR (Ou lógico)	y==2 y==3
!	NOT (Negação lógica)	if (!(a==2))

Operadores de incremento e decremento

A linguagem C possui um conjunto de mecanismos que reduzem significativamente o código editado. Por exemplo, instruções do tipo:

i = i + 1;

ou

j = j - 1;

podem ser simplificadas se forem usados os operadores ++ e --, respetivamente.

As mesmas instruções podem ser reescritas da seguinte maneira:

`i++;` `/* Incremento de 1 */`

e

`j--;` `/* Decremento de 1*/`

O primeiro operador incrementa uma unidade à variável **i**, enquanto o segundo decrementa uma unidade à variável **j**.

Operador	Exemplo	Equivalente
<code>i ++</code>	<code>i++</code> ou <code>++i</code>	<code>i=i+1</code>
<code>i --</code>	<code>i --</code> ou <code>-- i</code>	<code>i=i-1</code>

Alguns operadores podem ser colocados antes ou depois da variável mas é preciso cuidado quando na mesma expressão existem outros operadores.

Por exemplo, as instruções seguintes fazem as mesmas operações, no entanto, não são equivalentes.

`y = x ++;`

e

`y = ++ x;`

No primeiro caso, o valor de **x** é atribuído a **y** e só depois a variável **x** é incrementada de uma unidade. No segundo caso, o incremento é feito em primeiro lugar e só depois é realizada a atribuição.

Analise o programa abaixo apresentado e indique os resultados que são mostrados para o ecrã juntamente com o valor das variáveis depois de realizado cada **printf()**.

```
#include<stdio.h>

main()
{
    int x,y;
    x=y=2;
    printf("x=%d e y=%d\n",++x,y--);
    printf("x=%d e y=%d\n",x,y);
}
```

▶▶ **Atribuição composta**

É também possível, em C, reduzir a quantidade de código em expressões do tipo:

$x = x + 2;$

$x = x * (3 + y);$

A **atribuição composta** facilita a escrita destas expressões, conforme se pode verificar através dos exemplos:

$x += 2;$ equivalente a: $x = x + 2;$

$x *= 3 + y;$ equivalente a: $x = x * (3 + y);$

O formato geral da atribuição composta é o seguinte:

variável operador_aritmético = expressão

Para ser possível a utilização da atribuição composta é necessário que a variável que vai ser alterada esteja à esquerda e à direita do operador de atribuição.

Operadores de atribuição composta mais usados: **+=, -=, *=, /=, %=**.

Mais alguns exemplos:

Exemplos	Significado
$z += 1$	$z = z + 1$
$z -= y + 2$	$z = z - (y + 1)$
$z *= y + 3$	$z = z * (y + 3)$

▶▶ **Operador *sizeof***

Este operador tem como objetivo informar qual a dimensão, em Bytes, de qualquer tipo de dados ou variável.

A sintaxe deste operador é a seguinte:

sizeof <expressão> ou sizeof(<tipo>)

Exemplo :

```
#include <stdio.h>

main()
{
    printf("Um char ocupa %d Bytes\n", sizeof(char));
    printf("Um int ocupa %d Bytes\n", sizeof(int));
    printf("Um float ocupa %d Bytes\n", sizeof(float));
    printf("Um double ocupa %d Bytes\n", sizeof(double));
}
```

►►► Conversão de tipos de dados

A conversão de tipos de dados é normalmente útil quando é necessário que um dado valor pertencente a um tipo de dados seja convertido para outro tipo de dados sem que, para isso, sejam criadas novas variáveis.

A conversão de tipos pode ser realizada de forma implícita ou de forma explícita.

Modo Implícito: Ocorre quando numa expressão estão envolvidas variáveis de tipos de dados diferentes. Nesta situação, o resultado é automaticamente convertido para o tipo de dados de maior amplitude, ou seja, numa expressão aritmética, o operando cuja representação em memória é de menor dimensão, é convertido para a representação com maior dimensão.

Exemplos:

int x; long int i; char c;

O resultado da expressão **x+i** torna-se do tipo long int.

Na expressão **x+c**, o resultado é do tipo int.

Modo Explícito: Neste caso, é o utilizador que informa explicitamente o compilador do *casting* (conversão) que deseja ver efetuado. Se o valor resultante da conversão passar para um tipo superior diz-se que se trata de uma **promoção**, caso contrário, diz-se **despromoção**.

Exemplo prático:

```
#include<stdio.h>

main()
{
    int n;
    printf("Digite um valor inteiro (0-255): ");
    scanf("%d",&n);
    printf("Digitou o nº %d cujo carácter = %c\n",n, (char) n);
}
```

Todos os caracteres possuem um código, designado de código ASCII (*American Standard Code for Information Interchange*) e na linguagem C todos os caracteres são tratados como valores inteiros.

Neste programa é lido um valor inteiro (entre 0 e 255, porque é o intervalo de códigos onde se incluem todos os caracteres visíveis) e imprime esse número. Paralelamente, é mostrado o carácter que corresponde a esse valor inteiro. Nesse sentido, foi necessário colocar explicitamente o *casting* (char) antes da variável **n** para ser mostrado o carácter correspondente ao inteiro lido. Resultado do programa:

```
Digite um valor inteiro (0-255): 65
```

```
Digitou o nº 65 cujo carácter = A
```

De salientar que a utilização da função **scanf()** com formatos inadequados pode levar a erros muito graves. Como referido anteriormente, os formatos de leitura introduzidos no primeiro argumento desta função, informam o compilador do número de bytes necessários para guardar os valores das variáveis. Se, por exemplo, tivermos um formato do tipo **%c** e em seguida for lido um inteiro, como consequência haverá perda de informação porque os tamanhos não correspondem.

Analise o seguinte programa:

```
#include<stdio.h>

main()
{
    int n=999;
    printf("Digite um carácter: ");
    scanf("%c", &n);
    printf("O conteúdo de n=%d corresponde ao carácter=%c\n",
          n, (char)n);
}
```

Output:

Digite um carácter: C

O conteúdo de n=835 corresponde ao carácter=C

Qual a explicação que dá para este resultado?

▶▶▶ **Precedência de operadores**

A precedência de operadores identifica a ordem pela qual as operações envolvidas numa dada expressão são executadas. O compilador precisa de saber sem qualquer ambiguidade qual a operação a executar, por isso, todos os operadores precisam de ter associado um atributo que identifica a precedência.

Em expressões aritméticas, a multiplicação e a divisão têm a mesma precedência mas a adição já tem menor precedência do que estas últimas, por isso a expressão:

$c = 2 + 5 * a$ é equivalente a termos $c = 2 + (5 * a)$.

Quando numa mesma expressão existem operadores com a mesma precedência, o compilador resolve a expressão segundo a associatividade atribuída a esses operadores. Este atributo diz respeito ao sentido (esquerda para a direita ou direita para a esquerda) da execução das operações.

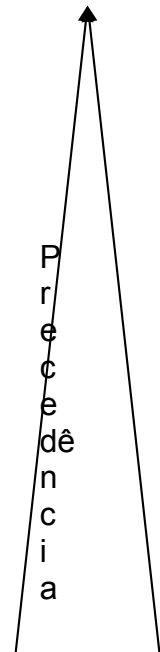
Exemplos:

A expressão: $2+3+5$ é equivalente a: $(2 + 3) + 5$, por isso, o operador $+$ é associativo à esquerda.

Já a expressão: $a=b=c$ é equivalente a: $a=(b=c)$ e por isso, o operador $=$ é associativo à direita.

Finalmente, a tabela abaixo apresentada, mostra a precedência e respectiva associatividade de vários operadores.

Operador	Associatividade
$() [] -> .$	\rightarrow
$! \sim ++ -- -(unário) +(unário)$ $*(apontado) \& sizeof$	\leftarrow
$* / \%$	\rightarrow
$+ -$	\rightarrow
$<< >>$	\rightarrow
$< <= > >=$	\rightarrow
$== !=$	\rightarrow
$\&$	\rightarrow
\wedge	\rightarrow
$ $	\rightarrow
$\&\&$	\rightarrow
$ $	\rightarrow
$? : (operador ternário)$	\leftarrow
$= += -= *= /= \% = <<= >>= \&=$ $\wedge= =$	\leftarrow
$,$	\rightarrow



Mecanismo Condicional

Instrução de decisão: *if*

A resolução de problemas não se reduz à execução de uma sequência de tarefas, previamente pensada. Muitas vezes, é preciso que os algoritmos tomem decisões. Deste modo, é possível dizer que o fluxo de execução de um determinado programa não é previamente conhecido.

Em programação, o mecanismo condicional mais utilizado é do tipo: **Se... Então... Senão** sendo a sua tradução para C feita da seguinte forma:

<code>if (condição*)</code>		<code>if (condição*)</code>
<code> instrução1;</code>		<code> { Bloco de Instruções;}</code>
<code>[else instrução2;]</code>	<code>ou</code>	<code>[else { Bloco de Instruções;}]</code>

Os parênteses retos, `[]`, significam que a instrução é opcional. As chavetas, `{ }`, marcam o início e o fim do bloco de instruções a realizar, sendo só necessário colocá-las quando existem duas ou mais instruções a executar, para qualquer uma das alternativas da instrução **if**. Quanto à condição, pode ser uma só condição ou então várias condições ligadas pelos operadores lógicos (`&&`, `|`).

Em C não existe nenhum tipo específico de dados para armazenar valores lógicos. O 0 (zero) representa o valor lógico FALSO, tudo aquilo diferente de 0 representa o valor lógico VERDADEIRO. Deste modo, é possível que a condição de teste seja substituída por uma variável, ou uma constante ou uma outra expressão relacional simples ou composta.

Uma expressão que contenha um operador relacional devolve sempre como resultado um valor lógico.

Através de alguns casos práticos, são em seguida apresentadas estas questões e também estudados mais alguns aspetos importantes na codificação de **ifs** simples e encaixados.

O primeiro caso prático, é um programa que lê dois valores do tipo inteiro e mostra para o ecrã o maior dos dois.

```
#include<stdio.h>
main()
{
    int x,y;
    printf("Introduza dois N°s");
    scanf("%d %d",&x,&y);
    if (x>y) {
        printf("O maior valor =%d\n",x); }
    else {
        printf("O maior valor =%d\n",y);}
}
```

De notar, que as chavetas existentes na estrutura condicional são redundantes uma vez que, quer a alternativa verdadeira quer a alternativa falsa só têm uma instrução a realizar.

O segundo caso prático, é idêntico ao programa anterior mas agora para três valores do tipo inteiro.

```
#include<stdio.h>
main()
{
    int x,y,z,maior;
    printf("Introduza três N°s");
    scanf("%d %d %d",&x,&y,&z);
    if (x>y)
        if(x>z)
            maior=x; /* x>y,x>z */
        else
            maior=z; /* z>x>y */
    else if (y>z)
        maior=y; /* y>x,y>z */
    else
        maior=z; /* z>y>x */

    printf("O maior valor =%d\n",maior);
}
```

O terceiro caso prático, é uma versão para o algoritmo anterior, e pretende mostrar como ligar duas condições no mecanismo condicional **if**.

```
#include<stdio.h>
main()
{
    int x,y,z,maior;
    printf("Introduza três N°s");
    scanf("%d %d %d",&x,&y,&z);
    if (x>y && x>z)
        maior=x;
    if (y>x && y>z)
        maior=y;
    if (z>x && z>y)
        maior=z;
    printf("O maior valor =%d\n",maior);
}
```

O quarto caso prático, é um exercício clássico que implementa um algoritmo de cálculo das raízes reais de uma equação do 2º grau.

```
#include<stdio.h>
#include<math.h>
main()
{
    float a, b, c, d, r, i;
    printf("Indique o coeficiente A:");    scanf("%f",&a);
    printf("Indique o coeficiente B:");    scanf("%f",&b);
    printf("Indique o termo independente C:"); scanf("%f",&c);
    printf("a=%f\nb=%f\nc=%f\n", a, b, c);
    if (a==0)
        printf("Não é equação do segundo grau!!\n");
    else
    {
        d = b*b-(4*a*c);
        if (d<0)
            printf("Não tem raízes reais!!\n");
        else if (d==0)
            printf("x1=x2=%.3f\n",-b/(2*a));
        else {
            printf("x1=%.3f\n", (-b+sqrt(d))/(2*a));
            printf("x2=%.3f\n", (-b-sqrt(d))/(2*a));
        }
    }
}
```

Algumas notas referentes a este último programa:

- O operador `==` é o operador relacional de igualdade;
- O `else` referente ao `if (a==0)` tem de ter obrigatoriamente chavetas, dado que no seu interior tem duas instruções (principais): uma atribuição e um mecanismo condicional;
- A função *built-in* **`sqrt()`**, existente na biblioteca `<math.h>` calcula a raiz quadrada de um número;
- Relembrar que a formatação `"%.3f"`, tem como objetivo mostrar, para o ecrã, um número de vírgula flutuante até à terceira casa decimal.

O quinto caso prático, implementa uma calculadora com as operações básicas de cálculo aritmético.

```
#include<stdio.h>
main() {
    float oper1,oper2;
    char op;
    // Leitura dos operandos
    printf("\nDigite o primeiro operando: ");
    scanf("%f",&oper1);
    printf("Digite o segundo operando: ");
    scanf("%f",&oper2);
    fflush(stdin);
    printf("\nEscolha um dos seguintes operadores: +,-,*,/ ");
    scanf("%c",&op);
    // Calcula e mostra o resultado
    if (op=='+')
        printf("\n %f + %f = %f",oper1,oper2,oper1+oper2);
    else if (op=='-')
        printf("\n %f - %f = %f",oper1,oper2,oper1-oper2);
    else if (op=='*')
        printf("\n %f * %f = %f",oper1,oper2,oper1*oper2);
    else
        if (op=='/')
            if (!oper2)
                printf("\nDivisor igual a zero!");
            else
                printf("\n %f / %f = %f",oper1,oper2, oper1/oper2);
        else
            printf("\nErro na escolha do operador\n");
}
```


Em C, existe uma outra instrução que pode implementar o mecanismo condicional *if...else...*. Trata-se da instrução **switch** a qual é apresentada a seguir.

- **Instrução *switch***

A instrução **switch** é uma boa alternativa à instrução **if...else...** principalmente quando o número de situações a testar é particularmente elevado.

A sintaxe geral da instrução **switch** é a seguinte:

```
switch (expressão)
{
    case constante1: {bloco de instruções; break;};
    case constante2: {bloco de instruções; break;};
    .....
    case constanten: {bloco de instruções; break;};
    [default: instrução;]
}
```

expressão pode ser uma simples variável ou uma expressão mais complexa, por exemplo, uma fórmula, cujo conteúdo/resultado deve pertencer a um dos seguintes tipos: **char**, **int** ou **long**. Dependendo do seu valor, o fluxo de controlo é desviado para o **case** respetivo.

Se nenhuma alternativa for satisfeita então são executadas as instruções que se seguem à opção de **default**. Esta opção é opcional.

constante_i representa um valor do domínio de *expressão*.

À direita de cada *case*, é colocado o conjunto de instruções que deve ser realizado no caso de se verificar esse caso concreto. O conjunto pode ser vazio.

A instrução **break**, colocada depois do bloco de instruções, não é obrigatória, mas é necessária para se implementar o mecanismo *Se ... else...* na instrução *switch*. A sua função é fazer com que o fluxo de execução passe diretamente para o exterior da instrução *switch* quando o primeiro *case* verdadeiro for executado, evitando assim, serem realizadas as instruções dos *cases* que vêm a seguir.

Para finalizar esta secção, é agora apresentada uma versão do programa da calculadora mostrando a fácil escrita de código com recurso ao mecanismo **switch**.

```
#include<stdio.h>
main()
{
    float oper1,oper2;
    char op;
    // Leitura dos operandos
    printf("\nDigite o primeiro operando: ");
    scanf("%f",&oper1);
    printf("Digite o segundo operando: ");
    scanf("%f",&oper2);
    fflush(stdin);
    printf("\nEscolha um dos seguintes operadores: +,-,*,/  ");
    scanf("%c",&op);
    // Calcula e mostra o resultado
    switch (op)
    {
        case '+' : {printf("\n %.3f + %.3f = %.3f\n",
            oper1,oper2,oper1+oper2);break;}
        case '-' : {printf("\n %.3f - %.3f = %.3f\n", oper1,oper2,oper1-
            oper2);break;}
        case '*' : {printf("\n %.3f * %.3f = %.3f\n",
            oper1,oper2,oper1*oper2);break;}
        case '/' : { if (!oper2) printf("\nDivisor igual a zero!"); else
            printf("\n %.3f / %.3f = %.3f",oper1,oper2, oper1/oper2);break;}
        default : printf("\nErro na escolha do operador\n");
    }
    system("pause");
}
```

Algumas notas:

- A instrução **fflush(stdin)** permite limpar os caracteres que estejam no *buffer* do teclado e que não interessem para leituras seguintes.
- A variável *op* é do tipo *char*. Cada *case* precisa de colocar entre plicas, e não entre aspas, o carácter que vai ser comparado com aquele guardado em *op*.

◆ Mecanismos de Repetição - Ciclos

Nas aplicações é muito frequente, uma ou uma série de instruções ter de ser repetida para a concretização de uma determinada tarefa. Nesse sentido, os mecanismos de repetição constituem o meio mais apropriado para o fazer.

A linguagem C prevê três mecanismos de controlo de fluxo com recurso a ciclos. Os formatos gerais são os seguintes:

- **Ciclo While**

```
while (condição)
    instrução;           ou           while (condição)
                                { bloco de instruções};
```

- **Ciclo For**

```
for(inicialização;condição;pós-instrução)
    instrução; | {bloco de instruções};
```

- **Ciclo Do...While**

```
do
    instrução; | { bloco de instruções};
while (condição);
```

No primeiro e segundo formatos, a condição é avaliada e se tiver sucesso então o fluxo de execução prossegue para a primeira instrução dentro do ciclo. No último formato, a instrução ou instruções, dentro do ciclo, são executadas pelo menos uma vez. Depois da primeira execução, é então avaliada a condição de ciclo e se o resultado for verdadeiro então o programa repete novamente as instruções existentes no interior do ciclo.

O ciclo **for** merece mais alguns comentários. Dentro dos parêntesis, é possível identificar três partes separadas por ponto e vírgula (;). Na primeira parte, existem normalmente instruções para inicializar variáveis, nomeadamente, a variável que controla o número de vezes que o ciclo é executado. Estas instruções só são executadas uma única vez durante a execução do **for**. Na segunda parte, define-se a condição (ou condições) de teste necessária para controlar o número de vezes que o ciclo é executado. Finalmente, na terceira parte, são colocadas as instruções pós-iteração (pós-ciclo), designadamente aquelas que controlam o andamento do ciclo. Estas instruções são realizadas após as instruções dentro do ciclo terem sido executadas. Quando o resultado da condição de ciclo for falsa, as instruções pós-iteração não são realizadas e o programa prossegue para a próxima instrução depois do **for**.

Exemplo:

```
...  
i=1;  
j=100;  
while (i<=j)  
{  
    printf("%d\n", i);  
    i++;  
    j--;  
}  
...
```

Ambos os segmentos de código são equivalentes.

```
...  
  
for(i=1, j=100; i<=j; i++, j--)  
    printf("%d\n", i);
```

As chavetas só precisam de ser colocadas se existirem duas ou mais instruções dentro do ciclo. Conforme referido anteriormente, apesar das instruções estarem dispostas em formato “embutido”, para a linguagem C este formato não tem qualquer significado.

Para concretizar, são apresentados alguns exemplos de algoritmos e/ou programas que implementam variados mecanismos de repetição.

O primeiro exemplo, é um programa que calcula e mostra a tabuada de multiplicação de um dado número (**num**), devendo ser este valor maior que zero.

```
#include<stdio.h>

main()
{
    int i, num;
    // Introdução do valor
    do
    {
        printf("\n Digite um valor inteiro positivo: ");
        scanf("%d", &num);
    } while(num<1);
    // Exibe a tabuada
    printf("\a\n    Tabuada do %d\n", num);
    for(i=1; i<=10; i++)
        printf("\n%2d * %2d = %3d", i, num, i*num);

    printf("\n\n");
    system("pause");
}
```

Os dois próximos segmentos de código são versões do programa anterior, no que se refere a possíveis implementações de mecanismos de repetição.

```
...
    printf("\a\n    Tabuada do %d\n", num);
    i=1;
    while(i<=10)
    {
        printf("\n%2d * %2d = %3d", i, num, i*num);
        i++;
    }
    printf("\n\n");
...

```

```
...
    printf("\a\n    Tabuada do %d\n", num);
    i=1;
    do
    {
        printf("\n%2d * %2d = %3d", i, num, i*num);
        i++;
    } while(i<=10);
    printf("\n\n");
...

```

Os algoritmos seguintes e respetivas traduções para C, dizem respeito ao cálculo do fatorial de um número.

Algoritmo Fatorial. Este algoritmo calcula o fatorial de um valor pedido ao utilizador. O valor lido deve ser do tipo inteiro e superior ou igual a zero.

```

10[ Ler o número]
    Ler(N)
20[Inicializar variáveis]
    PRODUTO ← 1
    MULT ← N
30[Calcular o fatorial]
    Se N ≠ 0
        Então Repetir enquanto MULT >=1
            PRODUTO ← PRODUTO * MULT
            MULT ← MULT -1
40[Imprimir resultados]
    Escrever('O fatorial de ',N,'=',PRODUTO)
50[Termina]
    Saída

```

Uma tradução possível para C:

/* Programa Fatorial. Este programa calcula o fatorial de um valor pedido ao utilizador. O valor lido deve ser do tipo inteiro e superior ou igual a zero. */

```

#include <stdio.h>

main()
{
    int n,mult;
    long int produto=1;
    printf("\nDigite um número : ");
    scanf("%d",&n);
    mult=n;
    if (n!=0)
        while (mult>=1)
        {
            produto = produto * mult;
            mult = mult -1;
        }
    printf("\nO fatorial de %d=%ld",n,produto);
}

```

Outra versão em pseudocódigo:

Algoritmo Fatorial. Este algoritmo calcula o fatorial de um valor pedido ao utilizador. O valor lido deve ser do tipo inteiro e superior ou igual a zero. (versão 1.0)

```

10[ Ler o número]
    Ler(N)
20[Inicializar variáveis]
    PRODUTO ← 1
30[Calcular o fatorial]
    Se N ≠ 0
        Então Repetir para MULT =N até 1 Passo -1
            PRODUTO ← PRODUTO * MULT
40[Imprimir resultados]
    Escrever('O fatorial de ',N,'=',PRODUTO)
50[Termina]
    Saída

```

Uma tradução possível para C:

```

/* Programa Fatorial. Este programa calcula o fatorial de um valor
pedido ao utilizador. O valor lido deve ser do tipo inteiro e superior
ou igual a zero. (versão 1.0) */

#include <stdio.h>

main()
{
    int n;
    long int produto=1;
    printf("\nDigite um número : ");
    scanf("%d",&n);
    if (n!=0)
        for(int mult=n;mult>=1;mult--)
            produto = produto * mult;
    printf("\nO fatorial de %d=%ld",n,produto);
}

```

O próximo programa lê uma sequência de valores inteiros, terminada por **-1**, e calcula, para cada valor da sequência, o fatorial respetivo.

Algoritmo Fatorial. Este algoritmo lê uma série de valores de entrada, calcula e imprime o fatorial de cada um. O valor '-1' termina a entrada de valores. Os valores lidos devem ser do tipo inteiro.

```

10[ Ler o primeiro número]
    Ler(N)
20[Iniciar ciclo externo: finaliza com -1]
    Repetir até ao passo 70 enquanto N ≠ -1
30[Inicializar variáveis]
    PRODUTO ← 1
40[Calcular o fatorial para o valor lido]
    Se N ≠ 0
        Então Repetir para MULT =N até 1 Passo -1
            PRODUTO ← PRODUTO * MULT
50[Imprimir resultados]
    Escrever('O fatorial de ',N,'=',PRODUTO)
60[Ler o próximo número]
    Ler(N)
70[Termina]
    Saída

```

Uma tradução possível para C:

```

/* Programa Fatorial. Este programa lê uma série de valores de
entrada, calcula e imprime o fatorial de cada um. O valor '-1' termina
a entrada de valores. Os valores lidos devem ser do tipo inteiro. */
#include <stdio.h>
main()
{
    int n;
    long int produto;
    printf("\nDigite um número : ");
    scanf("%d",&n);
    // Repete enquanto não for introduzido o valor -1
    while (n!=-1)
    {
        produto=1;
        if (n!=0)
            for(int mult=n;mult>=1;mult--)
                produto = produto * mult;
        printf("\nO fatorial de %d=%ld",n,produto);
        printf("\nDigite um número : ");
        // Leitura do próximo valor para o cálculo do fatorial
        scanf("%d",&n);
    }
}

```

De notar que o ciclo exterior é responsável pela leitura da sequência de números até ser digitado -1, devendo ser realizado o cálculo do fatorial para cada valor lido.

A instrução `scanf("%d",&n);` deve ser realizada antes do ciclo **while** e depois dentro do ciclo. Antes do ciclo, serve para prever a situação do utilizador digitar o valor -1 da primeira vez que lhe é solicitado um valor. Neste caso, a condição falha e as instruções dentro do ciclo não são executadas. Dentro do ciclo, o objetivo é o programa poder avançar para a leitura do valor seguinte.

Para terminar, é apresentado um exemplo clássico de um menu interativo, a partir do qual, o utilizador pode interagir com o programa, escolhendo iterativamente opções até selecionar a opção para terminar o programa.

```
/* Programa Menu - Modelo geral*/
#include<stdio.h>
main()
{
    short int op;
    do
    {
        printf("\tMenu \n");
        printf("\n\n\t\t 1- Opcao X");
        printf("\n\n\t\t 2- Opcao Y");
        printf("\n\n\t\t 3- Opcao W");
        printf("\n\n\t\t 4- Opcao Z");
        printf("\n\n\t\t 0 -Sair");
        printf("\n\n\t Digite a sua opção: ");
        scanf(" %hd",&op);
        switch(op)
        {
            case 1: printf("\nEscolheu a opcao 1"); break;
            case 2: printf("\n Escolheu a opcao 2"); break;
            case 3: printf("\n Escolheu a opcao 3"); break;
            case 4: printf("\n Escolheu a opcao 4"); break;
            default : printf("\nOpção inválida !");
        }
    }while (op!=0);
}
```

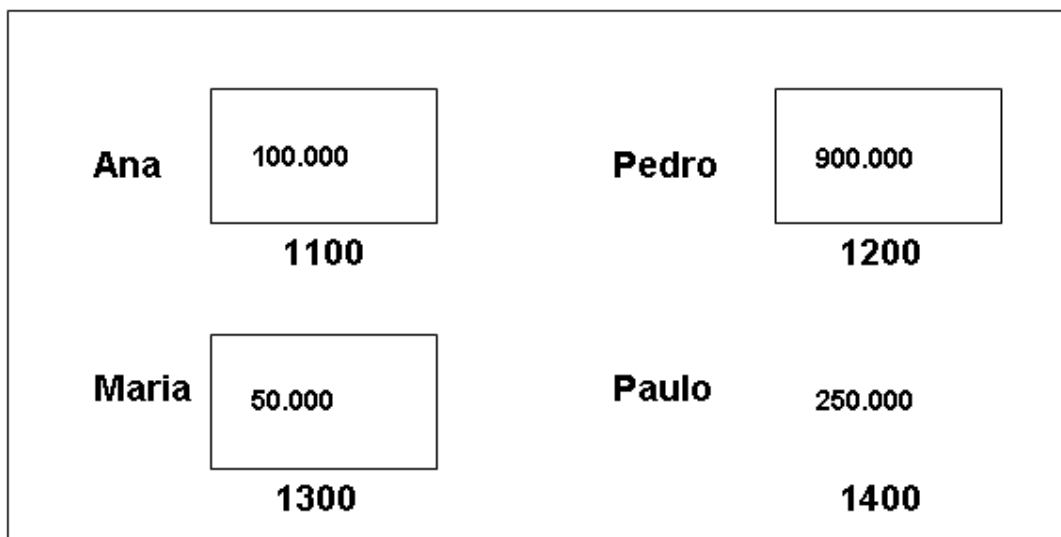
Noção de *Apontador*

Introdução

Em termos práticos, um apontador é uma variável que guarda um **endereço** de memória, ao invés de outros tipos de dados, tais como: tipo inteiro, real, carácter. A linguagem C proporciona o acesso a endereços de memória, fornecendo, deste modo, uma grande flexibilidade no manuseamento de dados, tirando também vantagens ao nível de poupança de recursos. Em contra partida, a programação perde em termos de declaratividade uma vez que os programas são de leitura mais difícil. Contudo, à medida que o utilizador tem mais experiência no manuseamento de apontadores, a falta de alguma declaratividade é remetida normalmente para segundo plano.

Para um melhor enquadramento neste assunto tão importante na programação em C, é apresentado de seguida uma situação fictícia, a qual pretende realçar o conceito de endereço e de apontador.

No Banco “Confiança Total” existem diversas contas: a conta da **Ana**, do **Pedro**, da **Maria**, do **Paulo**, entre outras.



Cada conta tem um saldo e um número. Com base no esquema anterior, tem-se:

- Conta nº 1100 da **Ana** tem um saldo de 100 000;
- Conta nº 1200 do **Pedro** tem um saldo de 900 000;

etc.

Através do número da conta é possível aceder à mesma, para assim se realizar variadas operações.

Por analogia:

- **Ana, Pedro, Maria e Paulo** são quatro variáveis que armazenam valores do tipo real;
- 1100, 1200, 1300 e 1400 são os endereços de memória das variáveis.

Para mostrar os saldos de cada uma das contas, é suficiente escrever instruções do tipo:

```
printf("Saldo %d", Ana);
printf("Saldo %d", Pedro);
.....
```

O número da conta, ou por analogia, o endereço de uma variável, atribuído a cada conta, pode ser acedido através do operador **&**. Este operador unário significa "**endereço de**" o qual permite conhecer o endereço de uma variável.

Comparativamente,

```
printf("Num. Conta: %d", &Ana); /* Num. Conta: 1100*/
printf("Num. Conta: %d", &Pedro); /* Num. Conta: 1200*/
.....
```

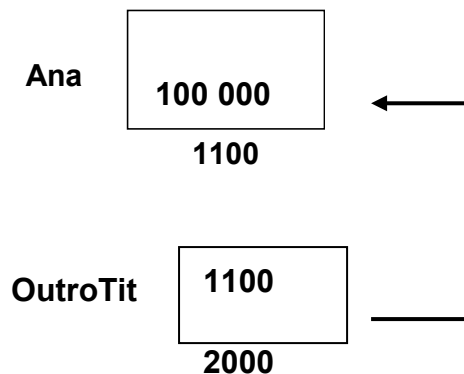
Enquanto que:

```
printf("%d", Ana); /* Saldo: 100.000 */
printf("%d", Pedro); /* Saldo: 900.000 */
.....
```

Prosseguindo com as contas bancárias, é normal uma conta ter mais do que um titular. Cada titular da conta precisa de conhecer o número da conta podendo haver vários titulares com acesso à mesma conta.

Na programação, para que o conteúdo de uma variável possa ser acedido por duas ou mais variáveis, é necessário que estas guardem o endereço da variável que tem o conteúdo. Nesse sentido, estas variáveis têm de ser do **tipo apontador**.

Na figura, a variável **OutroTit** armazena o endereço (1100) da variável **Ana** passando, assim, a poder manipular o conteúdo da variável **Ana**. Desta forma, a variável **OutroTit** precisa de ser declarada do tipo apontador.



Diz-se que **OutroTit** é um apontador que “aponta” para a variável **Ana** a qual é do tipo inteiro (saldo da conta).

- **Declaração de apontadores**

A declaração de um apontador segue a seguinte sintaxe:

```
tipo * ptr;
```

Onde:

- **tipo**: Define o tipo da variável para a qual **ptr** apontará;
- **ptr**: Define o nome do apontador;
- *****: Indica que é uma variável do tipo apontador.

Exemplos:

```
char *p;
int *idade;
int *OutroTit;
```

Conforme referido anteriormente, o endereço de uma variável é obtido através do operador **&**. Deste modo, a instrução:

```
OutroTit = &Ana;
```

inicializa a variável **OutroTit** ao endereço da variável **Ana**.

O **apontado pelo apontador**, isto é, o valor da variável cujo endereço está armazenado no apontador, é obtido colocando o símbolo ***** antes do nome do apontador.

Por exemplo, a instrução:

```
printf("Saldo da Conta: %d", *OutroTit);
```

mostra o saldo da conta da Ana, tendo em conta que **OutroTip** foi inicializado ao endereço de **Ana**.

A tabela abaixo, mostra a informação possível de se extrair em relação, por exemplo, à conta da **Ana**:

Expressão	Valor	Descrição
Ana	100000	Saldo da conta
&Ana	1100	Identificação da conta (endereço)
OutroTit	1100	Identificação da conta (valor)
&OutroTit	2000	Identificação de OutroTit (endereço)
*OutroTit	100000	Saldo da conta (valor Apontado por OutroTit)

Resumidamente, um apontador é uma variável apropriada para armazenar um endereço de uma outra variável. O *apontado de um apontador* refere-se ao conteúdo de uma variável cujo endereço foi utilizado para inicializar o primeiro.

- **Inicialização automática de apontadores**

Os apontadores devem ser sempre inicializados. Se na declaração do apontador ainda não for conhecido o endereço da variável para a qual aponta, então deve ser inicializado a **NULL** (constante da linguagem C).

Exemplos:

```
int *p = NULL;
int n= 3;
int *ptr=&n;
```

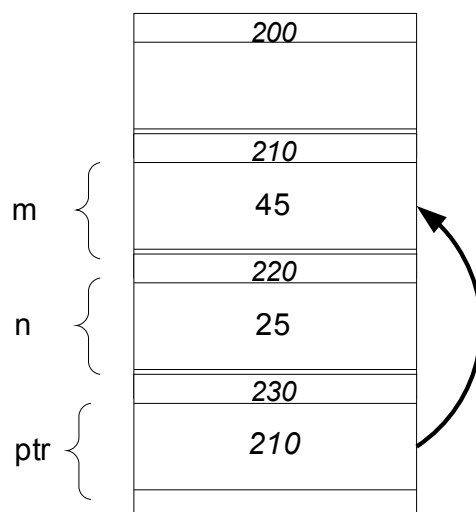
De notar que na última instrução é guardado o endereço da variável **n** na variável **ptr** (apontador). O símbolo '*' declara um apontador e, na declaração não significa "apontado de".

A constante simbólica **NULL** indica que o apontador não aponta, pelo menos por enquanto, para nenhuma variável.

Com base nas seguintes instruções:

```
int m=45, n=25;
int *ptr=NULL;
ptr=&m;
```

o esquema abaixo, ilustra uma representação em memória destas variáveis cujos endereços (em decimal) são: 200, 210, 220 e 230,

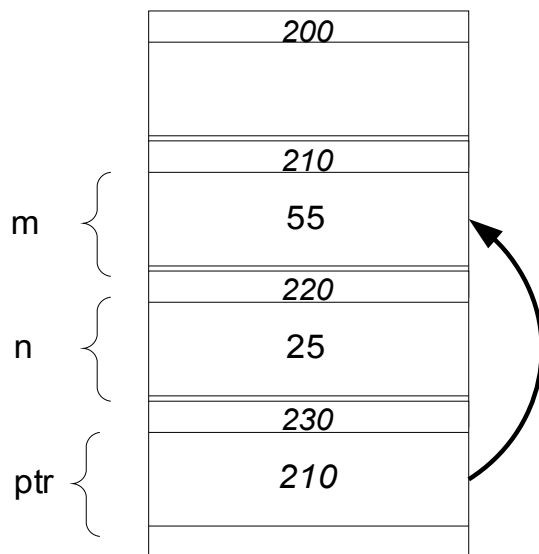


Significado:

Expressão	Valor
m	45
&m	210
ptr	210
&ptr	230
*ptr	45

Para alterar o valor da variável **m** para 55, pode-se usar qualquer uma das seguintes instruções:

```
m=55;  
// ou  
*ptr=55;
```



◆ Módulos

▶ Introdução

Conforme apresentado na secção *Módulos – Funções e Procedimentos*, na sebenta de Algoritmia, a criação de módulos constitui uma boa prática de programação. Por exemplo, um módulo específico para o cálculo do fatorial de um número pode ser utilizado por outros módulos sem haver necessidade de repetir código. Em algoritmia, estes módulos designam-se funções e procedimentos.

Na linguagem C, só existem funções, no entanto, o conceito de procedimento também pode ser implementado. Recordando sobre o que já foi referido sobre funções:

- Uma função deve ser concebida para realizar uma única tarefa, aquela que tem por função fazer.
- O código de uma função deve ser o mais independente possível do resto do programa.

Em C,

- A assinatura de uma função:
 - tem um nome único, o qual serve para ser invocada;
 - pode possuir zero ou mais parâmetros (aridade);
 - retorna, para a entidade que a invocou, um valor como resultado da sua execução.
- Se uma função não retornar nenhum valor, nesse caso, deve-se colocar antes do nome da função a palavra reservada: **void**.

▶▶ Modo de funcionamento

- Uma função só é executada se for invocada algures no programa.
- Quando uma função é invocada, o fluxo de execução do programa é desviado para o código da função. Realizada a função, o fluxo de execução regressa para o local onde foi feita a invocação.

- Quando uma função com **argumentos** é invocada, dá-se a passagem dos conteúdos destes para as variáveis locais que se encontram no cabeçalho da função. As variáveis presentes no cabeçalho da função chamam-se **parâmetros** da função. É indispensável que haja correspondência, em tipo e número, entre os argumentos e os parâmetros da função.

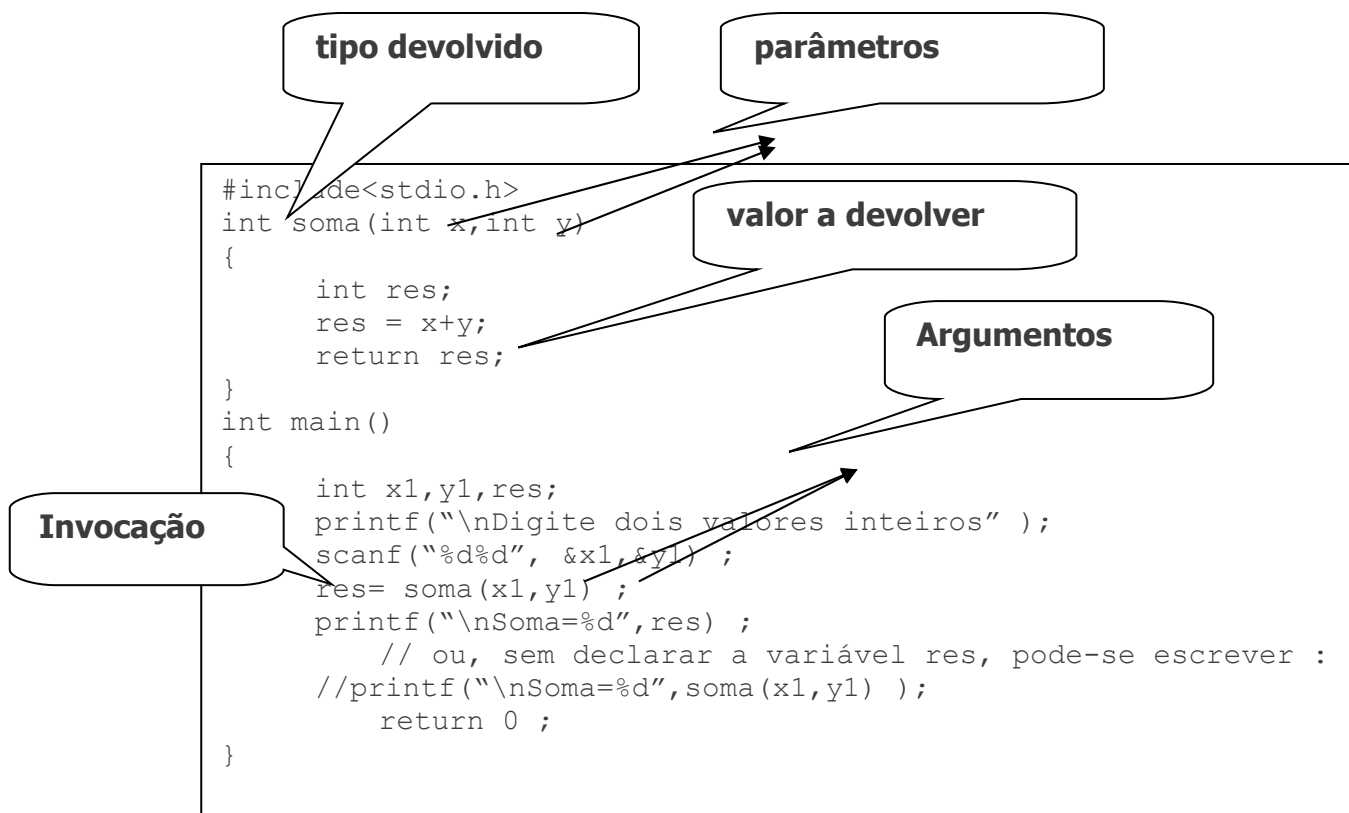
▶▶ Formato genérico de uma função

```
tipo0 nome_função(tipo1 param1, tipo2 param2, ..., tipon paramn)
{
    //corpo_da_função
}
```

tipo₀ : tipo de dados devolvido

tipo₁ a tipo_n: tipos de dados dos parâmetros

Exemplo:



Este programa começa por solicitar dois valores inteiros e a seguir, é invocada a função **soma()** que carrega os valores introduzidos pelo utilizador. Estes valores são depois copiados para os parâmetros da função **soma()**. Nesta função, os valores são somados e o resultado é retornado para a função **main()**, mais propriamente, para a linha onde foi invocada a função. Depois de atribuído o resultado à variável **res**, este é finalmente mostrado ao utilizador.

A função **soma()** também pode ser reescrita da seguinte forma:

```
int soma(int x,int y)
{
    return (x+y);
}
```

A seguir à instrução **return** pode ser colocada qualquer expressão válida em C. É possível em C, uma função estar num argumento de uma outra função, como mostra o exemplo:

```
soma(soma(x1,y1),z1);
```

▶▶ Onde colocar as funções

As funções podem ser escritas em qualquer ponto do programa. No entanto, se o seu código estiver definido depois de serem invocadas, é necessário acrescentar ao programa o **protótipo** da função.

O protótipo de uma função corresponde ao seu cabeçalho seguido de um ponto e vírgula (;). Deste modo, o compilador tem conhecimento que determinada função está implementada algures no programa e assim, quando compilar uma linha que possua uma invocação à função, estando o seu código definido mais abaixo no programa, já não dá erro de compilação. Os protótipos das funções devem ser escritos antes da utilização das funções. Tradicionalmente, eles são declarados a seguir aos **#includes**.

O exemplo anterior pode, por conseguinte, ser reescrito do seguinte modo:

```
#include<stdio.h>

int soma(int x,int y); /* Protótipo */
main()
{
    int x1,y1;
    ...
    printf("\nSoma=%d",soma(x1,y1));
}

int soma(int x,int y)
{
    return (x+y);
}
```

Exercício: Descreva a funcionalidade do seguinte programa:

```
#include<stdio.h>

int maior(int x,int y); /* Protótipo */
int lerInteiro(); /* Protótipo */

main()
{
    int x1,m;
    m=-9999;
    x1= lerInteiro();
    while (x1!=9999)
    {
        m=maior(m,x1);
        x1= lerInteiro();
    }
    printf("\nO maior é=%d",m);
}

int lerInteiro()
{
    int x;
    printf("\nDigite um valor inteiro: ");
    scanf("%d",&x);
    return x;
}

int maior(int x,int y)
{
    if (x>y)
        return x;
    else
        return y;
}

/* O mesmo que: return ((x>y)?x:y); */
```

► Passagem de argumentos por valor e por (cópia de) referência

Em C, existem duas formas de passar informação para dentro de funções:

Passagem por valor: São **copiados os valores** das variáveis, existentes nos argumentos da função que faz a chamada, para as variáveis existentes nos parâmetros da função recetora.

Passagem por cópia de referência: São **copiados os endereços** das variáveis, existentes nos argumentos da função que faz a chamada, para as variáveis existentes nos parâmetros da função recetora.

Para um melhor enquadramento neste assunto tão importante na programação em C, é em seguida estudado o caso clássico: trocar conteúdos de duas variáveis recorrendo a uma função apropriada.

```
#include <stdio.h>

void troca(int a, int b);
main()
{
    int n,m;
    puts("Introduza dois valores ");
    scanf("%d %d",&n,&m);
    printf("\nAntes n=%d e m=%d",n, m);
    troca(n,m);
    printf("\nDepois n=%d e m=%d",n, m);
}
void troca(int a, int b)
{
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
}
```

No programa anterior, seria de esperar que os valores viessem trocados a seguir à invocação da função **troca()**.

Mas isso não acontece!



Para uma melhor compreensão, será mostrada um conjunto de tabelas representando a sequência de operações envolvendo as variáveis: **n**, **m**, **a** e **b**. Os endereços destas variáveis estão em decimal para facilitar a sua leitura. Se forem lidos os valores **2** e **5** para as variáveis **n** e **m**, respetivamente, ter-se-ia a seguinte sequência de passos:

Endereços:	200	400	600	800
Variáveis:	n	m		
Conteúdos:	2	5		

Quando a função começa a ser executada:

Endereços:	1000	1100	1200	1300
Variáveis:	a	b		
Conteúdos:	2	5		

Durante a execução: (Passo: tmp=a)

Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	2	5	2	

Durante a execução: (Passo: a=b)

Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	5	5	2	

Durante a execução: (Passo: b=tmp)

Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	5	2	2	

Os valores só foram trocados dentro da função! Os valores originais continuam sem ser alterados. Porquê?

Todas as vezes que uma função é invocada, é criado um ambiente novo no qual é feita a criação de todas as variáveis necessárias à execução daquela.

Portanto, são criadas cópias dos valores existentes nas variáveis originais e estes passam para o interior da função. Sendo alterados os conteúdos das cópias das variáveis, as originais não são conhecedoras dessas alterações. Quando termina a execução da função, todo o ambiente criado para a função é destruído.

Como resolver? Para a função, devem ser copiados os **Endereços** das variáveis e não os conteúdos das mesmas.

Conforme foi estudado na secção *Noção de Apontador* desta sebenta, o endereço de uma variável é obtido através do operador '&' o qual tem de ser colocado antes do nome de cada argumento que pretendemos passar por referência.

O protótipo da função também tem de ser alterado em consequência deste tipo de mecanismo. Assim, quando um endereço é copiado para um parâmetro de uma função, esse parâmetro tem de ser declarado como um apontador. Só variáveis do tipo apontador podem armazenar endereços de memória. Para isso, basta colocar o símbolo '*' antes do nome de cada parâmetro passado por referência. Este tipo de variável armazena o endereço da variável original. O símbolo '*' no parâmetro da função, indica que a variável é do tipo apontador. Se este símbolo estiver colocado ao lado de uma variável numa instrução qualquer dentro da função, o significado é "**apontado de**", ou seja, contém um endereço que aponta para uma outra variável. Nesta variável, o que interessa é o seu conteúdo (daí, *apontado de*).

Para funcionar bem o programa anterior, existem várias alterações a fazer.

A invocação da função deve ser substituída por:

```
troca(&n, &m);
```

O código da função deve ser substituído por:

```
void troca(int *a, int *b)
{
    int tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}
```

De notar que a instrução **tmp=*a**, por exemplo, significa que a variável local **tmp** passa a armazenar o **apontado por a**. O que se pretende é trocar os apontados das variáveis e não trocar os conteúdos dos apontadores, os quais são endereços.

O novo esquema de memória para a função **troca()**, passa a ter esta representação:

Antes de invocar a função:

(lidos n=2 e m =5)

Endereços:	200	400	600	800
Variáveis:	n	m		
Conteúdos:	2	5		

Quando inicia a execução da função:

Endereços:	1000	1100	1200	1300
Variáveis:	a	b		
Conteúdos:	200	400		

Durante a execução:

(Passo: tmp=*a)

Endereços:	200	400	600	800
Variáveis:	n	m		
Conteúdos:	2	5		
Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	200	400	2	

Durante a execução: (Passo: $*a=*b$)

Endereços:	200	400	600	800
Variáveis:	n	m		
Conteúdos:	5	5		
Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	200	400	2	

Durante a execução: (Passo: $*b=tmp$)

Endereços:	200	400	600	800
Variáveis:	n	m		
Conteúdos:	5	2		
Endereços:	1000	1100	1200	1300
Variáveis:	a	b	tmp	
Conteúdos:	200	400	2	

A variável **tmp** é do tipo **int** uma vez que guarda temporariamente valores inteiros e não endereços.

Em conclusão, quando é necessário realizar alterações aos conteúdos de variáveis dentro de uma função não sendo esta a função aquela onde foram declaradas as variáveis originais, deve-se realizar a passagem de argumentos por cópia de referência para que as alterações aos conteúdos das variáveis originais sejam efetuadas. Se as variáveis originais não precisarem de ser alteradas deve-se realizar a cópia por valor.

Para sistematizar estes conceitos, é em seguida apresentado um programa que faz uma tarefa muito simples: Solicita dois valores inteiros e calcula o quociente e o resto. Estes resultados têm de ser realizados numa função específica tendo esta também de os devolver à função *main()*.


```
#include<stdio.h>
void divide(int d1, int d2, int *q1, int *r1)
{
    *q1= d1/d2; //O apontado de q1 (=quoc) vai ficar com o quociente
    *r1= d1%d2; //O apontado de r1 (=resto) vai ficar com o resto
    return;
}

int main()
{
    int div, divs, quoc, resto;

    // Leitura dos operandos
    printf("\nInsira o dividendo: ");
    scanf("%d",&div);
    printf("Insira o divisor: ");
    scanf("%d",&divs);
    // Os dois primeiros argumentos são passados por valor
    // Os dois últimos argumentos são passados por referência
    divide(div,divs,&quoc,&resto);
    printf("\n O quociente = %d",quoc);
    printf("\n O resto = %d\n",resto);
    system("pause");
    return 0;
}
```

Questão: Qual a razão da função **scanf()** do C, usar o símbolo **'&'** antes do nome de cada variável?

◆ Implementação de *arrays* em C

► Declaração de vetores em C

A declaração de um vetor segue a seguinte sintaxe:

tipo nome_variavel[Nº_de_elementos]

- **tipo**: tipo de dados de cada um dos elementos do vetor.
- **nome_variavel**: nome do vetor.
- **Nº_de_elementos**: valor constante que indica a quantidade de valores do vetor.

Exemplos:

- `int nota[20]` - vetor **nota** com capacidade para 20 inteiros.
- `float venc[100]` - vetor **venc** com capacidade para 100 números reais.
- `char a[5]` - vetor **a** com capacidade para cinco caracteres.

Em C, os índices de um vector com N elementos variam entre 0 e N-1.

Caso prático:

Algoritmo Turma. Este algoritmo lê 10 notas de alunos e imprime as notas superiores à média da turma.

```

10 [Leitura e soma das notas]
    SOMA ← 0.0
    Repetir para i=1 até 10 Passo 1
        Ler(NOTA[i])
        SOMA ← SOMA + NOTA[i]
20 [Calcular a média]
    MEDIA ← SOMA/10
30 [Escrever notas superiores à média]
    Repetir para i=1 até 10 Passo 1
        Se NOTA[i] > MEDIA
            Então Escrever(NOTA[i])
40 [Terminar]
    Exit
    
```

```

#include<stdio.h>

int main()
{
    int nota[10];
    float media,soma;
    soma=0.0;
    /* Leitura das notas */
    for(int i=0;i<10;i++)
    {
        printf("\nNota: ");
        scanf("%d",&nota[i]);
        soma += nota[i];
    }
    /* Cálculo da média */
    media= soma/10;

    /* Escreve as notas > media */
    for(i=0;i<10;i++)
        if (nota[i]>media)
            printf("\n%d",nota[i]);
    return 0;
}
    
```

Algoritmo
Tradução para C
(outras traduções seriam possíveis)

Outro exemplo consiste num programa que lê uma sequência de 20 elementos inteiros e calcula o desvio padrão:

$$\sqrt{\frac{(x_1 - \bar{X})^2 + (x_2 - \bar{X})^2 + \dots + (x_n - \bar{X})^2}{n-1}} = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{X})^2}{n-1}}$$

onde \bar{X} é a média da amostra

```
#include<stdio.h>
#include<math.h>
int main()
{
    int x[20], i;
    float desvio, media, soma, s;
    soma=0.0;

    /* Leitura e soma dos valores */
    for(i=0;i<20;i++)
    {
        scanf("%d",&x[i]);
        soma += x[i];
    }
    /* Calculo da media */
    media= soma/20;

    /* Calcular e escrever o desvio padrão */
    for(soma=0,i=0;i<20;i++)
        soma = soma + (x[i]-media)* (x[i]-media);
    desvio=sqrt(s/19);
    /* Mostra o desvio padrão */
    printf("\nDesvio padrão: %f",desvio);
    return 0;
}
```

► Inicialização automática de vetores

Durante a declaração de um vetor, é possível efetuar a inicialização do mesmo com valores que, deste modo, passa a armazenar.

Normalmente, é utilizada a inicialização automático quando os valores do vetor são muito estáveis. Por exemplo, um vetor com as letras do alfabeto, um vetor com a quantidade de dias de cada mês do ano.

Sintaxe:

tipo var[n]={valor₁,valor₂,...,valor_n}

Exemplo:

```
char vogal[5]={'a','e','i','o','u'}
```

OU

```
char vogal[]={ 'a','e','i','o','u'}
```

O vector **vogal** contém cinco elementos, os quais correspondem às vogais do alfabeto. É possível omitir a capacidade do vector porque os elementos estão por extensão.

Deste modo, evita-se a escrita das seguintes linhas de código:

```
char vogal[5];
vogal[0]='a';
vogal[1]='e';
...
vogal[4]='u';
```

► **Passagem de vetores para funções**

A passagem de um vetor para uma função, implementa-se conforme se segue:

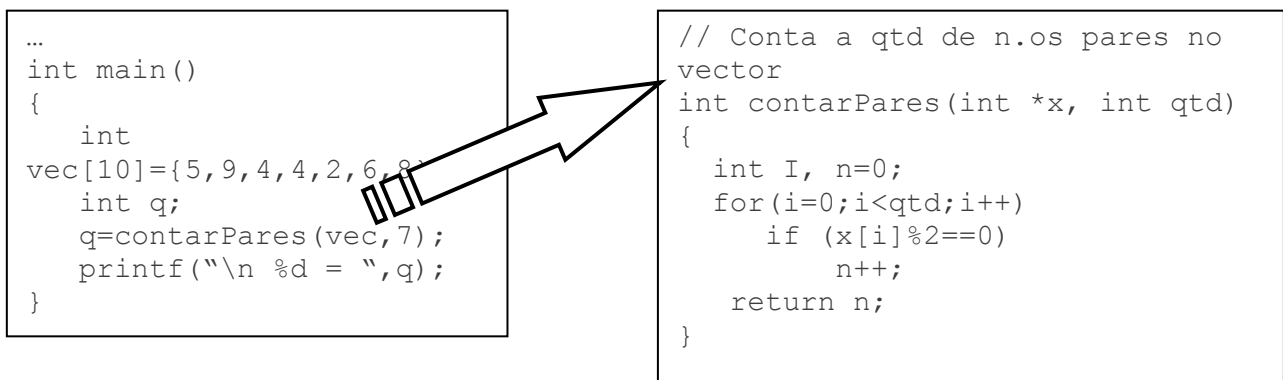
.....	... Funcao_X(tipo nome_vec1[])
tipo nome_vec[#Elementos];	{
.....
Funcao_X(nome_vec);
	}

Quando a função é invocada, carrega, num argumento, o nome do vetor. Por vezes, também é carregado noutro argumento, o número de elementos que possui o vetor. No cabeçalho da função devemos ter uma declaração compatível com este tipo de estruturas, por exemplo: **nome_vec1[]** ou *** nome_vec1**.

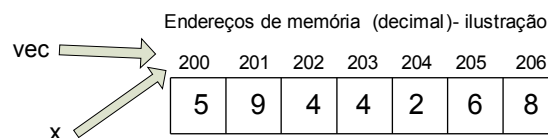
De salientar que os vetores são passados por cópia de referência e, deste modo, o símbolo '&' não deve ser colocado na invocação da função. O símbolo '*' pode ser usado, em vez dos parênteses retos, no parâmetro da função que recebe o vetor.

```
... Funcao_X(tipo *nome_vec1)
{
....
}
```

Exemplo:



O que for alterado em x[] também é alterado em vec[], porque:



Caso prático:

Construir um programa que leia um conjunto de 10 elementos inteiros, calcule a média desses elementos e mostre todos os elementos superiores à média.

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

void lerNotas(int a[])
{
    /* Leitura das notas */
    int i;
    for(i=0;i<TAM;i++)
    {
        printf("\nNota: ");
        scanf("%d",&a[i]);
    }
    return;
}

float calcularMedia(int a[])
{
    /* Calculo da media */
    float media,soma=0.0;
    int i;
    for(i=0;i<TAM;i++)
        soma += a[i];
    media= soma/TAM;
    return media;
}

void imprimir(int a[],float m)
{
    /* Mostra as notas > media */
    int i;
    for(i=0;i<TAM;i++)
        if (a[i]>m)
            printf("\n%d",a[i]);
    return;
}

int main()
{
    float media;
    int nota[TAM];
    lerNotas(nota);
    media=calcularMedia(nota);
    imprimir(nota,media);
    return 0;
}
```

A diretiva “**#define**” informa o pré-processador que deve substituir todas as ocorrências da constante simbólica **TAM** ao longo do programa, pelo seu valor.

É mostrada a seguir uma versão do programa anterior com passagem da quantidade de elementos do vetor para dentro das várias funções do programa.

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

void lerNotas(int a[], int n)
{
    /* Leitura das notas */
    int i;
    for(i=0;i<n;i++)
    {
        printf("\nNota: ");
        scanf("%d",&a[i]);
    }
    return;
}

float calcularMedia(int a[], int n)
{
    /* Calculo da media */
    float media,soma=0.0;
    int i;
    for(i=0;i<n;i++)
        soma += a[i];
    media= soma/n;
    return media;
}

void imprimir(int a[], int n, float m)
{
    /* Mostra as notas > media */
    int i;
    for(i=0;i<n;i++)
        if (a[i]>m)
            printf("\n%d",a[i]);
    return;
}

int main()
{
    float media;
    int nota[TAM], x;
    do {
        printf("Digite o número de elementos :");
        scanf("%d",&x);
    } while(x<1 || x>TAM);
    lerNotas(nota,x);
    media=calcularMedia(nota,x);
    imprimir(nota,x,media);
    return 0;
}
```


Caso prático:

Construir um programa que preencha uma matriz de dimensões 3 linhas por 3 colunas e calcule a soma dos elementos de cada linha.

```
#include <stdio.h>
#define NL 3
#define NC 4

int main()
{
    int mat[NL][NC], i, j, soma;
    printf("\nLeitura dos valores para a matriz");
    for(i=0; i<NL; i++)
        for(j=0; j<NC; j++)
        {
            printf("\n Elemento da linha: %d e coluna: %d --> ", i+1, j+1);
            scanf("%d", &mat[i][j]);
        }

    printf("\nSomar linha a linha");
    for(i=0; i<NL; i++)
    {
        soma=0;
        for(j=0; j<NC; j++)
            soma += mat[i][j];
        printf("\n A soma dos elementos da linha %d = %d", i+1, soma);
    }
    system("pause");
    return 0;
}
```

► Passagem de matrizes para funções

A passagem de vetores de dimensão **N**, para funções, obriga a que pelo menos as N-1 dimensões (mais à direita do vetor) sejam indicadas.

Qualquer uma das seguintes definições está correta em C:

```
f(int v[3][2])
f(int v[][2])
f(int *v[2])
```

```
....
tipo nome_mat[#Elementos1] [#Elementos2];
....
Funcao_X(nome_mat);
```

```
... Funcao_X(tipo nome_mat1[][#Elementos2])
{
....
....
}
```

Quando uma matriz precisa de ser tratada numa função diferente daquela onde foi declarada, o nome da matriz deve ser carregado para um dos argumentos da função. Por vezes, também são carregados, em dois argumentos extra, o número de linhas e o número de colunas da matriz.

Exemplo: `lerDadosMatriz(mat, 3, 4);`

Por seu lado, no cabeçalho da função, devemos ter no parâmetro que “recebe” a matriz, uma declaração do mesmo tipo da estrutura enviada: **nome_mat1[]** **[#Colunas]** .

Exemplo: `void lerDadosMatriz(int mat[][4], int lin, int col);`

Caso prático:

Este programa declara e inicializa a matriz **x**, a seguir altera a matriz de modo que os elementos troquem de posições relativamente à diagonal principal (i.e. o elemento em **[i][j]** troca com o elemento em **[j][i]**).

```
#include <stdio.h>
#define MAX 3

void lerDadosMatriz(int v[][MAX]){
    int i, j;
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++) {
            printf("Elemento da linha %d e coluna %d: ",i+1,j+1);
            scanf("%d",&v[i][j]);
        }
}

void invertePos(int v[MAX][MAX]){
    int i, j,tmp;
    for(i=0;i<MAX;i++)
        for(j=i+1;j<MAX;j++)
        {
            tmp=v[i][j];
            v[i][j]=v[j][i];
            v[j][i]=tmp;
        }
}

void imprime(int v[][MAX]){
    int i, j;
    for(i=0;i<MAX;i++) {
        for(j=0;j<MAX;j++)
            printf("%d ",v[i][j]);
        printf("\n");
    }
}

int main(){
    int x[][MAX]={1,2,3},{4,5,6},{7,8,9};
    int y[MAX][MAX];

    printf("\nLeitura dos dados para a matriz y");
    lerDadosMatriz(y);
    invertePos(x);
    invertePos(y);
    printf("Matriz x");
    imprime(x);
    printf("Matriz y");
    imprime(y);
    system("pause");
    return 0;
}
```

Exercício: Analise o seguinte código.

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

#define MAX 3
#define NUM 10

void ler(int x[][MAX])
{
    printf("\nLeitura da Matriz ");
    int i, j;
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            scanf(" %d",&x[i][j]);
}

void ini_random()
{
    long t;
    time(&t);
    srand((unsigned) t);
}

void gerar(int x[][MAX])
{
    int i, j;
    for(i=0;i<MAX;i++)
        for(j=0;j<MAX;j++)
            x[i][j]= rand()%NUM;
}

void imprime(int x[][MAX])
{
    int i, j;
    printf("\nVisualização da Matriz ");
    for(i=0;i<MAX;i++)
    {
        for(j=0;j<MAX;j++)
            printf("%d ",x[i][j]);
        printf("\n");
    }
}

int main()
{
    int m[MAX][MAX];
    // ler(m);
    ini_random();
    gerar(m);
    imprime(m);
    return 0;
}
```

Algumas notas em relação ao último programa:

- A função **ler()** inicializa uma matriz com valores introduzidos pelo utilizador, enquanto que a função **gerar()** preenche a referida matriz com valores inteiros, entre 0 e **NUM**, os quais são gerados aleatoriamente.
- Para gerar números aleatórios é necessário invocar a função **rand()**, existente na biblioteca **<stdlib.h>**. Esta função devolve um número aleatório entre 0 e **RAND_MAX** (uma constante). Portanto, se pretendermos um número entre 0 e **NUM**, é suficiente calcular o resto da divisão entre o número gerado e **NUM**. O operador **%** dá esse resultado.
- Entretanto, todas as vezes que o programa é executado, a sequência de números gerada é sempre igual. Isto acontece porque a função **rand()** parte sempre de uma mesma 'semente' (valor inicial) para depois gerar os números aleatórios (isto é, pseudo-aleatórios). De modo a que o valor da semente seja diferente, de execução para execução, deve-se inicializar a semente com um valor que esteja sempre a ser alterado. Por exemplo, pode-se aproveitar a hora do relógio do computador para fazer variar a semente.
- A função **ini_random()** é invocada na **main()** com a finalidade de atribuir à variável **t** a hora do relógio, mais precisamente, o número de segundos desde 01/01/1970. A seguir, a função **srand()** tem como responsabilidade inicializar a semente com o valor de **t**. A conversão desta variável do tipo *long int* para *unsigned int* deve ser realizada, uma vez que a função **srand()** precisa de receber um valor do tipo *unsigned int* em vez de *long int*.

◆ Cadeias de caracteres (*strings*), Funções de Entrada/Saída e Funções padrão de tratamento de texto

► Cadeias de caracteres - Representação de *strings*

Na linguagem C, não existe um tipo de dados primitivo para *strings* (ou cadeias de caracteres). Para guardar uma *string* em memória é necessário recorrer a vetores.

Uma *string* não é mais do que um vetor de caracteres. O último elemento do vetor tem de ser um marcador específico que marque o fim da *string*. O marcador universalmente utilizado, é o carácter número 0 (zero) da tabela ASCII¹ o qual se representa por `'\0'`. Este carácter não possui nenhuma representação gráfica. Não confundir `'\0'` cujo código ASCII é 0 (zero) com o carácter `'0'` cujo código ASCII é 48.

Exemplos de *strings*: "Mário André"

"Supermercado Tem Tudo"

"A".

Também, não confundir "A" com 'A'. No primeiro caso, trata-se de uma *string* enquanto que no segundo trata-se só do carácter 'A'.

Uma *string* é um vetor de caracteres mas o inverso pode não ser verdade. Se o vetor de caracteres não terminar com o carácter `'\0'` então não é uma *string* sendo simplesmente um vetor de caracteres.

Representação da *string* "Maria":

<code>'M'</code>	<code>'a'</code>	<code>'r'</code>	<code>'i'</code>	<code>'a'</code>	<code>'\0'</code>
<code>v[0]</code>	<code>v[1]</code>	<code>v[2]</code>	<code>v[3]</code>	<code>v[4]</code>	<code>v[5]</code>

Apesar desta *string* ter cinco caracteres úteis é necessário reservar espaço em memória para, pelo menos, seis caracteres. As posições excedentes ficam reservadas apesar de não serem usadas.

¹ ASCII (American Standard Code for Information Interchange) é um conjunto de códigos para o computador representar números, letras, pontuação e outros caracteres. Em C, trabalhar com caracteres ou com o seu código ASCII é exactamente o mesmo.

A biblioteca **<string.h>** disponibiliza um conjunto vasto de funções para manipulação de *strings*, por exemplo comparar *strings*, concatenar *strings*, determinar o comprimento de uma *string*.

»» Declaração de um vetor de caracteres (*strings*) em C

A declaração de uma *string* obedece à seguinte sintaxe:

```
char nome_s[Tamanho];
```

Tamanho: deve incluir o número de caracteres para guardar a *string* incluindo o carácter `'\0'`.

Exemplo: `char apelido[20]; /* espaço para 19 caracteres úteis */`

»» Inicialização automática de *Strings*

A inicialização automática ou direta de *strings* é possível em C e deve ser realizada no momento da declaração das mesmas, tal como é mostrado nos seguintes exemplos:

```
char bebida[10]="vinho";
```

```
char bebida[10]={'v','i','n','h','o'};
```

```
char bebida[ ]="vinho";
```

```
char *bebida="vinho";
```

Em qualquer dos casos, o compilador coloca automaticamente o terminador `'\0'`. Na primeira e segunda declarações, o compilador atribui o carácter `'\0'` a cada posição que não foi inicializada nos vetores.

► Funções padrão de entrada/saída de dados

As funções de entrada e saída da linguagem C encontram-se disponíveis na biblioteca de rotinas standard **<stdio.h>**. Estas funções podem ser divididas nas seguintes classes:

- E/S Formatada
- E/S Carácter/String
- E/S Bloco

No primeiro grupo englobam-se as funções: **printf()** e **scanf()**.

Ao segundo grupo pertencem um conjunto de funções para leitura/escrita de caracteres e *strings* as quais não incluem formatação específica.

No último grupo estão as funções que permitem o acesso direto a dados guardados em memória permanente, funções essas, utilizadas para manipulação de ficheiros. Este assunto será retomado na secção de ficheiros de texto.

►► Funções de entrada e saída formatada

►►► Entrada formatada

Uma função deste grupo particularmente usada é a função: **scanf()**.

Esta função lê qualquer tipo de dados, não sendo no entanto utilizada para leitura de cadeias de caracteres (*strings*) que envolvam mais do que uma palavra. Para isso, devem ser utilizadas outras funções.

Sintaxe da função **scanf()**:

int scanf(s_formato,arg1,arg2,...)

O primeiro argumento representa uma *string*, a qual contém os formatos respeitantes ao tipo ou tipos de dados a ler. Estes devem corresponder, tanto em tipo como em número, às variáveis existentes nos restantes argumentos da função.

O valor inteiro de retorno da função indica a quantidade de campos lidos.

Como referido anteriormente, os tipos de formatação utilizados em C ("%d", "%f", "%c", "%s", "%u", entre outros) existentes na *string* de formatação devem estabelecer uma relação correta com as variáveis previamente definidas. Nesta *string* só é permitido escrever caracteres indicadores de formatos. Por exemplo, caracteres como '\n' ou '\t' não devem ser incluídos.

Não esquecer, que as variáveis necessitam de ser precedidas do símbolo **&**, exceto para leitura de *strings* (o nome do vetor é um apontador para o primeiro elemento do vetor) com o objetivo dos valores lidos serem de imediato colocados nos endereços das variáveis originais (passagem por cópia de referência).

Um dos problemas da função **scanf()** diz respeito à sua utilização em chamadas consecutivas. O programa seguinte pretende ilustrar esta situação.

```
#include <stdio.h>

int main() {
    char ch1, ch2;
    printf("Introduza um carácter: ");
    scanf("%c", &ch1);
    printf("Introduza outro carácter: ");
    scanf("%c", &ch2);
    printf("Caracteres lidos '%c' e '%c'\n", ch1, ch2);
    return 0;
}
```

O objetivo deste programa é fazer a leitura de dois caracteres e, em seguida, mostrá-los para o ecrã. No entanto, só o primeiro carácter é lido. Porquê?

Este problema tem a seguinte explicação: a grande maioria das funções de leitura de dados executa esta operação encaminhando os dados lidos para o *buffer* do teclado. Depois da introdução do primeiro carácter, é necessário carregar na tecla <Enter>. Assim, em vez de um, é feita a leitura de dois caracteres, o que vai provocar que o primeiro carácter lido seja guardado em **ch1** e o segundo carácter em **ch2**. Deste modo, a segunda instrução de leitura do programa não irá ler o segundo carácter que o utilizador pretendia, porque o **carácter <Enter>**, existente no *buffer*, já foi colocado em **ch2**.

Para resolver este problema, podemos colocar, dentro da *string* de formatação, um espaço em branco imediatamente antes do %c no segundo **scanf()**.

Existe outra forma de resolver o mesmo problema utilizando a função **fflush(stdin)** entre chamadas à função **scanf()**. A função **fflush(stdin)** limpa todos os caracteres que existem no *buffer* do teclado (periférico *standard de input, stdin*).

▶▶▶ Saída formatada

Para mostrar informação, a função mais utilizada em C é a função **printf()**. Esta função constrói e mostra uma *string* formatada para o ecrã.

A sua sintaxe é a seguinte:

int printf(s_formato, arg1,arg2,...)

Ao contrário da função de leitura **scanf()**, a *string* de formatação, existente no primeiro argumento pode conter qualquer carácter.

Nos restantes argumentos da função, pode existir qualquer expressão válida em C, isto é, variáveis, constantes, expressões aritméticas, invocação de outras funções, etc.

Os caracteres de formatação em **s_formato** também devem corresponder em número e tipo (neste último caso, se não houver correspondência não implica à partida problemas) em relação aos restantes argumentos da função **printf()**. Os formatos dos valores são iguais aos utilizados pela função **scanf()**. É possível, também, incluir na *string* de saída, mais alguns símbolos para mostrar os dados de um modo mais profissional.

Por exemplo, a instrução:

```
printf("Ordenado mensal: %.2f\n",ordenado);
```

apresenta o ordenado (tipo **float**) com duas casas decimais (caracteres ".2"). Antes do carácter ponto (".") também pode ser colocado um valor inteiro, permitindo assim reservar um conjunto de espaços, no periférico de saída, para o valor (parte inteira e parte decimal) que vai ser exibido.

Na instrução:

```
printf("Mês %2d\n",mês);
```

os caracteres “%2d” reservam dois dígitos para representar o mês, independentemente do conteúdo da variável mês ter um ou dois algarismos.

O carácter ‘-’ indica que a saída é justificada à esquerda enquanto que o símbolo ‘+’ informa o compilador que os valores numéricos devem ser apresentados com sinal.

»» Funções de entrada e saída não formatada

»»» Leitura e escrita de *strings*

As funções **scanf()** e **printf()** são utilizadas normalmente para a leitura e escrita de *strings* (formatação “%s”). No entanto, a função **scanf()** não pode ler *strings* que contenham: espaços, caracteres do tipo ‘\t’, entre outros. Tudo o que for digitado a seguir a estes caracteres, não é colocado nas variáveis. Para resolver esta situação, deve ser usada a função **gets()**.

- Função **gets()** – (*get string*)

Esta função coloca na variável, existente no argumento, todos os caracteres introduzidos, via teclado, incluindo caracteres do tipo: <espaço>, <tab> e <enter>.

Exemplo:

```
#include<stdio.h>

int main()
{
    char morada[35];
    printf("Introduza a sua morada: ");
    gets(morada);
    printf("\nMorada: %s",morada);
    return 0;
}
```

- Função **puts()** – (*put string*)

Esta função permite escrever uma *string* no ecrã. Esta string é passada para a função através do seu único parâmetro. A mudança de linha é feita automaticamente.

puts("Sou uma string") é equivalente a:

```
printf("Sou uma string\n");
```

- Funções: **getchar()** e **putchar()**

Estas funções são utilizadas para ler e escrever, respectivamente, um só carácter. A primeira função, é útil, por exemplo, quando o programa necessita de ler carácter a carácter e ao mesmo tempo proceder a certas validações, por exemplo, não deixar introduzir alguns caracteres via teclado, por exemplo: 'w', 'y'. Deste modo, quando for detectado um carácter inválido, o programa não o guarda, passando para a próxima instrução.

Exemplos de utilização destas funções:

```
#include <stdio.h>

int main()
{
    char ch;
    printf("Introduza um carácter ");
    ch=getchar();
    printf("Carácter introduzido %c",ch);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    char c='*';
    for(int i=0;i<10;i++)
        putchar(c);
    putchar('\n');
    return 0;
}
```

► Passagem de *strings* para funções

Dado que *strings* são representadas como vetores de caracteres, a passagem de *strings* para funções é realizada da mesma forma que a passagem de vetores de outro tipo qualquer de elementos.

Caso prático 1.

```
#include <stdio.h>
#define TAM 80

int strY(char s1[]) //o mesmo que: int strX(char *s1)
{
    int i=0;
    while(s1[i] !='\0') i++;
    return i;
}
int main() {
    char str1[TAM];
    int x;
    puts("Digite uma string:");
    gets(str1);
    x=strY(str1);
    printf("\n %d",x);
    system("pause");
    return 0;
}
```

Caso prático 2.

```
#include <stdio.h>
#define TAM 80

int strX(char *s1, char *s2) {
    int i=0;
    while(s1[i]==s2[i] && s1[i] !='\0')
        i++;
    return (s1[i]-s2[i]);
}
int main() {
    char str1[TAM], str2[TAM];
    int x;
    puts("Escreva o nome completo:");
    gets(str1);
    puts("Escreva outro nome completo:");
    gets(str2);
    x=strX(str1,str2);
    printf("\n%d",x);
    return 0;
}
```

► Principais funções de manipulação de *strings*

A linguagem C dispõe de um conjunto muito vasto de funções *standard* para manipulação de *strings*. Na tabela abaixo, encontram-se descritas as funções mais utilizadas. Na biblioteca <string.h> pode encontrar muitas outras funções.

Assinatura	Descrição
int strlen(char *s)	Devolve o comprimento de uma string.
char * strcpy(char *dest, char *orig)	Copia uma string para outra.
char * strcat(char *dest, char *orig)	Concatena strings.
int strcmp(char *s1, char *s2)	Compara alfabeticamente strings
int stricmp(char *s1, char *s2)	Compara strings com ignore case
char * strchr(char *s1, char chr)	Procura o carácter chr na string s1
char * strstr(char *s1, char *s2)	Procura a string s2 na string s1

Caso prático 1.

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

#define TAM 80
#define SEP ", "

int main() {
    char nome[TAM], apelido[TAM], completo[2*TAM];

    printf("\nNome:");
    gets(nome);
    if (strlen(nome) != 0)
    {
        printf("\nApelido:");
        gets(apelido);
        strcpy(completo, apelido);
        strcat(completo, SEP);
        strcat(completo, nome);
        puts(completo);
    }
    else printf("\n Nome vazio");
    system("pause");
    return 0;
}
```

Este programa lê duas strings para os vetores nome e apelido, respetivamente, e constrói uma nova string com o seguinte formato: *apelido, nome*.

► Passagem de parâmetros na linha de comando

A função **main()** pode receber informação que é passada na linha de comando, através de dois parâmetros:

```
int main(int argc, char *argv[])
```

argc- É um valor inteiro que indica a quantidade de argumentos que foram passados na linha de comando (inclui o próprio nome do programa).

argv- É um vector contendo todas as *strings* passadas na linha de comando, por isso, é declarado com **[]**, que significa vector, acrescentando-se à declaração **char ***, que significa *string*. Portanto, *argv* é um vector de *strings* (`char *argv[]`). A primeira *string* é sempre o nome do programa.

Por exemplo, se for escrita a seguinte informação, na linha de comando:

```
c:\>concatena BB 123 C2
```

significa:

concatena – nome do programa (1º argumento)

"BB" – 2º argumento

"123" – 3º argumento

"C2" – 4º argumento

possui a seguinte representação:

argv

argv[0]	→	c	o	n	c	a	t	e	n	a	\0
argv[1]	→	B	B	\0							
argv[2]	→	1	2	3	\0						
argv[3]	→	C	2	\0							
argv[4]	→	NULL									

O valor **NULL** é colocado automaticamente indicando que as *strings* do vector já terminaram.

O objectivo de passar informação para um programa, através da linha de comandos, é semelhante aos programas da classe dos utilitários pertencentes aos sistemas operativos. Por exemplo, em MS-DOS, o comando:

```
c:\>rename ficX ficY
```

executa o programa *rename* o qual altera o nome do ficheiro *ficX* para *ficY*. Estes dois últimos nomes constituem os argumentos do programa.

Para finalizar, o programa abaixo, executa a soma de valores passados pela linha de comando.

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

int main(int argc, char *argv[])
{
    int i, soma=0;

    for(i=1;i<argc;i++)
        soma += atoi(argv[i]);
    printf("\n Soma = %d",soma);
    system("pause");
    return;
}
```

Nota: A função **atoi()** (*ascii to int*) recebe uma *string* e devolve o inteiro que nela está representado.

◆ Estruturas

► Introdução

Os tipos de dados primitivos da linguagem C (*char*, *int*, *float* e *double*) possibilitam o processamento simplificado de informação. No caso de ser necessário armazenar informação de entidades como: pessoa, produto, carro, entre outras, seria prático, se a informação de cada uma das entidades ficasse de algum modo agrupada ou relacionada. Por exemplo, a entidade *pessoa* agruparia o nome, número de bilhete de identidade, número de contribuinte e idade.

Na linguagem C, é possível definir estes agrupamentos através de estruturas próprias que passam a definir novos tipos de dados construídos pelo programador.

Uma *estrutura* é um tipo de dados composto por campos (ou membros) de diferentes tipos de dados agrupadas sob o mesmo nome, providenciando uma forma de manter a relação entre a informação.

Pode-se dizer que é um tipo de dados composto cujo formato é definido pelo programador.

► Declaração de estruturas

A declaração de uma estrutura deve obedecer à seguinte sintaxe:

```
struct [nome_da_estrutural]
{
    tipo1 campo1, campo2;
    ....
    tipo2 campo3;
};
```

struct nome_da_estrutura simboliza um novo tipo de dados, a partir do qual podem ser declaradas variáveis.

Os campos das estruturas podem ser de qualquer tipo primitivo (*int*, *float*,..), variáveis do tipo apontador, *arrays* ou mesmo outras estruturas.

Os nomes dos campos dentro duma estrutura devem ser distintos, apesar do nome de um campo poder ser igual ao nome de uma variável qualquer fora da estrutura.

Exemplo 1:

```
struct pessoa
{
    char nome[30];
    char bi[10];
    char contrib[10];
    int idade;
};
```

Exemplo 2:

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

struct pessoa
{
    char nome[30];
    char bi[10];
    char contrib[10];
    struct data data_nasc;
};
```

A declaração de estruturas pode ser feita em qualquer local de um programa. Contudo, geralmente não tem muito interesse, se for realizada dentro de uma função, dado que só é conhecida nessa função. Para outras funções, é como se não existisse.

Normalmente, as estruturas são definidas no início do programa, depois de todas as diretivas.

```
#include<stdio.h>
#define TAM 20
struct ....
// função X
...
// função Y
int main() {
...
}
```

► Declaração de variáveis do tipo de estruturas criadas

A declaração de variáveis do tipo de uma estrutura pode ser feita de duas formas. A primeira, colocando logo a seguir à chaveta de fecho da estrutura, um ou mais nomes de variáveis. Por exemplo,

```
struct pessoa
{
    char nome[30];
    char bi[10];
    char contrib[10];
    struct data data_nasc;
}p1;
```

p1 é uma variável do tipo *pessoa*.

A segunda forma, e a mais aconselhável, é fazer a declaração de variáveis juntamente com o restante código do programa (no interior das funções).

Exemplo:

```
struct pessoa p1, *ptr_p1, vec[20];
```

p1 ➤ é uma variável do tipo *pessoa*.

***ptr_p1** ➤ é um apontador para um tipo de dados *pessoa*.

vec[20] ➤ é um vetor de 20 elementos, sendo cada um deles uma estrutura *pessoa*.

Se as variáveis são declaradas quando se declara a estrutura, não há necessidade de se colocar o nome da estrutura, uma vez que este define o tipo de dados, no entanto, se houver necessidade de declarar outra variável desse tipo (por ex: dentro de uma qualquer função) é obrigatória a colocação do nome da estrutura (uma vez que é este quem identifica o tipo de dados criado).

► Acesso aos membros de dados

Para aceder aos membros ou campos de uma estrutura usa-se o operador ponto (.), conforme se mostra a seguir:

`p1.nome` ➤ campo *nome* da variável *p1* (`struct pessoa p1`).

`p1.bi` ➤ campo *bi* da variável *p1*.

`p1.data_nasc.ano` ➤ campo *ano* da subestrutura *data* pertencendo esta, à variável *p1*.

► Palavra reservada ***typedef***

A palavra ***typedef*** é inúmeras vezes utilizada na programação em C dado que permite renomear tipos de dados consoante as preferências do programador. Por exemplo, é possível utilizar no código C a palavra *inteiro* em substituição da palavra *int*, bastando para isso escrever a seguinte instrução:

```
typedef int inteiro;
```

A partir desse momento, o programador pode escrever instruções tipo:

```
inteiro x1;
```

É possível, contudo, utilizar-se as duas formas em simultâneo (*inteiro* e *int*).

Da mesma maneira, pode-se utilizar a palavra *typedef* antes do nome de uma estrutura (*struct ...*), permitindo simplificar a escrita na etapa de declaração de variáveis (ver exemplo).

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

```
typedef struct pessoa
{
    char nome[30];
    char bi[10];
    char contrib[10];
    TData data_nasc;
} TPessoa;
```

A estrutura *struct data* e a estrutura *struct pessoa* são agora denominadas por *TData* e *TPessoa*, respectivamente, (a formação destes nomes segue as mesmas regras usadas para os nomes das variáveis).

Desta forma, para se declarar uma variável do tipo *pessoa*, é suficiente escrever:

```
TPessoa p1;
```

Não se pode declarar variáveis na definição de um *typedef*.

O programa abaixo apresentado, pretende mostrar e clarificar a matéria apresentada até ao momento sobre estruturas.

```
#include <stdio.h>
typedef struct data {
    int ano;
    char mes[10];
    int dia;
}TData;

typedef struct computador {
    int codigo;
    char proc[10];
    float velocidade;
    int ram;
    TData datacomp;
}TComp;

int main() {
    TComp cpl;
    printf("\n\n\tLer dados do computador\n");
    printf("\nCodigo:");
    scanf("%d%c",&cpl.codigo);
    printf("\nProcessador:");
    gets(cpl.proc);
    printf("\nVelocidade:");
    scanf("%f",&cpl.velocidade);
    printf("\nRAM:");
    scanf("%d",&cpl.ram);
    printf("\nAno:");
    scanf("%d%c",&cpl.datacomp.ano);
    printf("\nMes:");
    gets(cpl.datacomp.mes);
    printf("\nDia:");
    scanf("%d",&cpl.datacomp.dia);
    printf("\n\n\tMostrar dados do computador\n");
    printf("\nCodigo      : %d",cpl.codigo);
    printf("\nProcessador   : %s",cpl.proc);
    printf("\nVelocidade    : %f",cpl.velocidade);
    printf("\nRAM          : %d",cpl.ram);
    printf("\nAno          : %d",cpl.datacomp.ano);
    printf("\nMes          : %s",cpl.datacomp.mes);
    printf("\nDia          : %d\n",cpl.datacomp.dia);
    system("pause");
    return 0;
}
```

► Inicialização automática de estruturas

A inicialização automática de uma estrutura só pode ser feita durante a sua declaração devendo seguir a sintaxe abaixo:

```
struct nome_da_estrutura var = {valor1, valor2, ..., valorn};
```

Exemplos:

```
TComp cp1 = {566, "Intel X", 1.83, 2, 2005, "Outubro", 12};
```

```
TData d1 = {2009, "Janeiro", 12};
```

No caso da variável a inicializar ser um vector, então a inicialização far-se-á colocando cada um dos elementos dentro de chavetas.

Exemplo:

```
TData vec_datas[3] = {{2009, "Janeiro", 12}, {2008, "Abril", 22},  
{2007, "Maio", 1}};
```

► Passagem de estruturas para funções

Conforme tem vindo a ser feito, deve-se construir funções próprias para realizar tarefas, tais como: ler os dados para uma estrutura, mostrar os dados armazenados numa estrutura, entre outras operações.

Uma estrutura é passada para uma função indicando o tipo associado à estrutura.

Tendo como base o programa anterior, serão codificadas e explicadas, as funções para ler e mostrar os dados de um qualquer computador.

Por exemplo, uma função para mostrar os dados de um computador poderia ser implementada da seguinte forma:

```
void mostrar_comp(TComp comp1) {  
    printf("\nCódigo      : %d", comp1.codigo);  
    printf("\nProcessador  : %s", comp1.proc);  
    printf("\nVelocidade   : %f", comp1.velocidade);  
    printf("\nRAM          : %d", comp1.ram);  
    printf("\nAno          : %d", comp1.datacomp.ano);  
    printf("\nMes         : %s", comp1.datacomp.mes);  
    printf("\nDia         : %d\n", comp1.datacomp.dia);  
}
```

Esta função recebe por parâmetro a variável **comp1**, a qual é do tipo **TComp**. Dentro da função são mostrados os membros ou campos de dados pertencentes a esta variável.

O cabeçalho desta função é equivalente a:

```
void mostrar_comp(struct computador comp1)
```

No caso de operações onde seja necessário alterar os dados da estrutura, esta não pode ser passada por valor para a função que a recebe, como no caso de *mostrar_comp()*, mas a passagem tem de ser realizada por cópia de referência, ou seja, é preciso passar para a função o endereço de memória da estrutura.

Neste caso, o código de uma função para ler, ou atualizar, os dados de uma estrutura, poderia ser escrito da seguinte forma:

```
void ler_comp(TComp * comp1)
{
    printf("\nCodigo:");
    scanf("%d%c", &(*comp1).codigo);
    printf("\nProcessador:");
    gets((*comp1).proc);
    printf("\nVelocidade:");
    scanf("%f", &(*comp1).velocidade);
    printf("\nRAM:");
    scanf("%d", &(*comp1).ram);
    printf("\nAno:");
    scanf("%d%c", &(*comp1).datacomp.ano);
    printf("\nMes:");
    gets((*comp1).datacomp.mes);
    printf("\nDia:");
    scanf("%d", &(*comp1).datacomp.dia);
}
```

Algumas considerações importantes sobre este código:

- O parâmetro da função *ler_comp()* deve receber um **apontador** para uma estrutura, para poder alterar os seus campos;
- O operador ponto (.) tem maior precedência do que o operador apontado de (*). Os parêntesis são necessários devido às regras de precedência. Se não fossem colocados os parênteses o compilador iria interpretar **comp1.proc* como **(comp1.proc)* o que daria um erro de compilação, uma vez que *comp1* não é uma estrutura e, por isso, não pode aceder ao membro de dados *proc*.

- Existe outra forma de aceder a um membro de dados. É através do **membro indireto** (->), o qual é colocado entre o apontador e o campo. São equivalentes, as seguintes instruções:

```
scanf ("%d%c", &(*comp1).codigo);  
scanf ("%d%c", &comp1->codigo);
```

da mesma forma, são equivalentes:

```
gets ((*comp1).proc);  
gets (comp1->proc)
```

- Na instrução: *scanf("%d%c",&(*comp1).codigo);* O símbolo **&** é necessário devido ao comportamento particular desta função. Conforme já explicado, *(*comp1).codigo* significa aceder ao membro (.) *codigo* da estrutura *(*comp1)*. É necessário o **&** para se obter o endereço desse membro de dados.

A seguir, é apresentado o código completo de um programa para ler e mostrar os dados de um computador com recurso a funções próprias.


```
#include <stdio.h>

typedef struct data {
    int ano;
    char mes[10];
    int dia;
}TData;

typedef struct computador {
    int codigo;
    char proc[10];
    float velocidade;
    int ram;
    TData datacomp;
}TComp;

void ler_comp(TComp * compl) {
    printf("\nCodigo:");
    scanf("%d%c",&(*compl).codigo);
    //<=>scanf("%d%c",&compl->codigo);
    printf("\nProcessador:");
    gets((*compl).proc);
    // <=> gets(compl->proc)
    printf("\nVelocidade:");
    scanf("%f",&(*compl).velocidade);
    printf("\nRAM:");
    scanf("%d",&(*compl).ram);
    printf("\nAno:");
    scanf("%d%c",&(*compl).datacomp.ano);
    printf("\nMes:");      gets((*compl).datacomp.mes);
    printf("\nDia:");      scanf("%d",&(*compl).datacomp.dia);
}

void mostrar_comp(TComp compl) {
    printf("\nCodigo      : %d",compl.codigo);
    printf("\nProcessador   : %s",compl.proc);
    printf("\nVelocidade     : %f",compl.velocidade);
    printf("\nRAM           : %d",compl.ram);
    printf("\nAno           : %d",compl.datacomp.ano);
    printf("\nMes           : %s",compl.datacomp.mes);
    printf("\nDia           : %d\n",compl.datacomp.dia);
}

int main() {
    TComp cp1;
    struct computador cp2;
    ler_comp(&cp1);
    mostrar_comp(cp1);
    system("pause");
    return 0;
}
```

Uma outra função bastante útil é a pesquisa de valores numa dada estrutura e a devolução de um ou mais campos. Neste exemplo, a função **pesquisa()** devolve a

quantidade de memória, a velocidade de processamento e um valor 1 (um) ou 0 (zero) significando que o computador cujo código é **cod** foi ou não foi encontrado, respetivamente.

Exemplo:

```
int pesquisa(comp_tipo compl, int cod, int *r, float *v) {
    if (compl.codigo==cod)
    {
        *r=compl.ram; *v=compl.velocidade; return 1;
    }
    else return 0;
}
```

► Vetores de estruturas

As estruturas, como qualquer outro tipo de dados, podem ser agrupadas em vetores. Em cada posição do vetor são armazenados todos os campos referentes a uma entidade (*struct*).

A declaração de um *array* de estruturas do tipo TComp pode ser feita de um dos seguintes modos:

<pre>#define MAX 50 typedef struct data { int ano; char mes[10]; int dia; }TData; struct computador { int codigo; char proc[10]; float velocidade; int ram; TData datacomp; }vec [MAX] ;</pre>	<pre>#define MAX 50 typedef struct data { int ano; char mes[10]; int dia; }TData; typedef struct computador { int codigo; char proc[10]; float velocidade; int ram; TData datacomp; }TComp; TComp vec [MAX] ;</pre>
---	---

vec

0	codigo	proc	velocidade	ram	datacomp		
					ano	mes	dia
1							
...							
4							
9							

Cada posição do vetor contém uma estrutura. Para aceder a um elemento do vetor considera-se o nome do vetor e a posição respetiva, mas nesse caso acede-se a toda a estrutura.

Exemplos:

`vec[i]` ➤ Acede à estrutura que ocupa a posição **i** no vetor.

`vec[i].ram` ➤ Acede ao campo *ram* da estrutura localizada na posição **i** do vetor *vec*.

`vec[i].datacomp.ano` ➤ Acede ao campo *ano* da subestrutura *datacomp* localizada na posição **i** do vetor *vec*.

`vec[i].datacomp.mes[2]` ➤ Acede ao terceiro carácter do campo *mes* da subestrutura *datacomp* localizada na posição **i** do vetor *vec*.

Na programação é útil a passagem destes vetores para dentro das funções para operações de leitura, pesquisa, exibição entre outras.

Alguns exemplos:

`ler_comps(vec);` ➤ Lê estruturas para o vetor *vec*.

`mostrar_comps(vec);` ➤ Mostra as estruturas do vetor *vec*.

`n=pesquisar_comp(vec,c);` ➤ Pesquisa, no vetor *vec*, uma estrutura que tenha como código o valor **c** e devolve a posição no vetor dessa estrutura.

O programa que se segue pretende mostrar algumas das operações mais comuns no tratamento de vetores de estruturas.

```
#include <stdio.h>
#define MAX 3

typedef struct data {
    int ano;
    char mes[10];
    int dia;
}TData;

typedef struct computador {
    int codigo;
    char proc[10];
    float velocidade;
    int ram;
    TData datacomp;
}TComp;

// Leitura de MAX estruturas para o vetor
void ler_comps(TComp v[]) {
    int i;
    printf("\n\n\tLeitura dos dados dos computadores");
    for(i=0;i<MAX;i++) {
        printf("\n\tComputador %d",i+1);
        printf("\n\nCodigo:");
        scanf("%d%c",&v[i].codigo);
        printf("\nProcessador:");    gets(v[i].proc);
        printf("\nVelocidade:");
        scanf("%f",&v[i].velocidade);
        printf("\nRAM:");
        scanf("%d",&v[i].ram);
        printf("\nAno:");
        scanf("%d%c",&v[i].datacomp.ano);
        printf("\nMes:");
        gets(v[i].datacomp.mes);
        printf("\nDia:");
        scanf("%d",&v[i].datacomp.dia);
    }
}
```

```
// Mostra MAX estruturas do vetor
void mostrar_comps(TComp v[]) {
    int i;
    printf("\n\n\tDados dos computadores");
    for(i=0;i<MAX;i++)
    {
        printf("\n\tComputador %d",i+1);
        printf("\nCodigo : %d",v[i].codigo);
        printf("\nProcessador : %s",v[i].proc);
        printf("\nVelocidade : %f",v[i].velocidade);
        printf("\nRAM : %d",v[i].ram);
        printf("\nAno : %d",v[i].datacomp.ano);
        printf("\nMes : %s",v[i].datacomp.mes);
        printf("\nDia : %d",v[i].datacomp.dia);
    }
}

// Mostra os campos da estrutura compl
void mostrar_comp(TComp compl) {
    printf("\n\n\tDados do computador");
    printf("\nCodigo : %d",compl.codigo);
    printf("\nProcessador : %s",compl.proc);
    printf("\nVelocidade : %f",compl.velocidade);
    printf("\nRAM : %d",compl.ram);
    printf("\nAno : %d",compl.datacomp.ano);
    printf("\nMes : %s",compl.datacomp.mes);
    printf("\nDia : %d",compl.datacomp.dia);
}

// Pesquisa no vector uma estrutura cujo código seja igual a cod.
// Se existir, devolve a respectiva posição no vector.
// Caso contrário, devolve -1.
int pesquisar_comp(TComp v[], int cod) {
    int i,flag=0;

    for(i=0;i<MAX && flag==0;i++)
        if (v[i].codigo==cod)
            flag=1;

    if (flag)
        return i-1;
    else
        return -1;
}
```

```
// Pesquisa no vector uma estrutura cujo código seja igual a cod.
// Se existir, devolve o endereço da estrutura.
// Caso contrário, devolve NULL.
TComp * pesquisar_comp_outra(TComp v[], int cod) {
    int i, flag=0;

    for(i=0; i<MAX && flag==0; i++)
        if (v[i].codigo==cod)
            flag=1;

    if (flag)
        return &v[i-1];
    else
        return NULL;
}

// Ordenação do vector de estruturas.
void ordenar(TComp v[]) {
    int i, j;
    TComp temp;
    for(i=0; i<MAX-1; i++)
        for(j=i+1; j<MAX; j++)
            if (v[i].codigo>v[j].codigo)
            {
                temp=v[i];
                v[i]=v[j];
                v[j]=temp;
            }
}

// Alguns testes. Função principal com chamada às outras funções do
// programa.
int main() {
    int n, c;
    TComp vec[MAX], *p;

    ler_comps(vec);
    mostrar_comps(vec);
    printf("\nQual o código a pesquisar ?");
    scanf("%d", &c);
    n=pesquisar_comp(vec, c);
    if (n!=-1)
        mostrar_comp(vec[n]);
    else
        printf("\n\tNao existe nenhum computador com o código %d ", c);
    p=pesquisar_comp_outra(vec, c);
    if (p!=NULL)
        mostrar_comp(*p);
    else
        printf("\n\tNao existe nenhum computador com o código %d ", c);
    ordenar(vec);
    printf("\n\tListagem ordenada dos computadores");
    mostrar_comps(vec);
    system("pause");
    return 0;
}
```

► Estruturas especiais (union)

A palavra reservada **union** permite que um conjunto de várias variáveis partilhem o mesmo espaço em memória. Trata-se de uma estrutura especial cuja sintaxe é semelhante à declaração das estruturas em C.

A diferença entre uma **struct** e uma **union** é a forma de armazenamento. No primeiro caso, cada um dos seus campos ocupa um espaço de memória próprio. No segundo caso, existe partilha de memória sendo o espaço ocupado por uma *union* igual ao espaço do maior elemento da mesma.

O programa apresentado a seguir, compara dois tipos de estruturas, *struct* e *union*, em termos de espaço ocupado em memória, e também pretende mostrar as consequências de alterar um dos campos da *union*. Teste o seu código e retire conclusões.

```
#include<stdio.h>

union un_u {
    char ch;
    int num;
    double db;
};

struct es_s {
    char ch;
    int num;
    double db;
};

int main() {
    union un_u u1, u2;
    printf("Tamanho, em bytes, da estrutura: %d\n",
           sizeof(struct es_s));
    printf("Tamanho, em bytes, da union: %d\n", sizeof(union un_u));

    u1.ch = 'A';
    printf("\nValores iniciais\n");
    printf(" %c %d %lf\n", u1.ch,u1.num,u1.db);
    printf("Alterado o campo real");
    u1.db=125.6;
    printf(" %c %d %lf\n", u1.ch,u1.num,u1.db);
    system("pause");
    return 0;
}
```

A estrutura *union* é utilizada em situações muito particulares, normalmente quando é preciso garantir que um mesmo número de Bytes é enviado para o interior de uma função independentemente da sua origem.

Apontadores

Os apontadores são muito usados em C, em parte porque muitas tarefas só podem ser realizadas com este tipo de elemento e ainda porque com a sua utilização o código torna-se mais simples, eficiente e compacto do que o obtido de outras formas.

A informação é armazenada em memória RAM, sendo esta constituída por um conjunto sequencial de localizações referenciadas pelos seus respetivos endereços. A memória RAM pode ser vista como uma sequência de Bytes consecutivos.

100	101	102	...	2000	2001	...
-----	-----	-----	-----	------	------	-----

É possível estabelecer uma analogia com as casas de uma rua, onde moram famílias.

- RUA ➡ Memória RAM.
- CASAS ➡ Endereços, têm tamanhos maiores ou menores dependendo da necessidades das famílias que nela habitam.
- FAMÍLIAS ➡ Informação, dependendo do seu tamanho (no caso das famílias nº de elementos), ocupam casas maiores ou mais pequenas, mais ou menos espaço de memória.

De sublinhar que independentemente do tamanho da casa ("quantidade de espaço de memória alocado") o endereço é único.

Definição

Não existe uma definição única para apontador, por isso, tomaremos como base as duas seguintes que são equivalentes.

Apontador é uma variável que permite guardar o endereço onde se encontra armazenada a informação que se pretende manipular.

Um apontador é uma variável que aponta para outra variável de um determinado tipo.

Declaração

Sabemos que um apontador é uma variável capaz de guardar um número, neste caso, um endereço de outra variável, por isso terá que estar em memória e sendo assim também ocupa espaço. Deste modo, o passo da declaração de um apontador também necessita de ser feito.

A declaração de apontadores obriga à colocação de um asterisco entre o tipo de dados para onde aponta o apontador e o nome da variável que lhe foi atribuída.

Sintaxe: tipo * ptr

onde

ptr - nome da variável do tipo apontador,

tipo – tipo de dados para o qual a variável ptr apontará,

* - indica que se trata de uma variável do tipo apontador.

Exemplos:

int * ptr; // declaração de um apontador para uma variável do tipo inteiro

char * p; // declaração de um apontador para uma variável do tipo char

Inicialização

A inicialização de apontadores deve ser feita através do operador de Endereço **&** ou através da constante simbólica **NULL**. No primeiro caso, estamos a referir-nos ao endereço de uma determinada variável enquanto que no segundo o apontador fica com o endereço de memória número ZERO.

Os dois operadores fundamentais em apontadores são:

& ➤ Fornece o endereço de memória do seu operando, utiliza-se com todos os tipos de dados excepto arrays (neste caso a definição do array indica a posição do seu 1º elemento)

***** ➤ Fornece o conteúdo da posição de memória referida pelo seu operando. Utiliza-se para efetuar a declaração de apontadores.

Na declaração este símbolo indica o "**tipo apontado**"; nas outras instruções indica "**a variável apontada por**"

Exemplos:

int a = 10;

int *ptr = NULL;

int *p = &a;

Estes operadores têm maior precedência que os operadores aritméticos básicos e a associatividade é realizada da direita para a esquerda.

Considerações sobre a manipulação de apontadores:

- Um apontador é um endereço.
- Existe um tipo de dados associado ao endereço, por isso o apontador deve apontar sempre para um objecto desse tipo.
- O endereço de uma variável é sempre o menor dos endereços que esta ocupa em memória.
- Um apontador vazio deve conter o valor da constante NULL , o que significa não apontar para nada.
- Um apontador tem três atributos distintos:

localização, onde ele está guardado \Rightarrow `&a`

conteúdo, o que guarda \Rightarrow `a`

valor indireto, o que é guardado no endereço apontado por `p` \Rightarrow `*p`

• Um apontador é um tipo de dados específico, sendo apenas possíveis as seguintes **operações**:

- **Armazenar** o endereço de um objeto do tipo para onde aponta.
- **Obter** ou **alterar** o conteúdo desse endereço.
- **Adicionar** ou **subtrair** um inteiro.
- **Subtrair** ou **adicionar** outro apontador para o mesmo tipo de dados ou efectuar comparações.
- **Passar** o apontador como argumento para uma função que espere um apontador para o mesmo tipo.

Exemplos:

```
int *a, b, q;
```

`a` \Rightarrow `a` é uma variável do tipo apontador para inteiro, ou seja, `a` é uma variável que contém o endereço de uma variável do tipo inteiro, ou simplificando, `a` é o endereço de um inteiro.

`a = &b` \Rightarrow `a` toma o valor do endereço da variável `b`, ou seja, `a` é o endereço e `*a` é o conteúdo, ou seja `*a = b`.

`q = *a` \Rightarrow `q` toma o valor da variável que está no endereço de `a`, ou seja `q = b`.

`int a, b, *q = &a;` \Rightarrow inicializa `q` com o endereço de memória da variável `a`, ou seja, diz--se que `q` aponta para `a`.

`a = 2;` \Rightarrow atribui o valor 2 a `a`.

`b = *q;` \Rightarrow atribui a `b` o conteúdo da posição para onde `q` aponta, ou seja, o valor 2.

Importante: Um apontador pode conter o endereço de qualquer variável, incluindo de outro apontador.

```
int a, *ap, **app;
```

`a` \Rightarrow declara-se que `a` é um inteiro, `ap` é um apontador para inteiros e que `app` é um apontador para apontadores para inteiros.

`ap = &a;`

`app = ≈`

☞ resulta **app** a apontar para **ap** e esta a apontar para **a**

Se existir em **a** o valor 10 então ****app** representa 10.

Concluindo:

`*ap ⇔ a`

`*app ⇔ ap`

`**app ⇔ *ap ⇔ a`

Exercícios:

I - Suponha que na memória do computador existe uma variável inteira **i** e dois apontadores para int, **p** e **q**. Justifique porque é que as expressões abaixo são legais e outras ilegais.

`p=i` Ilegal

`p=q` Legal

`*p=q` Ilegal

`*p=*q` Legal

`p=*&q` Legal

`p=*q` Ilegal

`*p=&i` ilegal

II – Analise o código do seguinte programa e indique o resultado de cada instrução printf().

```
#include "stdio.h"

int main()
{
    int index, *pt1, *pt2;

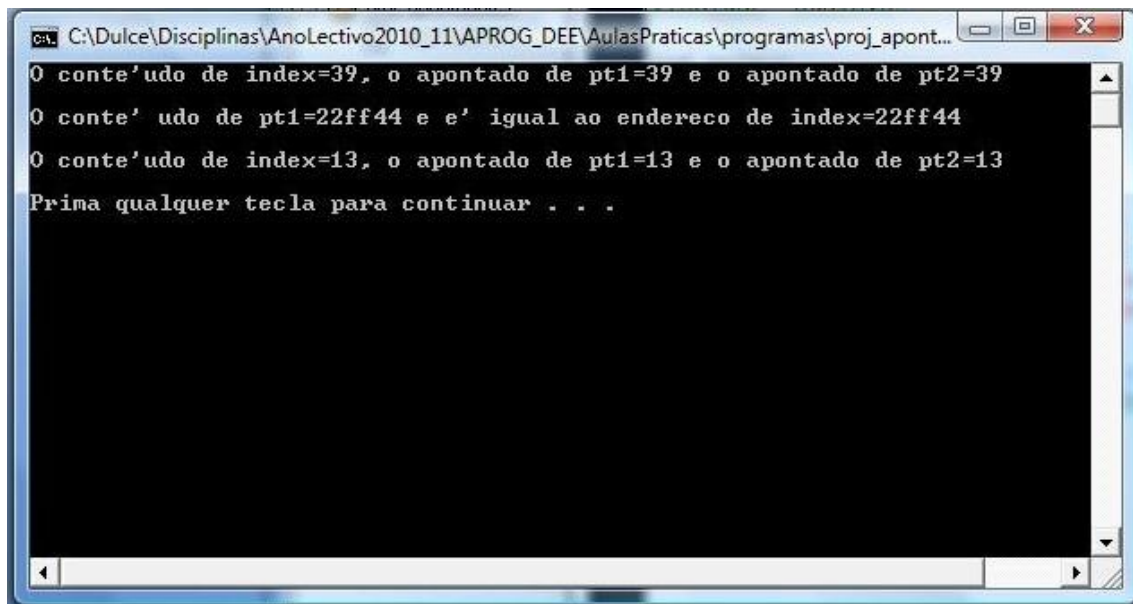
    index = 39;          /* É copiado o valor 39 */
    pt1 = &index;        /* É copiado o endereço de index */
    pt2 = pt1; // É copiado o conteúdo de pt1, ou seja, um endereço

    printf("O conteúdo de index=%d, o apontado de pt1=%d e o apontado de  
pt2=%d\n\n", index, *pt1, *pt2);
    printf("O conteúdo de pt1=%x e igual ao endereço de index=%x\n\n",  
pt1, &index);

    *pt1 = 13;           /* É alterado o conteúdo de index */
    printf("O conteúdo de index=%d, o apontado de pt1=%d e o apontado de  
pt2=%d\n\n", index, *pt1, *pt2);

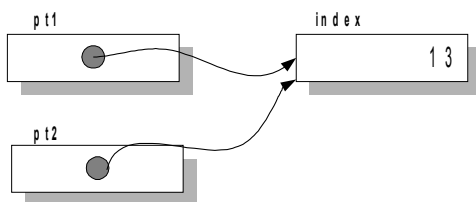
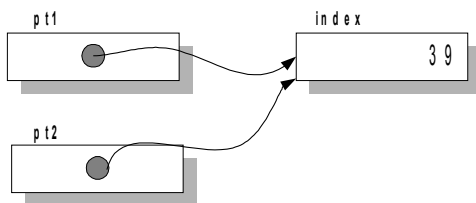
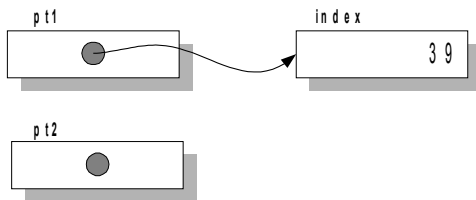
    system("pause");
    return 0;
}
```

Resultado da execução:



```
C:\Dulce\Disciplinas\AnoLectivo2010_11\APROG_DEE\AulasPraticas\programas\proj_apont...
O conteúdo de index=39, o apontado de pt1=39 e o apontado de pt2=39
O conteúdo de pt1=22ff44 e é igual ao endereço de index=22ff44
O conteúdo de index=13, o apontado de pt1=13 e o apontado de pt2=13
Prima qualquer tecla para continuar . . .
```

Ilustração:



Operador **sizeof**

O espaço de armazenamento de variáveis depende do tipo de dados e da própria arquitetura das máquinas.

Os tipos de dados da linguagem C (**char**, **int**, **float** e **double**) ocupam diferente número de Bytes em memória e por essa razão os apontadores devem saber quantos Bytes de memória terão de considerar para uma determinada variável. Essa informação é-lhe dada quando fazemos a sua declaração.

sizeof é um operador de C que devolve o número de Bytes que uma variável ou um determinado tipo ocupam numa determinada máquina.

x		n		pi	
b		27500		3.1415	
1000	1001	1002	1003	2000	2001 2002 2003
sizeof(char)		sizeof(int)		sizeof(float)	

Memória Dinâmica

Até este capítulo, um programa que precisasse de armazenar, por exemplo 200 valores do tipo inteiro, o espaço em memória teria de ser declarado durante a fase de construção do programa. As estruturas de dados, normalmente usadas para esse fim, eram vetores e matrizes.

Ter de conhecer a priori (antes da compilação do programa) a quantidade de elementos a armazenar na memória RAM nem sempre é possível. Para resolver este problema pode-se atuar de duas formas: declarar vetores/matrizes com tamanhos muito grandes ou então não declarar tamanho algum. A primeira hipótese traz normalmente consequências ao nível de reserva de RAM o que normalmente tem de acontecer por excesso, prejudicando a rentabilização dos recursos do computador. A segunda hipótese torna a programação mais flexível e eficiente porque a memória só é alocada durante a execução do programa.

Para uma melhor gestão da memória, a linguagem C permite alocar em tempo de execução, blocos de memória, à medida que for preciso. A contra partida desta importante flexibilidade está na passagem da responsabilidade desta gestão para o utilizador. Este fica responsável por efetuar a correta inicialização do espaço alocado, o correto manuseamento da informação armazenada, como também, por libertar a memória que pediu ao sistema.

As funções a estudar para a gestão dinâmica da memória são: **malloc()**, **calloc()**, **realloc()** e **free()**.

- **Função malloc() (*Memory allocation*)**

Esta função reserva um bloco com o número de Bytes indicado como parâmetro e devolve o endereço do (ou um apontador para o) início desse bloco de memória. Se a função não conseguir reservar a memória pedida, por não existir espaço suficiente, devolve o valor NULL (0).

Sintaxe da função:

void *malloc(size_t n_Bytes)

onde:

size_t: é um número inteiro positivo. Corresponde ao número de Bytes que se pretende alocar.

void *: apontador para o bloco de Bytes criado. O tipo é **void** porque este apontador deve, por defeito, apontar para qualquer tipo de dados. A instrução que invoca esta função deve fazer o *casting* (conversão) explícito para o tipo de dados do apontado.

Exemplos:

```
int *ap, n;
char *s;

/* reserva espaço p/ 10 inteiros */
ap=(int*) malloc(sizeof(int)*10);

/* reservar espaço p/ uma string c/100 caracteres */
s = (char*) malloc(sizeof(char)*100)

/* reservar espaço p/ N floats */
scanf("%d",&n)
s = (char*) malloc(sizeof(char)*n)

-----
int *ptr;
/* reserva espaço p/ 1000 inteiros */
ptr = (int*) malloc(sizeof(int)*1000);

/* altera o conteúdo do 3ª elemento p/ 100 */
*(ptr+2) = 100;

/* altera o conteúdo do 1ª elemento p/ 200 */
*ptr =200;
```


- **Função free()**

Esta função é responsável por libertar o espaço de memória dinamicamente alocado.

Sintaxe:

void free(void *ptr)

Por exemplo:

```
int *ptr;
/* reserva espaço p/ 5000 inteiros */
ptr = (int*) malloc(sizeof(int)*1000);
...
/* liberta o espaço anteriormente alocado */
free(ptr)
```

A seguir são apresentados alguns programas que pretendem evidenciar a gestão dinâmica da memória, com recurso à função malloc():

Caso prático 1:

/*Programa: Declara um apontador para um vetor de N inteiros, a seguir lê os valores e armazena-os, finalmente, mostra a informação armazenada. O valor N é dado pelo utilizador */

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

main() {
    int *ptr, i, n;
    printf("\nIntroduza a quantidade de valores inteiros a processar:");
    scanf("%d",&n);
    printf("\nIntroduza valores inteiros ");
    ptr=(int *)malloc(n*sizeof(int));
    if (ptr!=NULL)
        for(i=0;i<n;i++)
            scanf("%d",&ptr[i]);
    else
        printf("\nMemoria insuficiente");
    if (ptr!=NULL) { // Mostra os valores armazenados em memória
        printf("\nValores inteiros ");
        for(i=0;i<MAX;i++)
            printf("\n%d",ptr[i]);
        free(ptr);
    }
}
```

Caso prático 2:

/*Programa: Declara um apontador para um vetor de N inteiros, a seguir lê os valores e armazena-os e, finalmente, mostra a informação armazenada. O valor N é dado pelo utilizador.
Esta versão utiliza funções definidas pelo programador. */

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

int *lerInteiros(int n1) {
    int i, *p;

    printf("\nIntroduza valores inteiros ");
    p=(int *)malloc(n1*sizeof(int));
    if (p) // ou (p!=NULL)
    {
        for(i=0;i<n1;i++)
            scanf("%d",&p[i]);
    }
    else
        printf("\nMemoria insuficiente");
    return p;
/* devolve o endereço onde está armazenado o 1º inteiro lido */
}

void verInteiros(int *p, int n1) {
    int i;
    printf("\nValores inteiros ");
    for(i=0;i<n1;i++)
        printf("\n%d",p[i]);
    return ;
}

main() {
    int *ptr, n;
    printf("\nIntroduza a quantidade de valores inteiros a processar:");
    scanf("%d",&n);
    ptr=lerInteiros(n);
    if (ptr) ou (ptr!=NULL)
    {
        verInteiros(ptr,n);
        free(ptr);
    }
}
```

Caso prático 3: Diga o que faz o seguinte programa

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

main() {
    int *v,*p,*max,*min;

    puts("Introduza uma sequência de 20 numeros:\n");
    v=(int *)malloc(20*sizeof(int));
    if (v==NULL)
        exit(1);
    for (p=v;p<(v+20);p++)
        scanf("%d",p);

    if (*v>*(v+1))
        printf("\n%d", *v);

    for (p=v+1;p<(v+19);p++)
        if ( (*p>*(p-1)) && (*p>*(p+1)) )
            printf("\n%d", *p);

    if (*(v+19)>*(v+18))
        printf("\n%d", *(v+19);

    free(v);
}
```

- **Função calloc()**

Esta função reserva um bloco para **n_itens** elementos com **tam_itens** Bytes cada um. Devolve o endereço do (ou um apontador para o) início de um bloco de memória com **n_itens*tam_itens** Bytes inicializado com zeros. Se a função não conseguir reservar a memória pedida, por não existir espaço suficiente, devolve o valor NULL (0). Esta função é mais utilizada para alocação de memória para vectores de objectos.

Sintaxe:

void * calloc(size_t num, size_t size)

Exemplo:

```
...
#define N 10
....
int *p;
/*pedido de um bloco de memória para 10 inteiros */
p=(int)calloc(N,sizeof(int)); ...
```

- **Função realloc()**

Esta função permite adicionar espaço de memória ao já alocado pela função **malloc()** ou **calloc()**.

No caso de não ser possível arranjar espaço imediatamente a seguir ao espaço que já está alocado, a função **realloc()** tenta alocar novamente todo o espaço noutra zona de memória. Caso tenha sucesso, os valores que já existiam em memória são também copiados para a zona de memória alocada recentemente. Caso não exista espaço suficiente, tal como no caso anterior devolve um apontador para NULL (0).

Sintaxe:

void *realloc(void *ptr, size_t size)

Exemplo:

```
char *s;
s=(char *)malloc(20) /*Espaço para 20 caracteres */
...
s=(char *)realloc(s,120); /*Aloca mais 100 bytes */
...
```

Nesta função, é necessário indicar no primeiro parâmetro, o bloco de memória que se está a redimensionar (variável do tipo apontador).

Exemplo:

```
int *ap;
/* reserva espaço p/ 10 inteiros */
ap = (int*) malloc(sizeof(int)*10);
...
/* redimensiona o espaço para 20 inteiros, os 1ºs 10 foram copiados */
ap = (int*) realloc(ap,sizeof(int)*20);
```

Neste caso, o endereço do bloco encontra-se armazenado no apontador **ap**, o qual irá ser alterado depois da execução da instrução **realloc()**.

Ficheiros

Introdução

Até agora os programas criados trabalhavam apenas com a informação em memória principal, ou seja, os dados armazenados e manipulados pelos programas após o término do programa eram perdidos.

Para que os dados fiquem acessíveis entre várias execuções de programas é necessário armazenar os dados de forma permanente o que se consegue através da utilização de ficheiros.

Definição

Os ficheiros são estruturas de dados próprias para o armazenamento da informação em memória secundária (discos, disquetes, CD, Zip, ..)

Um ficheiro é uma coleção de bytes referenciados por um único nome.

Existem dois tipos distintos de ficheiros: Ficheiros de Texto e Ficheiros Binários.

Um ficheiro Texto é interpretado em C como sequências de caracteres agrupadas em linhas. As linhas são separadas por um único carácter <New Line> ,(\n) ou LF.

Num ficheiro Binário cada carácter é lido ou gravado sem alteração, ou seja, são guardados como estão na memória, dois bytes para um inteiro, quatro para um float, etc. Por isso é o mais adequado para estruturas de dados mais complexas, por exemplo matrizes e estruturas.

Os periféricos de I/O e os ficheiros - *Streams* padrão

A comunicação com o programa é efectuada através dos periféricos de entrada/saída (teclado, monitor, rato, ...). Estes dispositivos são tratados pelo compilador como **streams** ou ficheiros *standard* sendo um **stream** constituído por uma sequência de bytes.

Quando um programa arranca, o sistema operativo abre automaticamente três ficheiros, correspondendo a cada um deles, um apontador:

stdin	☞	standard input - entrada padrão	(teclado)
stdout	☞	standard output - saída padrão	(monitor)
stderr	☞	standard error output - saída erro padrão	(monitor)

O *stream* **stderr** é normalmente utilizado para escrever mensagens de erro que não dizem respeito às saídas normais.

Estes apontadores estão definidos como macros na biblioteca <stdio.h> que contém todas as declarações necessárias para as funções standard de I/O.

Para trabalhar com ficheiros, a biblioteca padrão ficheiro <stdio.h> fornece um vasto conjunto de funções.

Estas funções dividem-se em várias classes, tendo sido algumas delas alvo de estudo prévio.

Por exemplo:

I/O Formatada	☞ printf(), scanf()
I/O Carácter/String	☞ getchar(), gets(), putchar(), puts()

As três classes de funções standard I/O são:

- I/O Formatada
- I/O Carácter/String
- I/O Bloco

A seguir encontra-se uma tabela com as funções de I/O para manipular ficheiros, declaradas em <stdio.h>.

Além das funções de I/O podem ser encontradas também as funções de Acesso e Posicionamento. Estas funções serão alvo de estudo ao longo deste capítulo.

Função	Função
I/O Formatada	
fscanf()	Acesso
fprintf()	fopen()
	freopen()
I/O Carácter/String	fclose()
getc()	fflush()
fgetc()	setbuf()
fgets()	setvbuf()
putc()	
fputc()	Posicionamento
fputs()	fgetpos()
I/O Bloco	fsetpos()
fread()	fseek()
fwrite()	ftell()
	rewind()
Lidar c/ erro	
clearerr()	
feof()	
ferror()	

Em C todas as funções de entrada padrão são “bufferizadas”, ou seja, o Sistema Operativo guarda os caracteres num espaço de memória temporário até ser premida a tecla <enter>.

Existem dois tipos de ficheiros: de texto e binários.

Streams texto e binário

Um *stream* de ficheiro pode ser de texto ou binário. O *stream* de texto consiste num conjunto de linhas, cada uma com zero ou mais caracteres, terminadas pelo carácter <newline> (**\n**) ou (**linefeed**) <LF>.

O *stream* binário é composto por uma sequência de bytes que não são alterados pelo Sistema. A principal vantagem dos *streams* é a independência em relação a periféricos.

Declaração de um ficheiro

Um programa pode ler e escrever em diversos ficheiros e até mesmo ler e escrever no mesmo ficheiro.

Estes ficheiros são identificados através de nomes segundo as regras do sistema operativo e nenhum ficheiro tem qualquer relação particular com o programa.

Por exemplo no DOS o nome do ficheiro não pode ultrapassar os 8 caracteres, seguido de um ponto e uma extensão que não pode ultrapassar os 3 caracteres. Normalmente usa-se a extensão “.txt” para os ficheiros de texto e “.dat” para os ficheiros binários, mas qualquer uma outra é válida.

O File Pointer - FILE

Para que um programa trabalhe com um ficheiro em disco, o programa e o sistema operativo devem partilhar informações sobre o ficheiro. Estas informações estão armazenadas temporariamente numa estrutura especial existente na biblioteca <stdio.h> que tem o nome **FILE**. Assim um ficheiro é especificado em termos de um apontador para um FILE que se obtém através da chamada à função **fopen()**.

Este apontador de ficheiro (*file pointer*) define as características do ficheiro, nomeadamente o nome, *status* e posição actual, ou seja, identifica totalmente o ficheiro em disco e direcciona as funções de I/O .

```
FILE *fptr ;           /* Apontador para ficheiro */
```

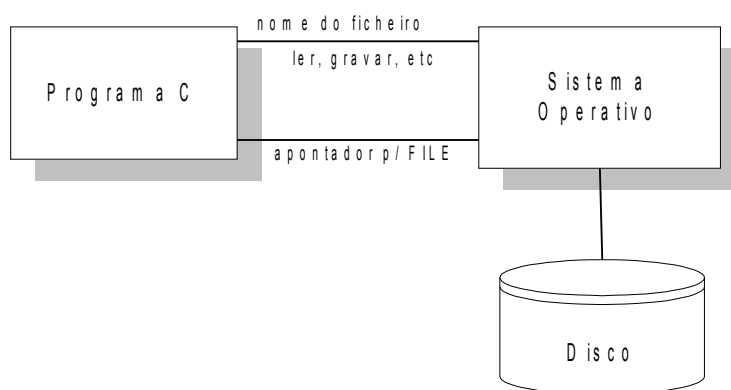
Neste apontador fica armazenada toda a informação sobre o ficheiro.

Funções para manipulação de ficheiros

Antes de se poder ler ou escrever num ficheiro é necessário abri-lo e, após a sua utilização fechá-lo. A abertura de um ficheiro também especifica o modo como ele será utilizado (leitura, escrita ou ambas).

Abertura de um Ficheiro

Como é aberto um ficheiro em C:



```
FILE *fopen(char *nome_fich, char *modo_abertura)
```

Esta função abre um ficheiro cujo nome é dado por **nome_fich** e retorna um apontador para o tipo **FILE** que aponta para a localização na memória da estrutura **FILE**.

```
FILE *fptr ;          /* Apontador para ficheiro */
```

```
fptr = fopen("c:\\path\\ficheiro.ext", "w")
```

O 1º parâmetro da função indica o *path* (caminho) e o ficheiro a abrir.
O 2º parâmetro indica o tipo de abertura que se pretende para o ficheiro.
É em **fptr** que se vai guardar a informação sobre o ficheiro.

A partir do momento da abertura todas as referências ao ficheiro são efectuadas usando o apontador para **FILE** definido na abertura.

Precauções na abertura de ficheiros

É importante que todos os programas que acedem a ficheiros verifiquem se o ficheiro foi aberto com sucesso, antes de ler ou escrever nele.

Se houver insucesso na abertura do ficheiro a função **fopen()** devolve o valor **NULL**, o que pode acontecer nas seguintes situações:

- utilização de um nome de ficheiro inválido,
- tentativa de abertura de um ficheiro que não tenha sido fechado,
- tentativa de abertura de um ficheiro num local inexistente.
- Tentativa de abertura de um ficheiro inexistente em modo de leitura (r).

Para efectuar a verificação do resultado da função de abertura `fopen()` é necessário executar a seguinte instrução:

```
if ((fptr = fopen("c:\path\ficheiro.ext", "w")) == NULL)
{
    printf("Insucesso na abertura do fx. !") ;
    exit() ;
}
```

Lista completa de opções de modos de abertura:

- r** Abre um ficheiro texto para leitura. O ficheiro deve existir no disco.
- w** Abre um ficheiro texto para escrita. Se o ficheiro existir no disco será destruído e reinicializado. Se não existir será criado.
- a** Abre um ficheiro texto para escrita. Os dados serão adicionados no fim do ficheiro existente, ou um novo ficheiro será criado.
- r+** Abre um ficheiro texto para actualização (leitura e escrita). O ficheiro deve existir no disco.
- w+** Abre um ficheiro texto para actualização (leitura e escrita). Se o ficheiro existir no disco será destruído e reinicializado. Se não existir será criado.
- a+** Abre um ficheiro texto para actualização (leitura e escrita). Os dados serão adicionados no fim do ficheiro existente, ou um novo ficheiro será criado.

Por definição, os ficheiros são do tipo texto. Se for pretendido um ficheiro binário é necessário acrescentar um modificador ao modo de abertura do ficheiro. Assim à especificação do modo de abertura pode juntar-se o modificador:

b - para ficheiros binários

Assim:

- rb+** Abre um ficheiro binário para leitura e escrita. O ficheiro deve existir no disco e pode ser actualizado.
- wb+** Abre um ficheiro binário para leitura e escrita. Se o ficheiro existir no disco será destruído e reinicializado. Se não existir será criado.
- ab+** Abre um ficheiro binário para actualizações e para adicionar dados no fim do ficheiro existente, ou um novo ficheiro será criado.

Fecho de um Ficheiro

Após a realização do processamento do ficheiro, ou seja, de se terem realizado todas as funções de manipulação pretendidas sobre ele, é necessário fechá-lo. Para isso, deve ser efectuado o comando:

fclose(FILE *fptr) ➤ Fecha o ficheiro associado ao apontador FILE usado como argumento.

```
FILE *fptr;
...
fptr=fopen("teste", "w");
...
fclose(fptr);
```

fclose () ➤ Esta função devolve **0** em caso de sucesso e **-1** (EOF) em caso de erro. O apontador para a estrutura FILE é desalocado e o ficheiro é fechado.

exit() ➤ Fecha todos os ficheiros abertos e devolve o controlo ao sistema operativo.

Exemplo

Este programa verifica se um determinado ficheiro já existe.

```
#include <stdio.h>
main()
{
    FILE *fptr;

    char nome[20];
    printf("Qual o nome do ficheiro?: ");
    scanf("%s", nome);

    if((fptr=fopen(nome,"r")) != NULL)
    {
        printf("o ficheiro %s já existe ",nome);
        fclose(fptr);
    }
    else
        printf("o ficheiro %s não existe ",nome);
}
```

Funções de controlo

fEOF(FILE *fptr)	Testa Fim Ficheiro. Se fim de ficheiro a função devolve 1 (Verd.) senão devolve 0 (Falso) e -1 se (Erro).
rewind(FILE *fptr)	Permite reposicionar um ficheiro no início, sem reabertura, ou seja, coloca o apontador no início do ficheiro.
ferror(FILE *fptr)	Testa a ocorrência de um erro durante a execução da última operação. A função devolve 1 (Verd.) senão devolve 0 (Falso), se houver erro devolve um qualquer valor.

Ficheiros binários

Para armazenar estruturas de dados complexas (ex: matrizes e estruturas) utilizam-se os ficheiros binários onde os blocos de dados de diferentes tamanhos podem ser armazenados. Desta forma acede-se à totalidade do bloco, em vez de se tratar elemento a elemento, como no caso dos ficheiros de texto.

Escrita de elementos no ficheiro

fwrite(buffer, size, count, FILE *fptr)

☞ escreve (**count**) objectos de tamanho (**size**) da posição de memória apontada por (**buffer**) para o ficheiro especificado pelo apontador **fptr**.

size ☞ depende do tipo de dados a manipular

count ☞ quantidade de objectos

buffer ☞ apontador para uma zona de memória onde se encontram armazenados os dados que se pretende guardar no ficheiro

Exemplo

Considere a estrutura livro

```
.....
typedef struct{
    char titulo[40];
    char autor[40];
    char editora[40];
    long preco;
}LIVRO;

.....
LIVRO livro;
FILE *fptr;
fptr= fopen("agenda.dat","wb");
//instruções para a leitura da estrutura via teclado
```

```
if(fptr == NULL)
{
    printf("erro na abertura do ficheiro %s ",nome);
    exit(0); }
else
    fwrite(&livro,sizeof(livro),1,fptr);
fclose(fptr);
```

- 1º arg → apontador para a localização de memória do dado a ser gravado
- 2º arg → inteiro que indica o tamanho do tipo de dado a ser gravado
- 3º arg → inteiro que indica o nº de itens a serem gravados
- 4º arg → apontador para a estrutura FILE.

A função **fwrite()** não está restrita à gravação de estruturas também pode ser usada para outros tipos de dados, por exemplo nºs reais e vectores:

Exemplo 1

Escrever um nº real (float) num ficheiro binário.

```
#include<stdio.h>

main() {
    FILE *fptr;
    float f=12.23;
    if((fptr=fopen("exp.dat", "wb"))==NULL)
        printf("erro na abertura do ficheiro");
    else
        fwrite(&f,sizeof(float),1,fptr);
    fclose(fptr);
}
```

Exemplo 2

Escrever um vector com 10 nºs inteiros num ficheiro binário.

```
#include<stdio.h>

main() {
    int tabela[10] = {1,2,3,4,5,6,7,8,9,10};
    FILE *fptr;
    if((fptr=fopen("tabela.dat", "wb"))==NULL) {
        printf("erro na abertura do ficheiro");
        exit(0);}
    fwrite(tabela,sizeof(tabela),1,fptr);
    fclose(fptr);
}
```

1º Exercício

Elabore um programa que crie um ficheiro de nome “pauta.dat”, para nele guardar a turma, o número, o nome, disciplina e nota de alunos.

```
#include <stdio.h>

struct aluno
{
    int num ;
    char turma [5] ;
    char nome[21] ;
    char disc[5] ;
    int nota ;
};

main()
{
    FILE *fp ;
    struct aluno fx_al ;
    char resp ;
    clrscr() ;
    fp = fopen("pauta.dat","wb") ; // abertura p/ escrita
    if (!fp)
        printf("Erro na abertura do ficheiro pauta") ;
    do
    {
        printf("\nTurma ") ; gets(fx_al.turma) ;
        printf("\nNº do Aluno ") ; scanf("%d%c",&fx_al.num) ;
        printf("\nNome ") ; gets(fx_al.nome) ;
        printf("\nDisciplina ") ; gets(fx_al.disc) ;
        printf("\nNota ") ; scanf("%d%c",&fx_al.nota) ;
        fwrite(&fx_al,sizeof(struct aluno),1,fp) ;
        printf("\nDeseja Continuar (S/N) ? ") ;
        scanf("%c%c",&resp) ;
    } while(resp!='n') ;
    fclose(fp) ;
}
```

Leitura de Elementos de um Ficheiro

int fread(buffer, size, count, FILE *fptr)

☞ lê (**count**) objectos de tamanho (**size**) cada um, a partir do ficheiro especificado pelo apontador **fptr** para a posição de memória indicada por (**buffer**) que é um apontador para uma zona de memória.

Esta função retorna o número de itens que se conseguiram ler com sucesso, normalmente este nº é igual ao 3º argumento. Caso contrário devolve 0 (zero).

size → depende do tipo de dados a manipular,
count → quantidade de objectos,
buffer → apontador para a zona de memória onde se encontram armazenados os dados que se pretende guardar no ficheiro.

Exemplo

Considerando a estrutura livro:

```
.....
FILE *fptr;
fptr= fopen("agenda.dat","rb");
//    instruções para a leitura da estrutura

if(fptr == NULL)
{
    printf("erro na abertura do ficheiro %s ",nome);
    exit(0); }
else
    fread(&livro,sizeof(livro),1,fptr);
fclose(fptr);
```

1º arg → apontador para a localização de memória onde vai ser guardado o dado a ler do ficheiro.

2º arg → inteiro que indica o tamanho do tipo de dados a ler

3º arg → inteiro que indica o nº de itens a ler

4º arg → apontador para a estrutura **FILE**.

Desde que o ficheiro tenha sido aberto como binário as funções **fread()** e **fwrite()** podem ler e escrever qualquer tipo de informação.

Exemplo 1

Leitura de um nº real (float) de um ficheiro binário.

```
#include<stdio.h>
main() { FILE *fptr;
    float f=12.23;
    if((fptr=fopen("exp.dat", "rb"))==NULL)
        printf("erro na abertura do ficheiro");
    else
        fread(&f,sizeof(float),1,fptr);
        printf(" o número lido é %f :\"f);
    fclose(fptr);
}
```

Exemplo 2

Leitura de um vector com 10 n.ºs inteiros num ficheiro binário.

```
#include<stdio.h>

main()
{
    int tabela[10] = {1,2,3,4,5,6,7,8,9,10} ;
    int i;
    FILE *fptr;
    if((fptr=fopen("tabela.dat", "rb"))==NULL)
    {
        printf("erro na abertura do ficheiro");
        exit(0);
    }
    fread(tabela,sizeof(tabela),1,fptr);
    for(i=0;i<10;i++)
        printf("%d: ", tabela[i]);
    fclose(fptr);
}
```

Alguns exemplos de Leitura e Escrita em Ficheiros Binários

Exemplo 1

Escrita e Leitura de um vector com 10 n.ºs inteiros num ficheiro binário.

1ª versão - escrita de um bloco com 10 n.ºs inteiros.

```
#include<stdio.h>
main()
{
    int i, vect[3], tab[]={1,2,3};
    FILE *fptr;
    clrscr();
    if((fptr=fopen("tabela1.dat","wb"))==NULL) {
        printf("erro");
        exit(0);
    }
    fwrite(tab,sizeof(tab),1,fptr);
    fclose(fptr);
    if((fptr=fopen("tabela.dat","rb"))==NULL) {
        printf("erro");
        exit(0);
    }
    fread(vect,sizeof(tab),1,fptr);
    fclose(fptr);
    for(i=0;i<3;i++) printf("%d", vect[i]);
}
```

2ª versão - escrita e leitura de um inteiro de cada vez

```
#include<stdio.h>
main()
{
    int i, vect[3], tab[]={1,2,3};
    FILE *fptr;
    if((fptr=fopen("tabela1.dat","wb"))==NULL) {
        printf("erro");
        exit(0);
    }
    for(i=0;i<3;i++) /* escrita elemento a elemento */
        fwrite(&tab[i],sizeof(int),1,fptr);

    fclose(fptr);
    if((fptr=fopen("tabela.dat","rb"))==NULL) {
        printf("erro");
        exit(0);
    }
    fread(vect,sizeof(int),3,fptr);
    fclose(fptr);
    for(i=0;i<3;i++) printf("%d", vect[i]);
}
```

Exemplo 2

Leitura e escrita de um vector de estruturas num ficheiro. Escreve uma estrutura de cada vez ou um bloco. Lê uma estrutura de cada vez ou um bloco.

```
#include<stdio.h>
typedef struct{
    char nome[20];
    long telef;
} DADOS; /* fx_est.c */

main() {
    DADOS dados[10];
    int i, n;
    FILE *fptr;
    clrscr();
    if((fptr=fopen("agenda.dat","wb"))==NULL) {
        printf("erro");
        exit(0);
    }
    printf("quantos dados vai inserir?");
    scanf("%d%c", &n);
    for(i=0;i<n;i++)
    {
        printf("nome=");
        gets(dados[i].nome);
        printf("telefone: ");
```



```

scanf("%ld%c", &dados[i].telef);
fwrite(&dados[i], sizeof(DADOS),1,fptr);
                                /* escrita de um valor */
}

/* fwrite(dados, sizeof(DADOS),n,fptr); */
                                /* escrita de um bloco */
fclose(fptr);
if((fptr=fopen("agenda.dat","rb"))==NULL) {
    printf("erro");
    exit(0);
}
    fread(dados,sizeof(DADOS),1,fptr);
                                /*leitura do bloco */
    /*fread(dados,sizeof(DADOS),n,fptr); */
                                /* leitura de 3 elementos */
fclose(fptr);
for(i=0;i<n;i++)
{
    /* fread(&dados[i],sizeof(DADOS),1,fptr); */
                                /* leitura de um valor */
    printf("nome = %s\n", dados[i].nome);
    printf("telefone = %ld\n", dados[i].telef);
}
}

```

Exemplo 3

Elabore um programa que crie um ficheiro de nome “pauta.dat”, para nele guardar a turma, o número, o nome, disciplina e nota de alunos. Mostre o seu conteúdo no ecrã.

```

#include <stdio.h>
struct aluno
{
    int num ;
    char turma [5] ;
    char nome[21] ;
    char disc[5] ;
    int nota ;
};

main()
{
    FILE *fp ;
    struct aluno fx_al ;
    char resp ;
    clrscr() ;
    fp = fopen("pauta.dat","wb") ; // abertura p/ escrita
    if (!fp)

```

```

printf("Erro na abertura do ficheiro pauta") ;
do
{ printf("\nTurma ") ; gets(fx_al.turma) ;
  printf("\nNº do Aluno ") ; scanf("%d
  %*c",&fx_al.num) ;
  printf("\nNome ") ; gets(fx_al.nome) ;
  printf("\nDisciplina ") ; gets(fx_al.disc) ;
  printf("\nNota ") ; scanf("%d%*c",&fx_al.nota) ;
  fwrite(&fx_al,sizeof(struct aluno),1,fp) ;
  printf("\nDeseja Continuar (S/N) ? ") ;
  scanf("%c%*c",&resp) ;
} while(resp!='n') ;
fclose(fp) ;

fp = fopen("pauta.dat","rb") ;// abertura p/ leitura
if (!fp)
  printf("Erro na abertura do ficheiro pauta") ;
  clrscr() ;
while(fread(&fx_al,sizeof(struct aluno),1,fp)!=0)
{ printf("\nTurma: %s",fx_al.turma) ;
  printf("\nNº Aluno: %d ",fx_al.num) ;
  printf("\nNome %s",fx_al.nome) ;
  printf("\nDisciplina: %s",fx_al.disc) ;
  printf("\nNota: %d",fx_al.nota) ; }
getch();
fclose(fp) ;
}

```

Acesso Aleatório – funções de posicionamento

O apontador para ficheiros aponta para o byte do próximo acesso. A função **fseek()** permite a movimentação deste apontador.

Quando um ficheiro é aberto em modo escrita “w”, o apontador para o ficheiro é fixado no seu primeiro byte, quando se abre em modo “a”(append), o seu apontador é posicionado no fim do ficheiro.

int fseek(FILE *fptr, long offset, int)

fseek(fptr,offset,0)

- 1º arg ➡ **fptr** é um apontador para a estrutura **FILE** do ficheiro. Após a chamada de **fseek()** o apontador será movimentado para a posição pretendida.
- 2º arg ➡ **Offset** representa o nº de bytes a partir da posição especificada pelo 3º argumento. Este offset é do tipo **long** e é obtido a partir do produto entre o tamanho da estrutura e a quantidade de registos que se pretende aceder.

3º arg ➦ **Modo** determina de onde o **offset** começará a ser medido, e pode ter três valores.

Modo	Medido a partir de
SEEK_SET - 0	Início do ficheiro, offset sempre positivo
SEEK_CUR - 1	Posição actual, offset positivo ou negativo
SEEK_END - 2	Fim do ficheiro, offset sempre negativo

long ftell(FILE *fptr)

ftell(fptr)

➦ Devolve a posição corrente do apontador de um ficheiro binário em relação ao seu início. Deve ser do tipo **long** porque indica o nº de bytes desde o início do ficheiro até à posição atual. Muito útil em pesquisas.

Exemplo 1

Sabendo que o ficheiro “pauta.dat” existe em disco e contém dados, mostre no ecrã o conteúdo de um qualquer registo, cuja ordem no ficheiro é lida do teclado

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

typedef struct
{ int num ;
  char turma [5] ;
  char nome[21] ;
  char disc[5] ;
  int nota ;
}ALUNO;

main()
{
  FILE *fp;
  ALUNO fx_al;
  long desloca, pos, pos_act;
  int n_reg;
  fp = fopen("paut.dat","rb");/* abertura p/ leitura*/
  if (!fp){
    printf("Erro na abertura do ficheiro pauta") ;
    exit(0); }
  printf("qual o registo a que pretende aceder?|N");
  scanf("%d", &n_reg);
  desloca=n_reg*sizeof(ALUNO);
```

```

/* if(fseek(fp,desloca,0)!= 0){ */
if(fseek(fp,desloca,SEEK_SET)!= 0){
    printf("Impossível mover o apontador p/ lá");
    exit(0); }
pos=ftell(fp);
pos_act=ftell(fp) - sizeof(ALUNO);
/* no inicio do registo */
printf("\n posição anterior %ld",pos);
printf("\n posição %ld",pos_act);
printf("\n tamanho %ld", (long)sizeof(ALUNO));
fread(&fx_al,sizeof(ALUNO),1,fp);
printf("\nTurma: %s",fx_al.turma) ;
printf("\nNº Aluno: %d ",fx_al.num) ;
printf("\nNome %s",fx_al.nome) ;
printf("\nDisciplina: %s",fx_al.disc) ;
printf("\nNota: %d",fx_al.nota) ;
getch();
fclose(fp);
}

```

Pesquisa de registos em Ficheiros Binários

Para efetuar uma pesquisa recorre-se à utilização das funções de posicionamento como se pode verificar no exemplo seguinte.

Exemplo

Função para verificar a existência de um registo com um dado nome - devolve o tamanho em bytes desde o início do ficheiro até ao início do respetivo registo.

```

long pesq_existe(ALUNO *ae)
{
    ALUNO ae;
    rewind(fp); /* supondo o ficheiro aberto */
    fread(&ae, sizeof(ALUNO),1,fp);

    while((!feof(fp)) && (strcmp(ae.nome,(*al).nome)!=0))
        fread(&ae, sizeof(ALUNO),1,fp);
    if(feof(fp))
        return(-1);
    else
        return(ftell(fp)-sizeof(ALUNO));
}

```

devolve o tamanho em bytes desde o início do ficheiro até ao início do registo que se está a pesquisar. Na função main() a chamada seria:

```

long pos;
pos=pesq_existe(&fx_al);

```

Outras funções de posicionamento

```
int fgetpos(FILE *fptr, fpos_t *pos)
```

```
fgetpos(fptr, &pos_act)
```

☞ Guarda a posição corrente do apontador **fptr** em **pos**, para utilização posterior na função **fsetpos()**. O tipo **fpos_t** está definido em <stdio.h> e é do tipo long.

```
int fsetpos(FILE *fptr, const fpos_t *pos)
```

```
fsetpos(fptr, &pos_act)
```

☞ Posiciona o ficheiro na posição **pos**, valor obtido pela função **fgetpos()**.

```
remove(fptr)
```

☞ Elimina o ficheiro

```
rename(fptr, fpt)
```

☞ **fptr** passa a apontar para **fpt**, ou seja, o ficheiro em questão passa a ser outro.

Ordenação de registos em Ficheiros Binários

A ordenação em ficheiros binários pode ser realizada de duas formas:

1. diretamente no ficheiro,
2. ou através da utilização de um vetor auxiliar, no qual se guardam o campo de referência (pelo qual se está a ordenar o ficheiro) e a respetiva posição.

Exemplo

Considerando como exemplo o programa que utiliza a estrutura livro, e supondo que se pretende ordenar por código de livro.

- a) ordenação diretamente no ficheiro, utilizando o método **bubble sort**.

```
void ordenar()
{
    LIVRO livro1, livro2;
    FILE *fp;
    long prec;
    char troca;
```

```

fp=fopen(FICH, "rb+");
if (fp==NULL)
{
    printf("erro , o ficheiro no existe");
    exit(0);
}
clrscr();

do {
    troca=0;
    rewind(fp);
    fread(&livro1,sizeof(LIVRO),1,fp);
    while(fread(&livro2,sizeof(LIVRO),1,fp))
        if(livro1.num_reg>livro2.num_reg)
        {
            troca=1;
            fseek(fp,-2*(long)sizeof(LIVRO),SEEK_CUR);
            fwrite(&livro2,sizeof(LIVRO),1,fp);
            fwrite(&livro1,sizeof(LIVRO),1,fp);
            fseek(fp,0,SEEK_CUR);
        }
        else
            livro1=livro2;
    }
    while(troca);
    fclose(fp);
}

```

b) ordenao atravs do mtodo **linear sort**, utilizando um vector auxiliar

```

void ordenar()
{
    LIVRO livro;
    FILE *fp, *fpt;
    struct{
        int num_reg;
        int pos;
    } aux, vaux[20]; /* limitao no max. de elementos */
    int ps,i,j;

    remove(FICH_B); /* remover qualquer backup */
    rename(FICH,FICH_B); /* passa a ser o backup */

    fp=fopen(FICH_B, "rb+");
    if (fp==NULL)
    {
        printf("erro , o ficheiro no existe");
        exit(0);
    }
    clrscr();
    ps=0;
    while(fread(&livro,sizeof(LIVRO),1,fp))

```

```

    {
        if(ps>19) /* fich. demasiado longo, ultrapassa o tam. do
vector */
        {
            printf("ficheiro demasiado longo");
            getch();
            fclose(fp);
            exit(0);
        }
        vaux[ps].num_reg=livro.num_reg;
        vaux[ps].pos=ps;
        ps++;
    }

    for(i=0;i<ps-1;i++)
        for(j=i+1;j<ps;j++)
            if(vaux[i].num_reg>vaux[j].num_reg)
            {
                aux=vaux[i];
                vaux[i]=vaux[j];
                vaux[j]=aux;
            }

    /* no vector encontra-se uma entrada ordenada p/ o ficheiro, o
elemento de ordem i encontra-se na posição do ficheiro dada por
vaux[i].pos*/

    fpt=fopen(FICH, "wb"); /* novo fich. ordenado */
    if (fp==NULL)
    {
        printf("erro na criação do ficheiro");
        exit(0);
    }

    clrscr();
    for(i=0;i<ps;i++)
    {
        fseek(fp,vaux[i].pos*sizeof(LIVRO),SEEK_SET);
        fread(&livro,sizeof(LIVRO),1,fp);
        fwrite(&livro,sizeof(LIVRO),1,fpt);
    }

    fcloseall();
}

```

Remoção de elementos em Ficheiros Binários

Remoção física

A remoção de um registo de um ficheiro em C é mais complicada, isto porque nenhuma das operações básicas de escrita ou leitura permite fazer isto diretamente. Para realizar esta operação é necessário usar um ficheiro auxiliar

para o qual se passam todos os registos do ficheiro inicial exceto aquele que se pretende eliminar. Em seguida copia-se o novo ficheiro para o ficheiro inicial.

Remoção lógica

Outra hipótese de realizar remoção em ficheiros binários consiste em definir um campo ao registo que indica o estado do registo, por exemplo estado Livre/Ocupado. Deste modo para remover um registo basta apenas marcá-lo “Livre”, para inserir um novo registo ocupa-se o primeiro registo livre ou, se não houver, acrescenta-se no fim do ficheiro.

Exemplo

Considerando que se pretende eliminar um registo, dado o seu código do livro.

Eliminação lógica de um registo:

```
void eliminar_log() {
    LIVRO livro;
    FILE *fp;
    int cod, existe=0;
    long pos=0;

    fp=fopen(FICH, "rb+");
    if (fp==NULL) {
        printf("erro , o ficheiro não existe");
        exit(0);
    }
    clrscr();

    printf("insira o código do livro que pretende eliminar ");
    scanf("%d%c", &cod);

    while(fread(&livro, sizeof(LIVRO), 1, fp) && !existe)
    {
        if(livro.num_reg==cod) /* encontrou */
        {
            existe=1;
            fseek(fp, pos, SEEK_SET); /* posiciona-se no registo */
            if(fwrite(&l_vazio, sizeof(LIVRO), 1, fp)!=1)
                /* escreve reg. vazio */
            {
                printf("erro ao remover utente");
                fclose(fp);
                exit(0);
            }
        }
        pos=ftell(fp);
    }
    if(!existe)
        printf("não existe nenhum livro c/ esse código");
    fclose(fp);
}
```


Eliminação física de todos os registos marcados para eliminação:

```
void eliminar_fis() {

    LIVRO liv[20] ;
    FILE *fp;
    int i,pos=0;

    fp=fopen(FICH, "rb"); /* abre p/ leitura */
    if (fp==NULL)
    {
        printf("erro , o ficheiro no existe");
        exit(0);
    }
    clrscr();

    /* le fich. para vector */
    while(fread(&liv[pos],sizeof(LIVRO),1,fp))
        pos++;
    fclose(fp);

    fp=fopen(FICH, "wb"); /* abre p/ escrita */
    if (fp==NULL)
    {
        printf("erro , o ficheiro no existe");
        exit(0);
    }
    clrscr();
    for(i=0;i<pos;i++)
        if(liv[i].num_reg!=-1)
            fwrite(&liv[i],sizeof(LIVRO),1,fp);
    fclose(fp);
}
```

Eliminação física de um registo utilizando dois ficheiros:

```
void eliminar() {
    FILE *fp, *fpant;
    LIVRO livro;
    int i,cod, existe=0;
    char op;
    long pos;

    clrscr();
    if ((fpant=fopen(FICH, "rb")) == NULL) {
        printf("Erro na abertura do ficheiro ");
        exit(0);
    }

    printf("insira o código do livro que pretende eliminar ");
    scanf("%d%c", &cod);
    while(fread(&livro,sizeof(LIVRO),1,fpant)&& !existe)
```

```

        if(livro.num_reg==cod) /* encontrou o registo */
        {
            pos=ftell(fpant)-(long)sizeof(LIVRO);
            existe=1;
            printf("\n");
            printf("Os dados so os seguintes: \n");
            imprimir(livro); /* mostra os dados do registo, so tem instrues
printf() */

        }
        if(existe==0) {
            printf(" o registo no existe\n\n ");
            getchar();
            fclose(fpant);
        }
        else {
            printf("\nConfirma elimina#o? (S/N) " );
            op=toupper(getche());
            if (op!='S') {
                fclose(fpant);
                return;
            }
            if((fp=fopen(FICH_B,"wb")) == NULL) {
                printf("Erro na abertura do ficheiro");
                fclose(fpant);
                getch();
                return;
            }

            rewind(fpant);
            rewind(fp);
            for(i=0; i<pos;i+=sizeof(LIVRO))
                /* copia todos os registos at, ao eliminado */
            {
                fread(&livro, sizeof(LIVRO), 1, fpant);
                fwrite(&livro, sizeof(LIVRO), 1, fp);
            }
            fseek(fpant, (long)sizeof(LIVRO), SEEK_CUR);
                /* posiciona-se a seguir ao eliminado */
            while(fread(&livro, sizeof(LIVRO), 1, fpant))
                /* escrever todos os registos a seguir ao eliminado
                */
                fwrite(&livro, sizeof(LIVRO), 1, fp);

            fclose(fp);
            fclose(fpant);
            remove(FICH);
            rename(FICH_B,FICH);
        }
    }
}

```

Ficheiros de texto

Escrita em ficheiros texto

A escrita num ficheiro é sempre sequencial, cada operação de escrita acrescenta determinado conteúdo no fim do ficheiro; a operação seguinte continua a escrever a partir desse ponto.

Leitura de ficheiros de texto

A leitura é mais pacífica que a escrita em ficheiros, uma vez que não afecta de forma alguma o conteúdo do ficheiro, mas é ligeiramente mais difícil do ponto de vista da programação.

É possível fazer a escrita/leitura de caracteres e *strings* directamente, uma vez que existem funções de escrita/leitura que o permitem fazer directamente. Outros tipos de dados podem ser também armazenados em ficheiros de texto através da utilização das funções de escrita/leitura formatadas.

Funções de Escrita/Leitura de caracteres

- fputc(ch, fptr)** ➡ Grava o carácter **ch** no ficheiro cuja estrutura **FILE** é apontada por **fptr**.
- ch=fgetc(fptr)** ➡ Lê um carácter do ficheiro cuja estrutura **FILE** é apontada por **fptr**.
- ch=getc(fptr)** ➡ Lê um carácter do ficheiro cuja estrutura **FILE** é apontada por **fptr**.

Exemplo

Gravar caracteres num ficheiro, carácter a carácter. Ler de um ficheiro caracteres, carácter a carácter.

```
#include<stdio.h>
main()
{
    FILE *fptr;
    char c;
    fptr=fopen("fichtext.txt", "w");

    while((c=getch()) != '\r')
        fputc(c, fptr);
    fclose(fptr);
}
```

```
#include<stdio.h>
main()
{
    FILE *fptr;
    char c;
    fptr=fopen("fichtext.txt", "r");

    while((c=getc(fptr)) != EOF)
        printf("%c", c);
    fclose(fptr);
}
```

Funções de Escrita/Leitura de strings

fputs(string,fptr) ➤ Grava **string** no ficheiro cuja estrutura **FILE** é apontada por **fptr**. Esta função não coloca automaticamente o carácter <new-line> no fim de cada linha. É necessário por isso acrescentar a instrução **fputs("\n",fptr)**.

fputs(string,stdprn) ➤ Imprimirá **string** na impressora.

fgets(string,n,fptr) ➤ Lê **string**, uma por cada instrução, do ficheiro cuja estrutura **FILE** é apontada por **fptr**. Esta função inclui o carácter de <new-line> e o carácter **NULL** no final da string.

fgets(string,80,stdin) ➤ Lê **string** do teclado (tamanho 80).

Exemplo

Escrita de linhas (de caracteres) num ficheiro, linha a linha. Leitura de linhas (de caracteres) do ficheiro, linha a linha.

<pre>#include<stdio.h> main() { FILE *fptr; char str[10]; fptr=fopen("fichtext.txt", "w"); while(strlen(gets(str))>0) { fputs(str,fptr); fputs("\n",fptr); } fclose(fptr); }</pre>	<pre>#include<stdio.h> main() { FILE *fptr; char str[10]; fptr=fopen("fichtext.txt", "r"); while(fgets(str,20,fptr) != NULL) printf("%s",str); fclose(fptr); }</pre>
--	---

Funções de Escrita/Leitura Formatada

Estas funções permitem a manipulação de qualquer tipo de dados, incluindo *arrays* e estruturas. No entanto, não é possível considerar esta informação como um bloco de informação como acontece nos ficheiros binários. Para escrever/ler os dados é necessário formatar cada tipo de dados, exactamente da mesma forma que na leitura/escrita do teclado/monitor.

fprintf(fprr,"%s %d",tit,num) ➤ permite escrever dados formatados (todas as opções de printf) na estrutura **FILE** apontada por **fprr**.

fscanf(fptr,"%d %s",&numero,&nome) ➡ permite ler dados formatados (todas as opções de printf) da estrutura **FILE** apontada por **fptr**.

Exemplo

Escrita de dados de tipos diferentes (string, int e long) num ficheiro de forma formatada. Leitura de dados de tipos diferentes (string, int e long) de um ficheiro de forma formatada.

<pre>#include<stdio.h> main() { FILE *fptr; int reg_num; char titulo[30]; long prec; fptr=fopen("fichtext.txt", "w"); do { printf("\n Digite nº, titulo e preço: "); scanf("%d %s %ld", &reg_num, titulo, &prec); if(reg_num==0) break; fprintf(fptr, "%d %s %ld \n"), reg_num, titulo, prec); }while(reg_num!=0); fclose(fptr); }</pre>	<pre>#include<stdio.h> main() { FILE *fptr; int reg_num; char titulo[30]; long prec; fptr=fopen("fichtext.txt", "r"); while(fscanf(fptr,"%d %s %ld",&reg_num, titulo, &prec)!= EOF) printf("%d %s %ld",reg_num,titulo, prec); fclose(fptr); }</pre>
---	--

Exercício I de ficheiros de texto

Escreva um programa que imprima um ficheiro de texto no ecrã, 20 linhas de cada vez. O nome do ficheiro deve ser especificado na linha de comando. O programa deve apresentar as 20 linhas seguintes depois do utilizador carregar em qualquer tecla.

```
#include <stdio.h>
#include <string.h>
/* fich_txt1.c */
void listar(char nomefich[]) {
    FILE *ft;
    int i=0;
    char linha[81];          /* fich_t3.c */

    clrscr();
    if ((ft = fopen(nomefich,"rt"))==NULL)
```

```

        {
            printf("erro na abertura\n");
            exit(1);
        }
        while (!feof(ft))
        {
            fgets(linha,80,ft);
            printf("%s",linha);
            i++;
            if(i==20)
            { printf("Pressione qualquer tecla <ENTER>");
              getchar();
              i=0;
            }
        }
        fclose(ft);
        printf("\nOperacao executada");
        getchar();
    }

void main(int argc, char *argv[]) {
    if (argc==2)
        listar(argv[1]);
    else
        printf(" N. de argumentos mal definido\n");
}

```

Exercício II de ficheiros de texto

Modifique o programa anterior para que aceite mais dois argumentos na linha de comando. O primeiro é um número inteiro e indica a primeira linha a ser impressa e o segundo é um número inteiro que indica a última linha a ser impressa.

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void listar(char nomefich[],int li, int lf)
{
    FILE *ft;
    int i=0,k=0;
    char linha[81];
    clrscr();
    if ((ft = fopen(nomefich,"rt"))==NULL)
    {
        printf("erro na abertura\n");
        exit(1);
    }
    while (!feof(ft))
    {
        fgets(linha,80,ft);
        k++;
        if (k>=li && k<=lf)

```

```

        {
            printf("%s", linha);
            i++;
            if(i==20) {
                printf("Pressione qualquer tecla <ENTER>");
                getch();
                clrscr();
                i=0;
            }
        }
    }
    fclose(ft);
    printf("\nOperação executada");
}

int main(int argc, char *argv[]) {
    int n=argc;
    if (n==4)
        listar(argv[1], atoi(argv[2]), atoi(argv[3]));
    else
        printf("numero de argumentos mal definido");
}

```

Ficheiros Binários versus Ficheiros Texto

Objetivos:

Comparação entre ficheiros binários e texto.

Argumentos da linha de comando.

Comparação entre ficheiros Binários e de Texto

Um ficheiro binário está organizado por registos ao contrário dos ficheiros texto que são constituídos por linhas. Os dados são representados sempre num bloco fixo de N caracteres, dentro desse bloco cada campo ocupa sempre a mesma posição e tem sempre o mesmo tamanho.

Pelo facto da organização de um ficheiro binário ser paralela à da memória torna a representação e manipulação em binário mais compacta e eficiente, porque não são necessárias conversões.

Uma vez que as dimensões e representações dos diversos tipos de dados variam de máquina para máquina, os ficheiros binários podem não ser transportáveis entre máquinas diferentes.

Os programas que trabalhem com grandes volumes de dados devem sempre usar ficheiros binários.

Os ficheiros texto contêm texto legível o que permite a manipulação e visualização do seu conteúdo a partir de um editor de texto normal.

Os ficheiros de texto são completamente independentes das particularidades da máquina.

Argumentos da linha de comando

A linguagem C fornece a capacidade de passar argumentos para dentro dos programas. Isto é feito através dos argumentos **argc** (Argument Count) e **argv** (Argument Values) do main.

```
/* arg.c*/
main(argc,argv)
int argc ;
char *argv[] ;
{
    int j ;
    printf("Número de argumentos é %d\n",argc) ;
    for (j=0; j<argc; j++)
        printf("Argumento N° %2d é %s\n",j,argv[j]) ;
}
```

Exemplo de execução do programa é:

c:\tc> arg um dois tres

número de argumentos ↗ 4
primeiro argumento ↗ c:\tc>arg.exe
segundo argumento ↗ um
terceiro argumento ↗ dois
quarto argumento ↗ três

Exercício

Escreva um programa que imprima um ficheiro de texto no ecrã, 20 linhas de cada vez. O nome do ficheiro deve ser especificado na linha de comando. O programa deve apresentar as 20 linhas seguintes depois do utilizador carregar em qualquer tecla.

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

void listar(char nomefich[])
{
    FILE *ft;
    int i=0;
    char linha[81];

    if ((ft = fopen(nomefich,"rt"))==NULL)
    {
        printf("erro na abertura\n");
        exit(1);
    }
}
```



```
while (!feof(ft))
{
    fgets(linha,80,ft);
    printf("%s",linha);
    i++;
    if(i==20)
    { printf("Pressione qualquer tecla <ENTER>");
      getchar();
      i=0;
    }
}
fclose(ft);
printf("\nOperacao executada");
}
void main(int argc, char *argv[])
{
    if (argc==2)
        listar(argv[1]);
    else
        printf(" N. de argumentos mal definido\n");
}
```