CTeSP
Desenvolvimento para a Web e
Dispositivos Móveis

Programação em Linguagem C

Escola Superior de Tecnologia e Gestão de Viseu
Instituto Politécnico de Viseu

Carlos Simões

ÍNDICE

1 Introdução à Linguagem de Programação C

A linguagem de Programação ***C*** foi criada e implementada para o sistema operativo (S.O.) *Unix* por Dennis Ritchie.

O S.O. Unix é escrito em ***C***.

1.1 Estrutura de um programa

Comecemos por uma breve introdução à linguagem ***C***. Pretende-se mostrar os elementos fundamentais num programa nesta linguagem, mas sem nos preocuparmos com os detalhes.

Um programa em C é constituído por uma ou mais "funções". Uma destas funções deve ter o nome *main*. As funções contêm instruções que definem as operações a executar pelo computador. As funções em C são semelhantes às funções e procedimentos da linguagem *Pascal*.

A identificação de uma função pode ser qualquer, à excepção de palavras reservadas da linguagem, mas tem de existir uma função com o nome *main* no início da qual o programa começa a ser executado. A função *main* normalmente chamará outras funções para executar o que é pretendido, algumas feitas pelo programador, outras disponíveis bibliotecas já incluídas no compilador desta linguagem.

Um programa em linguagem ***C*** tem a seguinte estrutura:

```
#include "biblioteca"
.
.
.
#define

tipos

variáveis globais

funções
```

Em *C*, um programa para mostrar no monitor a frase "`Olá Mundo!`" é:

```
#include <stdio.h>
void main()
{
  printf("Olá Mundo!");
}
```

A primeira linha do programa

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

informa o compilador que deve incluir a informação da biblioteca *`stdio.h`* (*`standard input/output`*).

Na segunda linha está declarada a única função deste programa, a função *main*, e nesta existe apenas uma única instrução que chama a função `printf` (disponível na biblioteca *stdio.h* da linguagem ***C***) para escrever a mensagem no monitor.

1.2 Funções

Uma função tem a estrutura

```
tipo identificação( declaração dos parâmetros )
{
       variáveis locais;

       instruções;
}
```

estando as instruções sempre delimitadas por chavetas {}. `tipo` define o tipo do dado que será devolvido pela função. Quando o tipo é omitido é considerado por omissão como sendo inteiro. Quando não se pretende que a função devolva um valor declara-se como sendo sem tipo usando a palavra reservada *void* (como foi usado no pequeno programa da página anterior).

A ordem pela qual as funções aparecem no programa é irrelevante. Para se poder transferir informação entre as funções são usados parâmetros que se incluem entre parêntesis a seguir ao nome da função. Cada parâmetro é separado do seguinte através de uma vírgula e é declarado de forma igual às variáveis locais.

2 Tipos de variáveis e constantes

As variáveis e constantes são os tipos de dados básicos manipulados num programa. A declaração das variáveis permite definir o seu tipo e, eventualmente, o seu valor inicial.

Os nomes dados às variáveis, tal como às constantes e funções, são denominados **identificadores**. Estes nomes são conjuntos de caracteres que começam obrigatoriamente por uma letra ou pelo carácter '_' (*underscore*). Os restantes caracteres podem ser letras, algarismos ou o *underscore*.

O número máximo de caracteres num identificador depende do compilador de *C* que estejamos a usar. No *Turbo C*, por exemplo, apenas são considerados os primeiros 32 caracteres. Assim, os dois identificadores seguintes

```
identificador_muito_longo_como_exemplo1
identificador_muito_longo_como_exemplo2
```

são considerados pelo compilador como

```
identificador_muito_longo_como_e
```

não havendo distinção entre eles. O compilador distingue, no entanto, as letras maiúsculas das minúsculas pelo que `contador`, `Contador` e `CONTADOR` são 3 identificadores distintos. Como identificadores não podem ser usadas as palavras reservadas da linguagem.

Em *C* todas as variáveis têm de ser declaradas antes de serem usadas, uma vez que o compilador tem de saber qual o tipo de dado armazenado na variável antes de a poder usar numa instrução.

Estão definidos cinco tipos básicos de dados na linguagem ***C***:

tipo	palavra reservada em ***C***	tamanho (*bytes*)	Gama de variação (1)
Carácter	`char`	1	`0` a `255`
inteiro	`int`	2	`-32 768` a `32 767`
		4	`-2 147 483 648 a 2 147 483 647`
real	`float`	4	`3.4E-38` a `3.4E+38`
real de dupla precisão	`double`	8	`1.7E-308` a `1.7E+308`
sem tipo	`void`	0	

(1) Na realidasde, a gama de variação dos tipos `int` e `float` dependem da máquina usada.

Adicionalmente, os quatro primeiros tipos podem ser alterados usando os chamados **modificadores de tipo**. Em *C* estão definidos os seguintes modificadores de tipo:

```
signed
unsigned
short
long
```

As variáveis são declaradas usando a sintaxe:

```
tipo nome_variavel;
```

As cadeias de caracteres (*string*'s) e os *array*'s são declarados colocando-se o número de elementos entre parêntesis rectos.

Exemplo de declaração de variáveis:

```
int x;                  /* variável inteira */
short int si;           /* variável inteiro curto */
char frase[80];         /* string ou array de caracteres */
float x,y;              /* variáveis reais */
```

As variáveis podem também ser inicializadas logo na declaração, por exemplo:

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

As constantes em *C* são valores que não podem ser alterados pelo programa, só podendo ser declaradas como sendo de um dos tipos básicos indicados acima.

As constantes inteiras são números sem parte fraccionária, por exemplo, **123** e **-35** são constantes inteiras. Constantes de vírgula flutuante requerem o uso de uma parte fraccionária ou um expoente, por exemplo **11.123** e **1E-2** são constantes de vírgula flutuante. Por omissão, estas constantes são do tipo `double` a menos que se use um sufixo. Os sufixos `f` ou `F` indicam um `float`, os sufixos `l` ou `L` indicam um `long double`.

Além do sistema **decimal**, é possível representar constantes inteiras usando os sistemas **octal** e **hexadecimal**. Uma constante no sistema octal é indicada usando um **`0`** (zero) inicial, por exemplo **`037`** (correspondente a `31` no sistema decimal). Em hexadecimal usa-se **`0x`** no início do número, por exemplo **`0x1f`**.

As constantes do tipo carácter devem ser delimitadas por **plicas**. Por exemplo, **'a'** e **'$'** são ambas constantes do tipo carácter.

Em ***C*** é também possível definir constantes do tipo *string*. Uma *string* é um conjunto de caracteres delimitado por **aspas**, por exemplo "`Isto é um teste`". Não se devem confundir *string*'s com caracteres. Por exemplo, '`a`' é um carácter enquanto "`a`" é uma *string* com apenas um carácter.

O uso de plicas para delimitar constantes do tipo carácter funciona com a grande maioria dos caracteres que podem ser mostrados, no entanto, existem alguns caracteres especiais tal como o ***carriage return*** que não podem ser mostrados. Para eles, a linguagem *C* fornece as chamadas sequências de *escape* que usam o caracter *backslash* '\'. Alguns dos caracteres que podem assim ser representados são:

Código	Significado
\b	backspace
\n	newline
\r	carriage return
\t	horizontal tab
\\	backslash
\0	constante octal
\x	constante hexadecimal

Para declarar constantes usa-se a palavra reservada **`define`** segundo a sintaxe:

```
#define nome_constante valor
```

Exemplos:

```
#define min 0
#define max 200
```

3 Instruções de Entrada/Saída

3.1 Saída de dados

Em *C*, para apresentar mensagens no monitor pode utilizar-se a função `printf` que já usámos no primeiro programa apresentado neste texto. Esta função está definida na biblioteca `stdio.h` e permite diversos argumentos.

Na exemplo apresentado, a instrução

```
printf("Olá Mundo!");
```

chama a função `printf` com o argumento "`Olá Mundo!`" que é a *string* que se pretende mostrar no monitor. De notar que a *string* pode conter sequências de *escape*, por exemplo

```
printf("Olá \nMundo!");
```

mostrará no monitor:

```
Olá
Mundo!
```

A função `printf` permite mais argumentos tendo como sintaxe:

`printf(string_de_controlo [, arg1, arg2, ...])`

(em que os argumentos entre **`[]`** são opcionais). A *`string`*`_de_controlo` pode conter os caracteres a mostrar, sequências de *escape* e **especificadores de conversão**. Estes especificadores de conversão permitem definir como os argumentos vão ser mostrados no monitor, devendo existir um argumento por cada especificador de conversão. Cada especificador de conversão começa pelo carácter '**%**' e termina com um caracter de conversão. Entre estes dois pode existir, por esta ordem:

- as seguintes *flags* (por qualquer ordem):
  - (sinal menos) - especifica um ajuste à esquerda do argumento
  + (sinal mais) - especifica que o número deve ser sempre mostrado com sinal
  espaço - se o primeiro carácter não for o sinal será mostrado um espaço em branco no início do número.
  0 (zero) - para conversões numéricas especifica que o campo deve ser preenchido com zeros antes do número.
- número de largura mínima. O argumento será mostrado num campo com pelo menos esta dimensão, e maior se necessário Se o argumento consistir em menos caracteres que o indicado serão adicionados à esquerda (ou à direita se for especificado um ajuste à esquerda) caracteres adicionais, até que se obtenha um número total de caracteres igual ao especificado. O carácter usado no preenchimento é normalmente o espaço em branco, mas é usado o carácter '**0**' se tiver sido usada a *flag* **0**.
- . (ponto) separa o campo anterior da precisão.
- número de largura máxima (também chamado precisão). Número máximo de caracteres de uma *`string`* que serão mostrados ou o número máximo de dígitos após o ponto decimal se o argumento for um valor em vírgula flutuante em conversões *`e`* ou *`f`* ou o número de algarismos significativos em conversões *`g`* ou o número mínimo de dígitos se o argumento for um valor inteiro.
- *`h`* ou *`l`* - Se o inteiro for para mostrar como *short* (*`h`*) ou *long* (*`l`*).

Quanto ao carácter de conversão, ele pode ser:

Carácter	**mostra argumento como...**
`d`	inteiro
`o`	octal sem sinal (nem zero inicial)
`x`	hexadecimal sem sinal
`u`	decimal sem sinal
`c`	carácter
`s`	string
`f`	float
`e`	float no formato [-]*m.nnnnnn*E[□]*xx* (6 casas decimais, por omissão)
`g`	mais curta de `%e` ou `%f` zeros não significativos não são mostrados
`%`	mostra o carácter `%` e não um argumento

Exemplos:

Valor	especificador de conversão	Saída
360	%10d	|_______360|
360	%-10d	|360_______|
360	%10o	|_______550|
360	%010o	|0000000550|
360	%-10x	|168_______|
360	%010x	|0000000168|
3.14159265	%10f	|__3.141593|
3.14159265	%10.3f	|_____3.142|
3.14159265	%-10.3f	|3.142_____|
3.14159265	%10.0f	|_________3|
3.14159265	%10g	|___3.14159|
3.14159265	%10e	|3.141593e+00|
3.14159265	%10.2e	|__3.14e+00|
"string teste!"	%s	|string teste!|
"string teste!"	%10s	|string teste!|
"string teste!"	%.10s	|string tes|
"string teste!"	%-10s	|string teste!|
"string teste!"	%.15s	|string teste!|
"string teste!"	%-15s	|string teste!__|
"string teste!"	%15.10s	|_____string tes|
"string teste!"	%-15.10s	|string tes_____|

Devemos ter em atenção que a função `printf` usa o seu primeiro argumento para determinar qual o número de argumentos que se seguem e qual o seu tipo. Se o número de argumentos não for o especificado ou forem de tipo diferente do especificado é provável que se obtenham dados incorrectos no monitor. Se a variável `s` contém uma *`string`* então

```
printf(s);
```

falha se a *`string`* contiver o carácter '`%`'. Teremos então de usar

```
printf("%s",s);
```

A precisão pode ser dada como argumento usando *, caso em que o valor da precisão é o valor do próximo argumento, que deve ser, obrigatoriamente, um inteiro. Para mostrar no máximo `max` caracteres da *`string`* `s` usa-se a instrução:

```
printf("%.*s",max,s);
```

3.2 Entrada de dados

Para a entrada de dados existe uma função análoga a `printf`, a função `scanf` com a sintaxe:

`scanf(string_de_controlo [, arg1, arg2, ...]);`

`scanf` lê caracteres da entrada padrão (o teclado) interpretando-os segundo o que é especificado na *`string`*`_de_controlo` e guardando os resultados nos argumentos seguintes.

Estes argumentos seguintes têm que ser obrigatoriamente os endereços das variáveis onde se pretende armazenar os resultados. Para indicar o endereço de uma variável usa-se o operador & antes do nome da variável. No caso de variáveis do tipo *array*, o nome da variável é já o endereço do seu primeiro elemento, pelo que não é necessário usar o operador &. A função `scanf` pára quando termina a *string*_de_controlo ou quando ocorre um erro.

Esta *string*_de_controlo pode conter:

- Espaços ou tabulações, que são ignorados.
- Caracteres normais (não '%'), que deverão corresponder aos próximos caracteres não brancos da sequência introduzida.
- Especificadores de conversão, que têm um formato semelhante ao já indicado para a função `printf`.

Exemplo:

```
int x;
char nome[80];

scanf("%d %s", &x,nome);
```

3.3 Exemplos

Programa para escrever uma mensagem no écran

```
#include <stdio.h> /*inclusão de biblioteca*/

void main()
{
    printf("hoje iniciamos a aprendizagem da linguagem C");
}
```

Notas:

Atenção à forma de inclusão de uma biblioteca
Atenção à omissão de controlo
Atenção à representação de uma *string*

Programa para ler um número e fazer a sua escrita

```
#include <stdio.h> /*inclusão de biblioteca */

void main()
{
    int num;

    printf("Introduza um número");
    scanf("%d",&num);
    printf("O número introduzido foi: %d",num);
}
```

4 Operadores e Expressões

Uma expressão é uma sequência de operadores e operandos, possivelmente contendo também parêntesis, que denotam um valor. Os operandos podem ser constantes, variáveis (incluindo *array*'s) e chamadas de funções.

A linguagem ***C*** possui um número elevado de operadores divididos em 4 grandes classes: **aritméticos**, **relacionais**, **lógicos** e **para manipulação de *bits***. Além destes, a linguagem C tem ainda um conjunto de operadores para tarefas específicas. Associado com cada operador há duas propriedades: a **associatividade** e a **prioridade**. Se uma expressão possuir apenas operadores com o mesmo nível de prioridade então a expressão será avaliada da direita para a esquerda (excepto se o operador for associativo para a esquerda). Se a expressão tem operadores com diferentes níveis de prioridade então são primeiro avaliados os operadores com maior prioridade, depois os seguintes e assim sucessivamente. O uso de parêntesis permite alterar a ordem pela qual os operadores são avaliados.

4.1 Operadores aritméticos

Existem os seguintes operadores aritméticos:

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

Quando se aplica o operador / a inteiros ou caracteres o resultado será a divisão inteira, truncando qualquer parte fraccionária. Por exemplo `10/4` terá como resultado `2`. O operador `%` é o resto da divisão inteira, como tal não pode ter como operandos valores do tipo `float` ou `double`.

O seguinte programa ilustra a divisão inteira e o operador `%`:

```
void main()
{
    int x,y;

    x=10;
    y=3;
    printf("%d",x/y);          /* Mostra 3 */
    printf("%d",x%y);          /* Mostra 1 */

    x=1;
    y=2;
    printf("%d %d",x/y,x%y);   /* Mostra 0 1 */
}
```

O operador - pode ainda ser aplicado a apenas um operando tendo como efeito alterar--lhe o sinal (multiplicação por -1).

Além destes operadores, a linguagem ***C*** possui ainda dois operadores especiais que normalmente não existem nas outras linguagens. São os operadores de **incremento** e

decremento: ++ e --, respectivamente. O operador ++ adiciona 1 ao seu operando e o operador -- subtrai 1 ao seu operando. Temos então as seguintes equivalências:

```
x++; ⇔ x = x+1;
x--; ⇔ x = x-1;
```

Estes operadores podem ser colocados antes ou depois do seu operando. Por exemplo, podemos escrever x=x+1 como x++ ou ++x. Há, no entanto, uma diferença quando estes operadores são usados em expressões. Quando o operador precede o operando o compilador incrementa ou decrementa o operando antes de o utilizar. Quando o operador sucede ao operando é usado o valor do operando antes dele ser incrementado ou decrementado. Consideremos a seguinte situação:

```
x=10;
y=++x;
```

Neste caso a variável y vai ficar com o valor 11 porque o compilador incrementa a variável x e depois atribui o seu valor à variável y. No entanto, se escrevermos:

```
x=10;
y=x++;
```

y ficará com o valor 10 porque o compilador primeiro atribui o valor da variável x à variável y e só depois incrementa a variável x. Em ambos os casos x fica com o valor 11.

Os operadores de incremento e decremento podem apenas ser aplicados a variáveis, sendo ilegal uma expressão do tipo (i+j)++.

Os operadores aritméticos associam-se da esquerda para a direita (excepto os operadores unários) e a sua prioridade é dada na seguinte tabela:

	Operador
Maior prioridade ↑	++ -- - (menos unário)
	* / %
Menor prioridade	+ -

4.2 Operadores Relacionais

Os operadores relacionais, tal como o nome indica, têm como objectivo obter a relação que os operandos podem ter entre si. Na linguagem ***C*** estão definidos os seguintes operadores relacionais:

Operador	Significado
>	maior
<	menor
>=	maior ou igual
<=	menor ou igual
= =	Igual
!=	diferente

Subjacente aos operadores relacionais está a ideia de **verdadeiro** e **falso**. Em ***C***, verdadeiro é qualquer valor diferente de zero e falso é zero. As expressões que usam operadores relacionais

devolvem 0 para falso e 1 para verdadeiro. O programa seguinte ilustra a utilização dos operadores relacionais e mostra o resultado como 0 ou 1.

```
void main()
{
  int i,j;

  printf("Intruduza dois números: ");
  scanf("%d %d",&i,&j);

  printf(" %d == %d é %d.\n",i,j,i==j);
  printf(" %d != %d é %d.\n",i,j,i!=j);
  printf(" %d <= %d é %d.\n",i,j,i<=j);
  printf(" %d >= %d é %d.\n",i,j,i>=j);
  printf(" %d > %d é %d.\n",i,j,i>j);
  printf(" %d < %d é %d.\n",i,j,i<j);
}
```

Os operadores relacionais podem ser aplicados a operandos de qualquer dos tipos de dados básicos. Os operadores relacionais têm uma prioridade inferior aos operadores aritméticos, pelo que uma expressão do tipo `i < x-1` será considerada como `i < (x-1)`.

4.3 Operadores lógicos

Os operadores lógicos são usados para efectuar as operações lógicas elementares: E lógico, OU lógico e Negação. Os operadores lógicos usados na linguagem ***C*** são:

Operador	significado
&&	E lógico
||	OU lógico
!	Negação

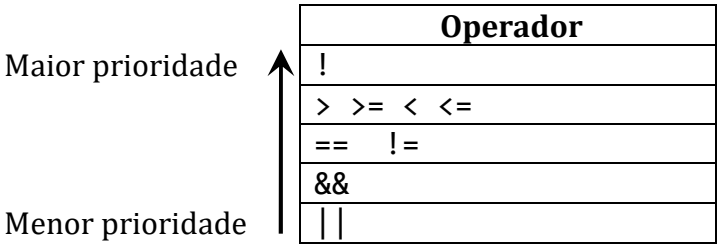
O programa seguinte mostra a utilização dos operadores lógicos:

```
void main()
{
  int i,j;

  printf("Introduza dois números: ");
  scanf("%d %d",&i,&j);

  printf(" %d E %d é %d.\n",i,j,i&&j);
  printf(" %d OU %d é %d.\n",i,j,i||j);
  printf(" NÃO %d é %d.\n",i,!i);
}
```

Os operadores lógicos têm também uma prioridade inferior relativamente aos operadores aritméticos. A prioridade dos operadores relacionais e lógicos é dada na tabela seguinte:



Operador
!
> >= < <=
== !=
&&
||

Tal como nas expressões aritméticas, podem usar-se parêntesis para alterar a ordem de avaliação natural da expressão. Assim

```
1 && !0 || 1
```

terá o valor `1` (verdadeiro) porque o `!` tem maior prioridade e o `E` fica verdadeiro. No entanto, se a expressão for

```
1 && !(0 || 1)
```

terá o valor `0` (falso) porque (`0 || 1`) é verdadeiro e a negação altera o seu valor para falso, implicando que o `E` seja falso.

Não esquecer que uma expressão relacional ou lógica produz sempre um resultado que é 0 ou 1. Portanto, o programa seguinte não só está correcto como mostra o número 1 no monitor.

```
void main()
{
  int x;

  x=100;
  printf("%d",x>10);
}
```

4.4 Operadores para manipulação de *bits*

Contrariamente à maioria das outras linguagens, a linguagem ***C*** possui um conjunto completo de operadores para manipulação de *bits*, também chamados **operadores *bit* a *bit***. Como o ***C*** foi projectado para substituir o ***assembly*** na programação, deve suportar operações que podem ser feitas em ***assembly***. Estes operadores permitem testar, alterar ou deslocar os *bits* num *byte* ou numa word e só podem ser aplicados a operandos inteiros, i. e., do tipo int, char, short e long, com ou sem sinal. Não se podem aplicar estes operadores aos tipos float, double, void. Há 6 operadores para manipulação de *bits*:

Operador	significado
&	E *bit* a *bit*
|	OU *bit* a *bit*
^	OU exclusivo *bit* a *bit*
~	complemento para 1
<<	*shift* à esquerda
>>	*shift* à direita

O ***E***, ***OU***, ***OU exclusivo*** e ***complemento para 1*** para manipulação de *bits* operam como as operações lógicas correspondentes só que *bit* a *bit*.

Os operadores para manipulação e *bits* são usados normalmente em *drivers*, tais como programas para o uso de *modem´*s, de impressoras, porque permitem mascarar certos *bits*, por exemplo o *bit* de paridade.

O ***E bit a bit*** permite colocar *bits* a '0'. Para colocar o *bit* de paridade a '0' basta fazer o E *bit* a *bit* com um *byte* cujos *bits* 1 a 7 sejam '1' e o *bit* 8 seja '0'. A expressão `x & 127` representa o E entre os *bits* de `x` e os *bits* que compõem o número `127`.

```
  1 1 0 0 0 0 0 1  x
& 0 1 1 1 1 1 1 1  127
  0 1 0 0 0 0 0 1
```

O ***OU bit a bit*** permite colocar *bits* a '1' pois qualquer *bit* que estiver a '1' em qualquer dos operandos força o correspondente *bit* do resultado a ficar a '1', por exemplo:

```
  1 0 0 0 0 0 0 0   128 em binário
| 0 0 0 0 0 0 1 1     3 em binário
  1 0 0 0 0 0 1 1   resultado
```

O ***OU exclusivo*** coloca um *bit* a '1' apenas no caso dos *bits* comparados serem diferentes, por exemplo 127^120:

```
  0 1 1 1 1 1 1 1   127  em binário
| 0 1 1 1 1 0 0 0   120  em binário
  0 0 0 0 0 1 1 1   resultado
```

Não confundir estes operadores com os correspondentes operadores lógicos `&&` e `||`, que implicam uma avaliação da esquerda para a direita. Por exemplo, se `x` for 1 e `y` for 2, então `x & y` é `0` e `x && y` é `1`. De notar que os operadores lógicos e relacionais produzem sempre um resultado que é '`0`' ou '`1`', enquanto as correspondentes operações *bit* a *bit* podem produzir qualquer valor.

Os operadores de deslocamento `>>` e `<<` movem todos os *bits* do seu operando esquerdo para a direita ou esquerda um número de *bits* dado pelo seu operando à direita (que tem que ser positivo). A sintaxe do *shift* à direita e do *shift* à esquerda é então, respectivamente:

```
variável >> nº de posições        shift à direita
variável << nº de posições        shift à esquerda
```

À medida que são eliminados *bits* de um dos lados são introduzidos '`0`'s no outro. De notar que um deslocamento não é uma rotação, os *bits* retirados de um dos lados não são introduzidos no outro, são perdidos.

A instrução `x << 2` desloca o valor de `x` para a esquerda 2 posições, o que é equivalente a uma multiplicação por 4. Cada posição deslocada para a esquerda equivale a uma multiplicação por 2 e cada posição deslocada para a direita equivale a uma divisão por 2.

```
char x;                              valor de x
x=7;          0 0 0 0 0 1 1 1          7 em binário
x << 1;       0 0 0 0 1 1 1 0         14 em binário
x << 3;       0 1 1 1 0 0 0 0        112 em binário
x << 2;       1 1 0 0 0 0 0 0        192 em binário
x >> 1;       0 1 1 0 0 0 0 0         96 em binário
x >> 2;       0 0 0 1 1 0 0 0         24 em binário
```

4.5 Operadores de atribuição e expressões

Na linguagem ***C*** o operador de atribuição é o carácter '='. Expressões do tipo

```
x = x+2;
```

em que a variável do lado esquerdo é repetida imediatamente a seguir ao sinal = podem ser escritas na forma mais compacta

```
x += 2;
```

O operador += é também chamado operador de atribuição. Grande parte dos operadores que têm dois operandos, um à esquerda e outro à direita, têm um correspondente operador de atribuição *op*=, em que *op* pode ser:

```
+  -  *  /  %  <<  >>  &  ^  |
```

Se exp_1 e exp_2 são expressões, então

exp_1 op= exp_2;

é equivalente a

exp_1 = (exp_1) op (exp_2);

excepto que exp_1 é calculada apenas uma vez. De notar o uso dos parêntesis em exp_2 pois

```
x *= y + 1;
```

significa

```
x = x * (y + 1);
```

e não

```
x = x * y + 1;
```

4.6 Conversão de tipos

Quando um operador tem operandos de diferentes tipos (constantes ou variáveis), é efectuada uma conversão para um tipo comum segundo um pequeno conjunto de regras. Em geral, as únicas conversões automáticas são aquelas que convertem um tipo "menor" num tipo "maior" sem perder informação. Se nenhum dos tipos envolvidos é `unsigned`, podemos aplicar as seguintes regras pela ordem indicada:

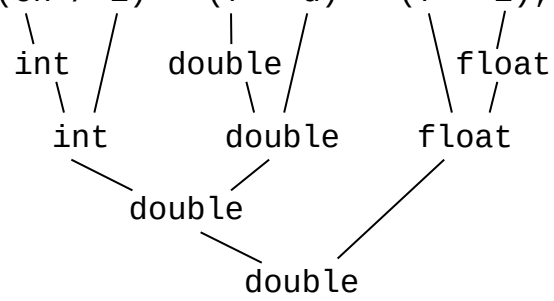
Se algum dos operandos é um `long double`, converte-se o outro para `long double`.
Senão, se algum dos operandos é um `double`, converte-se o outro para `double`.
Senão, se algum dos operandos é um `float`, converte-se o outro para `float`.
Senão, converte `char` e `short` para `int`.
Depois, se algum dos operandos é um `long`, converte-se o outro para `long`.

As regras de conversão são mais complicadas quando envolvem tipos `unsigned`.

Na figura seguinte pode ver-se um exemplo de conversão de tipos:

```
char ch;
int i;
float f;
double d;

x = (ch / i) + (f * d) - (f + i);
```



As conversões também ocorrem nas atribuições. O valor do lado direito é convertido para o tipo à esquerda, que será o tipo do resultado. Se `i` é um `int` e `c` um `char`, então em

```
i = c;
c = i;
```

o valor de `c` não é alterado, pois um carácter é convertido para inteiro extendendo o seu *bit* mais à esquerda e um inteiro é copiado para um carácter eliminando os *bits* de maior peso. No entanto se trocássemos a ordem das instruções poderia haver perda de informação.

Se `x` é um `float` e `i` um `int`, então `x = i` e `i = x` provocam ambos conversões: de `float` para `int` trunca-se a eventual parte fraccionária.

As conversões ocorrem também quando são passados argumentos para funções. Não existindo protótipo da função, `char` e `short` são convertidos em `int` e `float` em `double`.

As conversões de tipo podem também ser forçadas explicitamente em qualquer expressão usando um operador unário chamado *`cast`*. A sintaxe é:

```
(tipo) expressão
```

Se por exemplo a variável `x` é inteira e queremos garantir que o computador avaliará a expressão `x/2` como `float` (para que não seja perdida a parte fraccionária) podemos escrever:

```
(float) x/2
```

Neste caso o *`cast`* está associado a `x`, alterando o seu tipo para `float`. Devido à existência de um `float` na expressão, o `2` é também convertido para `float` e o resultado será também do tipo `float`. No entanto, se escrevermos:

```
(float) (x/2)
```

o computador efectua a divisão inteira e modifica o tipo do resultado para `float`, não avaliando a parte fraccionária.

O *`cast`* é considerado um operador unário tendo a mesma prioridade que os outros operadores unários já definidos.

No caso dos argumentos serem declarados no protótipo da função, a conversão de tipos dos argumentos é automática cada vez que a função é invocada. Assim, se a função `sqrt` tiver o protótipo

```
double sqrt(double);
```

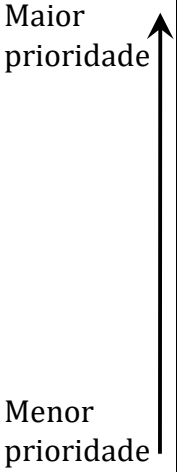
a chamada

```
r2 = sqrt(2);
```

força o inteiro 2 a ser convertido para o valor `double 2.0` sem necessidade de usar um *`cast`*.

4.7 Prioridade dos operadores

A prioridade dos operadores e a associatividade é apresentada na tabela abaixo:



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

De notar que os operadores *`bit a bit`* `&`, `^` e `|` têm uma prioridade inferior a `==` e `!=` pelo que para fazer um teste dos *bits* de uma expressão terão de ser utilizados parêntesis

```
(x & mask) == 0
```

5 Estruturas de controlo

As estruturas de controlo são a essência de qualquer linguagem de programação uma vez que determinam a sequência pela qual as instruções de um programa são executadas. As estruturas de controlo da linguagem ***C*** são semelhantes às da linguagem ***Pascal***.

5.1 Instruções simples e compostas

Na linguagem ***C*** o ponto e vírgula `;` é o terminador de instruções.

As chavetas { } são usadas para agrupar instruções em instruções compostas ou blocos de modo a serem sintacticamente equivalentes a uma instrução única. Assim uma instrução em sentido lato pode ser:

- Uma única instrução.
- Um conjunto de instruções.
- Nenhuma instrução (instrução vazia).

Uma instrução composta ou bloco tem a estrutura:

```
{
  declarações (opcional)
  instruções
}
```

5.2 Estruturas de Decisão Condicional

5.2.1 Instrução `if..else`

Uma das estruturas de decisão é a instrução `if..else` que admite diversas variantes. A forma genérica desta instrução é:

```
if condição
      instrução1;
else
      instrução2;
```

sendo o `else` opcional. A condição é uma expressão que é avaliada. Se for verdadeira (tiver um valor não nulo) é executada a $\texttt{instrução}_1$, se for falsa e existir o `else` então é executada a $\texttt{instrução}_2$. É claro que cada uma destas instruções pode ser na realidade um bloco delimitado por chavetas.

Podemos assim ter os seguintes casos:

`if (condição)`
`  instrução;`
	`if (condição)`
	`{`
	`  conj. instruções;`
	`}`

`if (condição)`
`  instrução1;`
`else`
`  instrução2;`
	`if (condição)`
	`{`
	`  conj. instruções1;`
	`}`
	`else`
	`{`
	`  conj. instruções2;`
	`}`

As instruções `if` podem ser encadeadas, podendo ser usada uma combinação entre as duas sintaxes (com e sem `else`). Há, no entanto, um caso que pode levar a confusão. Consideremos a seguinte sequência:

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

Podemos considerar que se pretende especificar

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

ou

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

A duas interpretações foram mostradas usando identações diferentes, no entanto o compilador de ***C*** é insensível à identação. Felizmente, a linguagem ***C*** tem uma regra simples que permite resolver esta ambiguidade . Um `else` pertence ao `if` mais interior (o último) que ainda não tem associado um `else`. Sendo assim a interpretação correcta seria a primeira. Se se pretender a segunda interpretação terão de ser utilizadas chavetas, como mostrado a seguir:

```
if (condição1)
{
      if (condição2)
            instrução1
}
else
      instrução2
```

Uma maneira de evitar esta ambiguidade é evitar o uso combinado das duas sintaxes da instrução `if`. Pode usar-se uma **instrução nula** se necessário.

```
if (condição1)
      if (condição2)
            instrução1
      else
            ;      /* instrução nula*/
else
      instrução2
```

5.2.2 Instrução `else..if`

Uma construção comum em ***C*** são os **`else..if`**'s encadeados. A sua sintaxe é:

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

```
if (condição1)
{
      conj. instruções1
}
else if (condição2)
{
      conj. instruções2
}
.
.
else
{
      conj. instruçõesn
}
```

Esta sequência de `else..if`´s é uma maneira de implementar uma decisão múltipla. O computador avalia as condições de cima para baixo e logo que encontre uma verdadeira executa a instrução que lhe está associada, ignorando o resto da cadeia. Se nenhuma das condições for verdadeira então é executada a instrução correspondente ao último `else`. Este último `else` pode não existir, caso em que, se as condições forem todas falsas, nenhuma instrução é executada.

Consideremos o seguinte exemplo que ilustra o uso das estruturas `if`:

```
void main()
{
  int n_secreto;
  int palpite;

  n_secreto = rand();        /* gera um nº aleatório */

  printf("Qual o seu palpite? ");
  scanf("%d",&palpite);

  if (palpite == n_secreto)
  {
       printf("Acertou! \n");
       printf("%d é o nº secreto.",n_secreto);
  }
  else if (palpite > n_secreto)
       printf("  Errado! Muito alto. \n");
  else
       printf("  Errado! Muito baixo. \n");
}
```

Dado que um `if` apenas testa o valor numérico de uma expressão, podem ser usadas determinadas formas abreviadas, por exemplo, pode usar-se

```
if (expressão)
```

em vez de

```
if (expressão != 0)
```

uma vez que para o ***C*** qualquer valor diferente de zero é assumido como o valor lógico verdadeiro.

5.2.3 Instrução `switch`

Apesar da instrução `else..if` permitir construir uma estrutura de escolha múltipla, a linguagem ***C*** fornece uma instrução específica para isso, a instrução `switch`. Nesta instrução é testada sucessivamente uma variável para verificar se coincide com uma lista de valores inteiros (ou caracteres).

A sintaxe desta instrução é:

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

Cada um dos casos é rotulado por uma ou mais constantes inteiras ou expressões com constantes, i. e., `exp_i` pode ser um inteiro, um carácter ou uma expressão de constantes. Se a variável coincidir com um dos casos a execução começa a partir desse ponto. Todos os casos

`exp_i` têm de ser diferentes. O caso com rótulo `default` é executado se não houver mais nenhuma coincidência. O uso de `default` é opcional, se não for usado e nenhuma coincidência for encontrada não será executada nenhuma acção. É claro que `instr_i` pode ser uma única instrução ou um conjunto de instruções. Como já foi dito, quando se verifica uma coincidência entre o valor da variável e um caso, a execução do programa prossegue a partir desse ponto, executando tudo o que vem a seguir, inclusive as instruções correspondentes aos casos posteriores. Para o evitar pode usar-se a instrução `break` que força a saída imediata do `switch`. A razão pela qual a execução de um `switch` não se limita às instruções de um `case` mas continua a partir desse ponto é que essa característica tem bastante utilidade em diversas situações. Por exemplo, podemos ter:

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

Neste exemplo, é executada a `instr_2` se a variável tiver o valor `exp_1` ou `exp_2`.

É boa prática usar `break` após o último caso, embora tal não seja necessário.

A diferença fundamental entre o `switch` e o if é que o switch apenas testa igualdade enquanto a expressão condicional do if pode ser de qualquer tipo.

A instrução switch é bastante usada para processar comandos a partir do teclado, tais como a selecção de opções num *menu*. Por exemplo podemos ter uma função que devolva um valor consoante a opção seleccionada.

```
void menu()
 {
  char ch;

  printf(" Ler ficheiro    -  1\n ");
  printf(" Gravar ficheiro -  2\n ");
  printf(" Eliminar ficha  -  3\n ");
  printf(" Adicionar ficha -  4\n ");
  printf("Escolha opção: ");
  scanf("%d",&ch);

  switch (ch)
   {
       case '1' : ler_fich(); break;
       case '2' : grava_fich(); break;
       case '3' : apaga_ficha(); break;
       case '4' : adiciona_ficha(); break;
       default  :
            printf("Opção inválida!");break;
   }
 }
```

Exercícios resolvidos:

1 - Escreva um programa que leia um caracter e atribua a uma variável o código ASCII deste.

```
#include <stdio.h>
void main()
 {
  int num;
  char car;

       printf("Introduza um caracter");
       scanf("%c",&car);
       num = car;
       printf("O código ASCII do caracter e: %d",num);
 }
```

2 - Escreva um programa que leia dois números inteiros **a** e **b** e verifique se **a** é múltiplo de **b** ou se **b** é múltiplo de **a**.

```
#include <stdio.h>
void main()
 {
  int a,b;

  printf("Introduza o primeiro numero");
  scanf("%d",&a);
  printf("Introduza o segundo numero");
  scanf("%d",&b);
  if (!(a % b))
       printf(" O numero %d e multiplo de %d",a,b);
  else if (!(b% a))
       printf(" O numero %d e multiplo de %d",b,a);
  else
       printf("os numeros não são multiplos");
 }
```

3 - Escreva um programa que leia um caracter qualquer e verifique se este é um dígito, uma letra minúscula ou uma letra maiúscula.

```
#include <stdio.h>
void main()
 {
  char car;

  printf("Introduza um caracter");
  scanf("%c",&car);
  if (car >= '0' && car <= '9')
       printf("e um digito");
  else if (car >= 'a' && car <= 'z')
       printf("e uma letra minuscula");
  else if (car >= 'A' && car <= 'Z')
       printf("e uma letra maiuscula");
  else
       printf("e outro caracter.");
 }
```

5.3 Estruturas de Repetição ou Ciclos

Os ciclos permitem ao computador executar uma sequência de instruções repetidamente até que deixe de ser satisfeita uma determinada condição. Em ***C*** estão definidos três tipos de ciclos: ciclos **`for`**, ciclos **`while`** e ciclos **`do..while`**.

5.3.1 Ciclo `while`

O ciclo `while` tem a sintaxe:

```
while (expressão)
       instrução;
```

em que o tipo da expressão deve ser escalar. A expressão é avaliada, se for não nula (verdadeira) a instrução é executada e a expressão é em seguida re-avaliada. Este ciclo continua até que a expressão tenha um valor nulo (falso), passando então o controlo do programa para a instrução na linha seguinte ao ciclo.

A instrução que constitui o corpo do ciclo pode na realidade ser a instrução nula, uma única instrução ou um grupo de instruções.

Como exemplo deste tipo de ciclo temos a seguinte função que simplesmente espera até que se introduza o carácter 'A'.

```
void espera()
 {
  char c;

  c = '\0';
  while (c!='A')
       scanf("%c",&c);
 }
```

5.3.2 Ciclo `for`

O ciclo `for` tem a sintaxe

```
for (expr_1 ; expr_2 ; expr_3)
       instrução;
```

sendo equivalente ao seguinte ciclo `while`

```
expr_1;
while (expr_2)
 {
  instrução;
  expr3;
 }
```

Normalmente `expr_1` é uma inicialização (com uma instrução de atribuição), `expr_2` é uma condição (expressão relacional), que testa a variável de controlo do ciclo para verificar quando deve sair do ciclo, e `expr_3` é um incremento que define como a variável de controlo do ciclo deve ser alterada cada vez que o ciclo é executado. O ciclo `for` será executado enquanto a

condição for verdadeira. Quando a condição se tornar falsa o programa prossegue na instrução a seguir ao ciclo.

Em sentido lato `expr_1` e `expr_3` são atribuições ou chamadas de funções e `expr_2` é uma expressão relacional. Qualquer uma desta três partes pode ser omitida, apesar de terem de ser mantidos os caracteres ponto e vírgula. Quando é omitida expr_2 é substituída por uma constante não nula pelo que o teste é sempre verdadeiro. Por exemplo

```
for (;;)
       instrução;
```

é um ciclo infinito equivalente ao ciclo

```
while (1)
       instrução;
```

Um ciclo infinito pode ser interrompido se no seu corpo existir uma instrução `break` ou `return`.

O programa seguinte permite escrever todos os inteiros entre 1 e 100.

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

  for(x=1; x<=100; x++)
            printf("%d\n",x);
 }
```

É claro que o ciclo for permite ser executado pela ordem inversa. Por exemplo, para os inteiros de 100 a 1 poderíamos usar

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

  for(x=100; x>0; x--)
            printf("%d\n",x);
 }
```

Também são permitidas outras operações com a variável de controlo do ciclo além do incremento e decremento. Para mostrar os números entre 5 e 95 de 5 em 5 pode usar-se:

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

  for(x=5; x<100; x=x+5)
            printf("%d\n",x);
 }
```

A condição é sempre testada no início do ciclo. Isto significa que na realidade o computador pode não executar o ciclo se a condição for falsa. O ciclo

```
x=10;
for (y=10; y!=x; ++y)        printf("%d ",y);
```

nunca será executado porque de facto x e y são iguais quando o programa entra no ciclo.

5.3.3 Ciclo `do..while`

Como já vimos, os ciclos `while` e os ciclos `for` testam a condição de ciclo no início do ciclo. Pelo contrário, o ciclo `do..while` testa a condição no fim do ciclo pelo que as instruções que fazem parte do corpo do ciclo são executadas pelo menos uma vez. A forma genérica do ciclo `do..while` é:

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

A instrução é executada e depois é avaliada a expressão. Se for verdadeira a instrução é executada novamente e assim sucessivamente até que a expressão seja falsa (valor nulo).

O ciclo `do..while` é análogo ao ciclo `repeat..until` da linguagem Pascal, excepto que neste caso o ciclo é repetido até a condição ser falsa enquanto no Pascal era até a condição ser verdadeira.

O programa seguinte lê inteiros do teclado até que seja introduzido um número não superior a 100.

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

  do {
       scanf("%d",&n);
  } while (n>100);
}
```

Uma das utilizações deste ciclo é a validação das opções de um menu, como no seguinte exemplo:

```
void menu()
{
  char ch;

  printf(“ Ler ficheiro    -  1\n ”);
  printf(“ Adicionar ficha -  2\n ”);
  printf(“ Gravar ficheiro -  3\n ”);

  do {
      printf(“ Qual a sua opção?\n ”);
      scanf(“%c”,&ch);
  } while (ch!= ‘1’ && ch!= ‘2’ && ch!= ‘3’);

  switch (ch)
   {
      case ‘1’ : ler_fich(); break;
      case ‘2’ : adiciona_ficha(); break;
      case ‘3’ : grava_fich(); break;
}
```

Após mostrar as opções, o programa entra em ciclo até que se introduza uma opção válida.

5.4 Instruções `break` e `continue`

Por vezes é útil sair de um ciclo sem ser pelo teste no início ou no fim do ciclo. A instrução `break`, que já foi usada para sair da instrução `switch`, força a saída do ciclo `for`, `while` ou `do.. while` mais interior sem testar a condição de fim de ciclo.

No programa seguinte o ciclo `for` mostra no monitor os números de 0 a 10 e depois o `break` termina o ciclo, ultrapassando o teste condicional `t<100` incluído no ciclo.

```
#include <stdio.h>
void main()
{
  int t;

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

É importante realçar que a instrução `break` apenas força a saída do ciclo mais interior. Consideremos, por exemplo, o programa seguinte

```
#include <stdio.h>
void main()
{
  int t,count;

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

O programa mostrará os números 1 a 9 no monitor 100 vezes. De cada vez que é executado o `break` o programa passa a executar o ciclo `for` exterior. De igual forma, um `break` numa instrução `switch` apenas afecta esse `switch` e não qualquer ciclo em que possivelmente esse `switch` ocorra.

A instrução `continue` obriga a que o ciclo `for`, `while` ou `do..while` inicie um novo ciclo saltando qualquer código intermédio, i. e., se se tratar de um ciclo `while` ou `do..while` a condição de teste é imediatamente executada, se for um ciclo `for` é executada imediatamente a expressão correspondente ao incremento. A instrução `continue` apenas se aplica a ciclos, não à instrução `switch`.

No programa do seguinte exemplo serão mostrados no monitor apenas os números pares.

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

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

De cada vez que `x` é um número ímpar, a instrução condicional `if` é executada porque `x%2` é `1`, correspondendo ao valor lógico verdadeiro. Um número ímpar leva assim à execução da instrução `continue` que provoca uma nova iteração do ciclo, passando por cima da instrução `printf`.

5.5 Instrução `return`

Como já tivemos oportunidade de verificar, uma função permite encapsular uma determinada tarefa, que pode depois ser usada sem nos preocuparmos com a sua implementação. Se as funções forem convenientemente concebidas e implementadas, é possível ignorar como a tarefa é feita, basta saber o que é feito.

Até agora temos usado funções já disponíveis na linguagem ***C***, tais como as funções `printf()` e `scanf()`, e funções implementadas por nós mas do tipo `void`. Mas é possível retornar valores das funções. O retorno de valores de funções é feito usando a instrução `return`, que tem a seguinte sintaxe

```
return expressão;
```

em que a `expressão` é opcional e pode ser um valor ou uma expressão cujo resultado evolui ou é convertido para o tipo de dados devolvido pela função.

A instrução `return` faz com que o controlo do programa passe para a função chamadora. Consideremos como exemplo o seguinte programa que efectua a cálculo da área de um quadrado:

```
#include <stdio.h>
/* ======================================
  função que retorna a área de um quadrado
       la - lado do quadrado   ===========*/
float area(float la la)
 {
  return(la*la);
 }

/* ======================================
  função efectua a leitura do lado de um quadrado
  e retorna o valor lido=================*/
float leitura()
 {
  float la;

  printf("Introduza a medida do lado do quadrado ");
  scanf("%f",&la);
  return(la);
 }

/* ======================================*/
void main()
{
  float la;
  la = leitura();
  printf("A área do quadrado de lado %f é %f",la,area(la));
}
```

Alternativamente, a função main poderia ser:

```
void main()
{
  printf("A área do quadrado é %d",area(leitura()));
}
```

mas o resultado no monitor seria pouco explícito... Verifique porquê!

6 *Array*´s e Ponteiros

6.1 *Array*'s

Um *array* é um conjunto de variáveis do mesmo tipo que podem ser referenciadas por um identificador comum. Em ***C*** um *array* consiste num conjunto de posições de memória contíguas, o menor endereço corresponde ao primeiro elemento e o maior endereço corresponde ao último elemento. Um *array* pode ter apenas uma dimensão (*array* unidimensional ou vector) ou várias (*array* multidimensional ou matriz). Como já foi referido, em ***C*** os *array*'s são declarados da mesma forma que uma variável elementar só que se coloca a dimensão entre parêntesis rectos a seguir à identificação da variável:

```
tipo nome_variável[dimensão];
```

em que `tipo` determina o tipo básico de cada elemento do *array*.

O índice que permite tratar individualmente cada um dos elementos de um *array* inicia-se em `0` (zero). O seguinte excerto de programa declara um *array* de inteiros com 10 elementos (`x[0]` a `x[9]`) e preenche os elementos com os valores de `0` a `9`, usando um ciclo `for(..)`.

```
void main()
{
  int x[10];
  int t;

  for(t=0;t<10;++t)
       x[t]=t;
...
}
```

A linguagem ***C*** não verifica os limites de um *array*, pelo que podemos referenciar um elemento não existente. Assim se ultrapassarmos os limites de um *array* numa operação de atribuição estaremos a atribuir um valor a alguma outra variável ou mesmo a um pedaço de código do programa. O programador deve garantir que os limites dos *array*'s não serão ultrapassados.

A linguagem ***C*** permite *array*'s multidimensionais. Para declarar um *array* bidimensional de inteiros com dimensão 10x20 usa-se

```
int tab[10][20];
```

sendo a primeira dimensão referente ao número de linhas e a segunda ao número de colunas. Assim, para aceder ao elemento na 4.ª linha e 2.ª coluna deste *array* usa-se `tab[3][1]`.

Mais genericamente, podemos ter *array*'s com k dimensões, declarados usando a sintaxe:

```
tipo  nome_variável[dimensão1][dimensão2]…[dimensãok];
```

6.2 String's

As *string*'s mais não são que *array*'s de caracteres. Aquando da declaração de uma variável do tipo *string* especifica-se a sua dimensão máxima, no entanto, quando se pretende usar a variável para armazenar uma *string* com um número de caracteres inferior ao máximo, usa-se o carácter nulo '\0' para indicar o fim da *string*. Por esta razão devem-se declarar os *array*'s de caracteres com mais um caracter que a maior *string* que vão conter. Se pretendermos declarar uma *string* para armazenar um conjunto de 10 caracteres devemos escrever

```
char st[11];
```

reservando espaço para o caracter nulo '\0' no fim da *string*.

Algumas das funções específicas para o tratamento de *string's* são:

```
strlen()
strcmp()
strcpy()
strcat()
```

Estas funções estão definidas na biblioteca *string.h*.

A função **strlen** tem a sintaxe

```
strlen(st)
```

em que st é uma *string*, e devolve o número de caracteres de st, não contabilizando o caracter nulo.

Exemplo:

```
#include <stdio.h>
#include <string.h>
void main()
{
  char st[20] = "Estruturas";

  printf("A string tem %d caracteres\n", strlen(st));
}
```

A função **strcmp** tem a sintaxe

```
strcmp(st1, st2)
```

e compara duas *string's*, st1 e st2, devolvendo 0 se forem iguais. Se a *string* st1 for lexicograficamente superior a st2 será devolvido um número inteiro positivo, se st1 for menor que st2 será devolvido um número negativo.

A função **strcpy** tem a sintaxe

```
strcpy(std, sto)
```

permitindo copiar uma *string* origem sto para uma *string* destino std. std tem de ter uma dimensão suficiente para armazenar sto.

Exemplo:

```
#include <stdio.h>
#include <string.h>
void main()
{
  char st[11];

  strcpy(st, “Estruturas”);
}
```

A função `strcat` tem a sintaxe

`strcat(std, sto)`

e acrescenta a *`string`* `sto` no fim da *`string`* destino `std`. O tamanho da *`string`* resultante é `strlen(std)+strlen(sto)`.

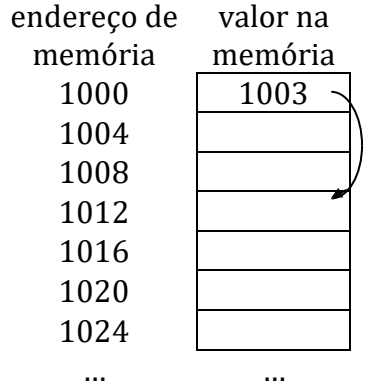
Exemplo:

```
#include <stdio.h>
#include <string.h>
void main()
{
  char st1[20] = “Estruturas”;
  char st2[11];

  strcpy(st2, “ de Dados”);
  strcat(st1, st2)
  printf("%s", st1);
}
```

6.3 Ponteiros

Um ponteiro é uma variável que permite guardar o endereço de outra variável. Assim quando uma variável contém o endereço de memória de outra variável diz-se que a primeira aponta para a segunda.



Um ponteiro mais não é que um endereço.
Para declarar uma variável como ponteiro usa-se a sintaxe:

`tipo *nome_variável;`

em que tipo pode ser qualquer tipo válido da linguagem ***C***. Por exemplo

```
int *tem, *x;
char *p;
```

A variável x vai permitir armazenar o endereço de uma variável inteira e não um valor inteiro.

Estão definidos dois operadores especiais para trabalhar com ponteiros. Os operadores unários * e &. O operador & devolve o endereço do seu operando, por exemplo

```
end_x = &x;
```

coloca em end_x o endereço de memória onde se encontra armazenada a variável x, dizendo-se que end_x aponta para x. De notar que o endereço de uma variável não tem nada a ver com o valor nela armazenado. O operador * devolve o valor armazenado no endereço de memória dado como operando. Se por exemplo o ponteiro end_x contém o endereço da variável x, então a instrução de atribuição

```
y = *end_x;
```

coloca na variável y o valor da variável x.

Podemos lembrar o significado destes operadores como:

& - endereço de ...
* - no endereço ...

Mas, numa instrução do tipo

```
y = *px;
```

como é que o compilador sabe o número de *bytes* a copiar do endereço px para a variável y? Para isso é que é necessário declarar o tipo base do ponteiro. O compilador assume que o ponteiro está a apontar para um dado deste tipo. Se, por exemplo, px tiver sido declarado como um ponteiro para inteiro, são copiados quatro[1] *bytes* para a variável y.

Se px é um ponteiro para inteiro e tiver o valor 2000, após a instrução

```
px++;
```

px terá o valor 2004 e não 2001. Isto porque de cada vez que o ponteiro é incrementado passa a apontar para o próximo inteiro, que se encontra no endereço 4 *bytes* acima. De forma análoga

```
px = px + 9;
```

faz com que px passe a apontar para o 9º elemento a seguir ao actual.

Considerando as seguintes variáveis

```
int x,y,*px;
```

e

```
x = 10;
px = &x;
```

[1] O tamanho de um inteiro depende do sistema, podendo ser diferente de quatro, mas pode ser conhecido usando a função *sizeof(int)*

a instrução

```
y = *px +1;
```

toma o valor apontado por px, adiciona-lhe 1 e atribui o resultado à variável y, enquanto

```
*px += 1;
```

incrementa o valor no endereço px, tal como ++*px e (*px)++. Neste último caso são necessários os parêntesis. Sem eles a instrução incrementa o ponteiro e não o valor por ele apontado, dado que os operadores unários * e ++ são associativos da direita para a esquerda.

Em ***C*** existe uma relação muito próxima entre os ponteiros e *array´s*. Consideremos que as variáveis st e p são declaradas da seguinte forma:

```
char st[80], *p;
```

Para colocar na variável p (que é um ponteiro para char) o endereço do primeiro elemento de st (*array* de caracteres ou *string*) podemos fazer a atribuição

```
p = &st[0];
```

Mas, na linguagem ***C***, o nome de um *array* sem índice é o endereço do elemento inicial, pelo que, no essencial, o nome de um *array* é um ponteiro para o *array*. Poderíamos então fazer a atribuição `p = &st[0];` usando a forma equivalente:

```
p = st;
```

Após isto, para aceder ao quinto elemento do *array* poderíamos usar

```
st[4]
```

ou

```
*(p+4)
```

Ainda de forma mais surpreendente, podemos aceder ao quinto elemento do *array* usando

```
*(st+4)
```

ou

```
p[4]
```

ou seja, podemos indexar ponteiros e usar os operadores de ponteiros com variáveis do tipo *array*. Assim a linguagem ***C*** permite duas formas de aceder a *array´s*: usando índices ou a aritmética de ponteiros. Como esta última pode ser mais rápida, e dado que a velocidade de execução é importante em programação, o seu uso é bastante comum em ***C***.

Pode-se também atribuir o endereço de qualquer outro elemento de um *array* a um ponteiro. Para atribuir ao ponteiro p o endereço do terceiro elemento do *array* usa-se:

```
p = &st[2];
```

Existe, no entanto, uma diferença fundamental entre o nome de um *array* e um ponteiro que não podemos esquecer. Um ponteiro é uma variável, pelo que são permitidas as atribuições `p=st` e `p++`. Pelo contrário, o nome de um *array* não é uma variável, sendo ilícito usar construções do tipo `st=p` e `st++`.

Tal como com qualquer outro tipo de dados, também é possível definir *array´*s de ponteiros. Para declarar um *array* com 10 ponteiros para inteiros usa-se

```
int *x[10];
```

e para atribuir o endereço da variável `y` ao terceiro elemento do *array* usa-se:

```
x[2]=&y;
```

O valor da variável `y` é dado por `*x[2]`.

Um *array* de ponteiros é equivalente a um ponteiro para ponteiro.

Os ponteiros são usados para fazer passagem de parâmetros por referência.

Quando se chama uma função com um parâmetro passado por valor o compilador efectua uma cópia do valor do parâmetro para usar dentro da função. Assim, qualquer alteração do parâmetro na função não surte efeito fora dela. Considerando o seguinte exemplo

```
#include <stdio.h>
void incrementa(int x)
{
  x++;
}

void main()
{
  int a;

  a = 10;
  incrementa(a);
  printf(“ a = %d”,a);
}
```

o valor que vai ser mostrado no monitor será 10 pois a função apenas incrementa a cópia do valor de `a` e não o valor contido na variável `a`. Por outras palavras, não há uma maneira directa de uma função chamada alterar uma variável na função chamadora, pois na linguagem ***C*** os parâmetros para as funções são passados por valor. A maneira de obter o efeito pretendido consiste em usar ponteiros. Neste exemplo se usarmos:

```
#include <stdio.h>
void incrementa(int *px)
{
  (*px)++; /*  ou *px=*px+1  ou *px+=1  */
}

void main()
{
  int a;

  a = 10;
  incrementa(&a);
  printf(" a = %d",a);
}
```

px vai ficar com o endereço onde está guardada a variável a, por isso a instrução (*px)++ incrementa directamente o conteúdo da memória e vai ser mostrado o valor 11 no monitor.

No seguinte programa temos uma variante que pretende mostrar a invocação de uma função dentro da função incrementa.

```
#include <stdio.h>
void inc(int *w)
{
  *w +=1;
}

void incrementa(int *px)
{
  inc(px);
}

void main()
{
  int a;

  a = 10;
  incrementa(&a);
  printf(" a = %d",a);
}
```

No seguinte exemplo mostra-se uma função para trocar o valor de duas variáveis mostrando uma versão errada (chamada usando **`troca(a, b);`**) e a versão correcta (chamada usando **`troca(&a, &b);`**).

Versão errada `troca(a, b);`

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

   temp = x;
   x = y;
   y = temp;
}
```

Versão correcta `troca(&a, &b);`

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

  temp = *px;
  *px = *py;
  *y = temp;
}
```

A passagem de *array*'s como parâmetro para funções é sempre efectuada por referência. O seguinte programa lê um conjunto de 10 valores reais e depois faz a sua escrita no monitor pela ordem inversa à de leitura

```
#include <stdio.h>
/*=============================================
Subrograma que le para um array um conjunto de 10 valores reais
=============================================*/
void le_ar(float ar[])    /* atenção à omissão da dimensão*/
{
  int i;

  for(i=0; i<=9; i++)
  {
       printf("Introduza o valor %d\n",i+1);
       scanf("%f",&(ar[i]));
  }
}
/*=============================================
subprograma que escreve os valores de
um array desde a posição 9 até à 0
=============================================*/
void escreve(float ar[])
{
  int i;

  for(i=9; i>=0; i--)
       printf("Valor na posição %d e %f\n",i+1, ar[I])
}

void main()
{
  float ar[10];

  le_ar(ar);
  escreve(ar);
}
```

O mesmo programa pode ser escrito usando ponteiros nas chamadas das funções. Resultaria então:

```
#include <stdio.h>
/*================================================
Subrograma que lê para um array um conjunto de 10 valores reais
================================================*/
void le_ar(float *ar)
{
  int i;

  for(i=0; i<=9; i++)
  {
       printf("Introduza o valor %d\n",i+1);
       scanf("%f",ar++);
  }
}
/*============================================
subprograma que escreve os valores de
um array desde a posição 9 até à 0
============================================*/
void escreve(float *)
{
  int i;

  ar+=9;
  for(i=9; i>=0; i--)
       printf("Valor na posição %d e %f\n",i+1, *(ar--));
}

void main()
{
  float ar[10];

  le_ar(ar);
  escreve(ar);
}
```

Apesar da grande utilidade dos ponteiros, devemos saber exactamente como usá-los. Com efeito, se um ponteiro tem um valor errado provavelmente produzirá um erro sem nexo, sendo muito difícil associá-lo à causa. Consideremos os seguinte programa:

```
#include <stdio.h>
void main()            /* PROGRAMA ERRADO */
{
  int x, *px;

  x = 10;
  *px = x;
}
```

Na última instrução o valor 10 é atribuído a um endereço de memória desconhecido. Aquando da declaração do ponteiro px não lhe foi atribuído nenhum valor pelo que ele conterá lixo. Assim estamos a destruir o conteúdo de uma zona da memória desconhecida que pode

conter parte do código do programa ou mesmo do sistema operativo podendo abortar o programa.

6.4 Alocação Dinâmica de Memória

Existem duas formas através das quais um programa em ***C*** pode armazenar dados na memória principal do computador. A primeira consiste em utilizar variáveis globais e locais. No caso das variáveis globais, o espaço de memória necessário está reservado durante todo o tempo de execução do programa. Para variáveis locais é reservado espaço só durante a execução do subprograma onde estão definidas. No entanto este método exige que se conheça *a priori* (na fase de compilação) a quantidade de memória necessária para todas as situações que possam ocorrer.

A segunda forma de armazenar dados consiste em alocar dinamicamente a memória necessária, usando as funções de alocação dinâmica de memória **`malloc()`** e **`free()`**.

A função `malloc()` está definida na biblioteca `<stdlib.h>` como:

```
void *malloc(int número_de_bytes);
```

e reserva um bloco de memória para armazenar o número de *bytes* especificado por `número_de_bytes`, devolvendo o endereço de base da memória alocada (no caso de não existir espaço suficiente devolve `NULL`). Como se pode ver na declaração da função `malloc()`, é devolvido um ponteiro para `void`. Isto significa que devemos usar um `cast` quando atribuímos o ponteiro devolvido por `malloc()` a um ponteiro do tipo pretendido.

Como os tipos de dados não ocupam o mesmo número de *bytes* em todos os sistemas, uma maneira de tornar o nosso código portável consiste em usar a função **`sizeof()`**. Esta função retorna o tamanho, em número de *bytes*, de uma expressão ou tipo. No seguinte programa é dado um exemplo de utilização desta função.

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

No seguinte programa dá-se um exemplo de utilização da função `malloc()`.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
  int *x;

  x =(int *) malloc(sizeof(int)); /* atenção ao cast */
  *x= 10;
  printf("%d",*x);
}
```

Em oposição à função `malloc()`, a função `free()` permite libertar memória previamente reservada usando a função `malloc()`. A função `free()` está definida como:

```
void *free(void *p);
```

O seguinte programa exemplo alocará espaço de memória suficiente para armazenar 40 inteiros, mostra o seu valor e liberta o espaço de memória para ficar novamente disponível no sistema.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
  int *p, x;

  p =(int *) malloc(40*sizeof(int));     /* atenção ao cast */
  if (!p)                          /* verifica se houve espaço */
       printf("Não há memória disponível!\n");
  else
  {
       for(t=0;t<40;++t)     *(p+t)=t;
       for(t=0;t<40;++t)     printf("\n %d",*(p+t));
       free(p);
  }
}
```

Uma função pode devolver um ponteiro para um determinado tipo de dado (inteiro, real, etc.), usando

```
tipo *identificação( parâmetros )
declaração dos parâmetros;
{
  ...

return(...);
}
```

Programa exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int *aloca()
{
  return((int *) malloc(sizeof(int))); /* atenção ao cast */
}

void main()
{
  int *x;

  x = aloca();
  *x= 10;
  printf("%d",*x);
}
```

7 Estruturas

Uma estrutura é um conjunto variáveis, eventualmente de diferentes tipos, agrupadas de modo a poderem ser referenciadas por um identificador comum. As estruturas em ***C*** são o equivalente aos *record*'s em ***Pascal***. As estruturas permitem assim organizar grupos complexos de dados, de grande utilidade principalmente em programas grandes.

7.1 Declaração de Estruturas

Em geral, os elementos de uma estrutura estão relacionados entre si. Por exemplo, pode-se representar a informação de nome e endereço de uma lista de contactos sob esta forma. O seguinte pedaço de código declara uma estrutura com os campos do nome e do endereço

```
struct endereco
  {
       char nome[30];
       char rua[50];
       char localidade[20];
       long int cod_postal;
  };
```

A palavra reservada struct inicia a declaração de uma estrutura, que é constituída por uma lista de declarações entre chavetas. A seguir à palavra `struct` pode ser indicado um identificador da estrutura, tal como no exemplo apresentado atrás. As variáveis incluídas na estrutura são normalmente chamadas elementos da estrutura ou campos da estrutura. De notar que a declaração da estrutura termina com o ponto e vírgula.

A declaração de uma estrutura estabelece apenas a composição da estrutura, i. e., declara um tipo, mas não declara nenhuma variável desse tipo. Para declarar uma variável com este tipo de estrutura pode usar-se

```
struct endereco end_x;
```

Esta linha declara uma variável com estrutura do tipo endereco a que se chamou `end_x`. De forma alternativa, as variáveis podem ser declaradas a seguir à definição da estrutura, a seguir à chaveta e antes do ponto e vírgula. Neste caso poderíamos usar

```
struct endereco
  {
       char nome[30];
       char rua[50];
       char localidade[20];
       long int cod_postal;
  } end_x, end_y;
```

Este pedaço de código declara um tipo estrutura de nome endereco e declara também as variáveis `end_x` e `end_y` desse tipo. No caso de ser apenas necessária uma variável de um tipo de estrutura podemos não incluir o nome da estrutura, apenas o nome da variável, por exemplo:

```
struct
  {
       char nome[30];
       char rua[50];
       char localidade[20];
       long int cod_postal;
  } end_x;
```

A sintaxe genérica da declaração de estruturas é

```
struct nome_tipo_estrutura
  {
       tipo nome_var;
       tipo nome_var;
       ...
  } variáveis_estrutura;
```

em que se pode omitir o nome da estrutura `nome_tipo_estrutura` ou as `variáveis_estrutura`, mas não ambos.

7.2 Referência a Campos de Estruturas

Para referenciar os campos de uma estrutura usa-se o operador **.**, chamado operador ponto. Para atribuir o valor 3510 ao campo `cod_postal` da variável `end_x` usa-se a instrução

```
end_x.cod_postal=3510;
```

e para mostrar no monitor esse mesmo campo usa-se

```
printf("%d",end_x.cod_postal);
```

Para mostrar o campo nome usa-se

```
printf("%s",end_x.nome);
```

e neste caso, como o campo `nome` é na realidade uma *`string`* ou *`array`* de caracteres, podemos aceder a cada um dos seus elementos individualmente. Para aceder ao segundo carácter do campo nome usa-se `end_x.nome[1]`.

7.3 *Array*'s de Estruturas

Uma das utilizações mais comuns das estruturas é em *`array`*'s de estruturas. Para declarar um *`array`* de estruturas é necessário primeiro definir um tipo de estrutura e depois declarar uma variável desse tipo. A sintaxe é

```
struct nome_tipo_estrutura variável[dimensão];
```

Para declarar, por exemplo, um *`array`* com 100 estruturas do tipo `endereco` usamos

```
struct endereco tab[100];
```

Para aceder a uma estrutura específica indexa-se o *array*. Para mostrar no monitor o código postal da terceira estrutura usa-se

```
printf("%d",tab[2].cod_postal);
```

Tal como todas as variáveis do tipo *array*, os índices dos *array*'s de estruturas iniciam-se em zero.

7.4 Passagem de Estruturas para Funções

Nesta secção falaremos de como se podem passar estruturas e os seus elementos para funções.

Quando se passa um elemento de uma variável do tipo estrutura para uma função estamos na realidade a passar o seu valor para a função (passagem de parâmetro por valor), a menos que o elemento não seja uma variável simples (*array* ou *string*). Consideremos como exemplo a seguinte estrutura:

```
struct exemplo
  {
       char x;
       int y;
       float z;
       char s[10];
  } var1;
```

A passagem de cada um dos elementos da variável `var1` para uma função faz-se da forma indicada a seguir:

```
func1(var1.x);    /*  passa o caracter contido em x  */
func2(var1.y);    /*  passa o valor inteiro de y  */
func3(var1.z);    /*  passa o valor real de z  */
func4(var1.s);    /*  passa o endereço da string s  */
func1(var1.s[2]);/*  passa o caracter contido em s[2]  */
```

No entanto, para fazer passagem de parâmetros por referência teremos que usar o operador & para obter o endereço da respectiva variável. Neste caso:

```
func1(&var1.x);         /*  passa o endereço do caracter x  */
func2(&var1.y);         /*  passa o endereço do inteiro y  */
func3(&var1.z);         /*  passa o endereço do real z  */
func4(var1.s);          /*  passa o endereço da string s  */
func1(&var1.s[2]);      /*  passa o endereço do caracter s[2]  */
```

De notar que o operador & precede o nome da estrutura e não o nome do campo. Note-se ainda que o nome do campo *string* é já um endereço, não necessitando do operador &.

Quando pretendemos passar uma estrutura completa para uma função é feita uma passagem por valor. Assim qualquer alteração à estrutura recebida não terá efeitos na estrutura fora da função. É claro que a estrutura tem que ser do mesmo tipo que o parâmetro recebido pela função e o tipo da estrutura tem de ser conhecido tanto fora como dentro da função. Por esta razão é normal fazer a declaração dos tipos estrutura como globais e depois usá-los para declarar variáveis e parâmetros. No seguinte programa é exemplificado o uso de estruturas como parâmetros.

```
#include <stdio.h>
struct tipo_est
  {
       int a,b;
       char ch;
  };

void func(struct tipo_est param)
{
  printf("\n%d",param.a);
 }

void main()
{
  struct tipo_est x;

  x.a=1000;
  func(x);
}
```

Quando se pretende passar para uma função uma estrutura grande é mais eficiente passá-la por referência. Neste caso passa-se o endereço da estrutura, ou seja, passa-se um ponteiro para a estrutura. No exemplo anterior, para declarar um ponteiro para a estrutura `tipo_est` usa-se:

```
struct tipo_est *p;
```

e depois podemos atribuir a este ponteiro o endereço da variável `x` usando a instrução

```
p = &x;
```

`*p` refere-se a toda a estrutura e acede-se aos elementos da estrutura usando `(*p).a`, `(*p).b` e `(*p).ch`. São necessários os parêntesis em `(*p).a` porque o operador ponto tem uma prioridade superior ao operador *. A expressão `*p.a` significa `*(p.a)`, o que é ilícito porque `p.a` não é um ponteiro.

A seguinte função recebe como parâmetro um ponteiro para estrutura, sendo chamada usando `func(&x)`. A atribuição do valor 5 ao campo `a` vai alterar a estrutura fora da função.

```
void func(struct tipo_est *px)
 {
    (*px).a=5;
 }
```

Os ponteiros para estruturas são usados tão frequentemente para passar estruturas para funções que está definida uma forma alternativa de referenciar os campos de uma estrutura dado um ponteiro para essa estrutura. Esta forma alternativa consiste em usar o operador **->** (operador seta) que é exactamente equivalente à forma usada atrás. Assim, a função anterior poderia ser:

```
void func(px)
struct tipo_est *px;
 {
  px->a=5;
 }
```

O operador seta e o operador ponto têm uma prioridade elevada. Na estrutura e ponteiro declarados anteriormente instrução `++px->a` incrementa o campo `a` e não o ponteiro `px` sendo equivalente a `++(px->a)`. Pelo contrário, `(++px)->a` incrementa `px` e acede ao campo `a` e `(px++)->a` acede ao campo `a` da estrutura apontada por `px` e só depois incrementa `px` (neste último caso os parêntesis são desnecessários).

Cada um dos campos de uma estrutura pode ser um elemento de tipo simples ou complexo. Um elemento é de tipo simples se for de um dos tipos de dados previamente definidos na linguagem, tais como inteiros ou caracteres. Tipos complexos são *array*´s e *string*'s, já usados em estruturas definidas em exemplos anteriores, e estruturas. Se for definida a seguinte estrutura

```
struct x
  {
       int a[10][10];   /*  array de 10x10 inteiros  */
       float b;
  } y;
```

para referenciar o elemento 3,4 (quarta linha, quinta coluna) do *array* a usa-se `y.a[3][4]`.

Uma estrutura pode ser o campo de outra estrutura. Por exemplo, se definirmos a estrutura `endereco`:

```
struct endereco
  {
       char rua[50];
       char localidade[20];
       long int cod_postal;
  };
```

podemos depois incluí-la na estrutura `empregado`, definida a seguir:

```
struct empregado
  {
       char nome[30];
       struct endereco ender;
       float ordenado;
  } emp_x;
```

Portanto a estrutura `empregado` tem 3 campos, um dos quais é por si uma estrutura. Para atribuir o valor 3510 ao campo `cod_postal` do campo `ender` da variável `emp_x` usa-se a instrução

```
emp_x.ender.cod_postal=3510;
```

7.5 A palavra reservada typedef

A linguagem ***C*** permite definir novos nomes para os tipos de dados usando a palavra reservada `typedef`. Na realidade não é criado um novo tipo, apenas é definido um novo nome para um tipo já existente. Por exemplo

```
typedef int numero;
```

faz com que o identificador `numero` seja um sinónimo de `int`. Este novo nome pode ser usado em declarações, `cast`'s, etc., exactamente como o tipo `int`. Da mesma forma, a declaração

```
typedef char *string;
```

faz com que `string` seja um sinónimo de `*char`, ou seja, um ponteiro para carácter que poderá ser usado em declarações e `cast`'s

```
string p;
p=(string)malloc(100);
```

Uma utilização mais complexa de `typedef` é em estruturas, como exemplificado no seguinte programa

```
#include <stdio.h>
typedef struct pessoa
 {
  char nome[80];
  char bi[12];
 } Pessoa,*pes;

void leitura()
 {
  pes id;

  id =(pes) malloc(sizeof(pessoa));
  scanf("%s",id->nome);
  printf("\n\n%s\n",id->nome);
 }

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

8 Ficheiros

Até agora considerámos que todas as entradas e saídas de dados eram feitas da consola, i. e., a entrada de dados era feita a partir do teclado e a saída era feita para o monitor. Para aumentar as funcionalidades dos programas desenvolvidos é útil poder ler e escrever dados em ficheiros.

Antes de um programa poder ler um ficheiro é necessário que esse ficheiro seja aberto. Para tal é usada a função **fopen** que devolve um ponteiro para o ficheiro. Este ponteiro, chamado ponteiro de ficheiro, aponta para uma estrutura que contém informação relativa ao ficheiro, tal como o seu nome, a posição actual, se foi aberto para escrita ou leitura, etc., sendo depois usado quando se acede ao ficheiro, tanto para leituras como para escritas. Um ponteiro para ficheiro é uma variável que tem de ser declarada previamente. Para declarar `fp` como um ponteiro para ficheiro usa-se

```
FILE *fp;
```

em que `FILE` é um tipo já definido na linguagem ***C***, tal como `int` e `char`.

A função `fopen` abre um ficheiro e associa-o a um ponteiro para ficheiro, tendo o seguinte protótipo

```
FILE *fopen(char *nome_ficheiro, char *modo);
```

O primeiro argumento da função é uma *string* com o nome do ficheiro a abrir e o segundo é uma *string* com o modo de abertura do ficheiro. Alguns dos modos de abertura de um ficheiro no Turbo C são apresentados na tabela a seguir:

Modo	significado
"r"	abre para leitura
"w"	abre para escrita
"a"	abre para adicionar
"r+"	abre para leitura e escrita (posicão no início)
"w+"	abre para leitura e escrita
"a+"	abre para leitura e escrita (posicão no fim)

Para abrir um ficheiro teste.dat para escrita podemos usar:

```
fp = fopen("teste.dat","w");
```

em que `fp` é uma variável do tipo `FILE *`. Dado que a função pode não conseguir abrir um ficheiro, devolvendo nesse caso `NULL`, é mais usual usar o código

```
if ((fp = fopen("teste.dat","w"))==NULL)
  {
       printf("\n Não é possível abrir o ficheiro!");
       exit(1);
  }
```

que permite detectar erros na abertura de ficheiros, tais como tentar escrever um ficheiro num disco protegido para escrita ou tentar criar um ficheiro num disco cheio.

Quando se usa a função `fopen()` para abrir um ficheiro para escrita com **"w"** será criado um novo ficheiro com o nome especificado. Se já existir um ficheiro com esse nome é apagado e criado um novo. Se pretendemos adicionar informação a um ficheiro então deveremos abri-lo com o modo **"a"** para que a informação que ele já contém não seja perdida. Se for usado o modo **"a"** para abrir um ficheiro que não exista a função `fopen()` devolverá um erro. De forma análoga, a abertura de ficheiros para operações de leitura requer que o ficheiro exista, caso contrário será devolvido um erro. Se um ficheiro for aberto para operações de leitura/escrita não será apagado o seu conteúdo se ele já existir. No entanto, se o ficheiro não existir será criado.

Após a abertura de um ficheiro podemos então escrever ou ler a sua informação. Para escrever um carácter num ficheiro previamente aberto podemos usar a função `putc()`. O protótipo desta função é

```
int putc(int ch, FILE *fp);
```

em que `fp` é o ponteiro de ficheiro devolvido por `fopen()` e `ch` é o carácter a escrever no ficheiro. Por razões históricas `ch` está definido como inteiro, mas apenas é escrito o primeiro *byte* no ficheiro.

Se a função `putc()` for bem sucedida devolverá o carácter escrito, se for mal sucedida devolve `EOF`. `EOF` é um carácter especial definido na linguagem ***C*** e que significa fim de ficheiro.

A função `getc()` permite ler caracteres num ficheiro previamente aberto com `fopen()` no modo leitura. A função está declarada como

```
int getc(FILE *fp);
```

Também por razões históricas, `getc()` devolve um inteiro mas o *byte* de maior ordem é zero.

A função `getc()` devolve `EOF` quando o fim do ficheiro é atingido. Assim, para ler um ficheiro até ao fim poderemos usar o código

```
ch = getc(fp);
while (ch!=EOF)
  {
       ch = getc(fp);
  }
```

No entanto, se o ficheiro for binário é possível que o valor lido do ficheiro tenha um valor igual ao `EOF`. Se isto acontecer o código anterior não lê o ficheiro até ao fim. Para evitar este problema o *Turbo C* inclui a função `feof()` que testa quando se atinge o fim do ficheiro. A função `feof()` devolve o valor 1 quando é atingido o fim do ficheiro. Para ler um ficheiro binário poderemos então usar

```
while (!feof(fp))  ch = getc(fp);
```

Este método funciona tanto com ficheiros de texto como com ficheiros binários.

Após o acesso a um ficheiro aberto usando a função `fopen()` este tem de ser fechado usando a função `fclose()`. A função `fclose()` permite terminar a associação entre um ponteiro de ficheiro e o nome externo do ficheiro estabelecida por `fopen()`, e escreve para o ficheiro os dados que eventualmente ainda estejam no *buffer* e fecha o ficheiro ao nível do sistema operativo.

A função `fclose()` está declarada como

```
int fclose(FILE *fp);
```

em que `fp` é o ponteiro devolvido por `fopen()`. Se o valor devolvido por `fclose()` for zero a operação de fecho foi concluída com sucesso.

Estão definidas duas funções para ler e escrever *string's* para ficheiros: `fgets()` e `fputs()`. Estas funções estão declaradas como:

```
char *fgets(char *s, int n, FILE *fp);
int fputs(const char *s, FILE *fp);
```

A função `fgets()` permite ler a próxima *string* (incluindo o carácter nova linha) a partir do ficheiro `fp` para a *string* apontada por `s`. No máximo serão lidos `n-1` caracteres. A *string* resultante será terminada pelo carácter nulo '`0`'.

A função `fputs()` escreve uma *string* (que não pode conter um carácter nova linha) para um ficheiro. Esta função devolve `EOF` se ocorrer um erro e zero no caso contrário.

A escrita e leitura de dados em ficheiros pode ser feita usando as funções `fprintf()` e `fscanf()`, respectivamente, que têm a seguinte declaração

```
int fprintf( FILE *fp, const char *format[, arg1, arg2, ...]);
int fscanf( FILE *fp, const char *format[, arg1,arg2, ...]);
```

O modo de utilização destas funções semelhante às congéneres `printf()` e `scanf()` só que enquanto estas escrevem e lêem na saída e entrada padrão (monitor e teclado, respectivamente), as funções `fprintf()` e `fscanf()` permitem escrever e ler do ficheiro associado ao ponteiro `fp`, dado como primeiro argumento destas funções.

Para ler e escrever blocos de dados estão ainda definidas as funções `fread()` e `fwrite()` que têm a seguinte declaração

```
int fread(void *ptr, int size, int num, FILE *fp);
int fwrite(void *ptr, int size, int num, FILE*fp);
```

No caso de `fread()`, ptr é um ponteiro para a região de memória que receberá os dados lidos do ficheiro. Em `fwrite()`, `ptr` é um ponteiro para a informação a escrever no ficheiro. Em ambos os casos `size` é o tamanho em número de *bytes* de cada elemento a escrever ou ler do ficheiro e `num` é o número de elementos a escrever ou ler. `fp` é o ponteiro associado ao ficheiro a que se quer aceder.

Até aqui, o acesso a ficheiros tem sido feito de forma sequencial, i. e., acede-se a um elemento (leitura ou escrita) e o ponteiro de ficheiro fica a apontar para o elemento imediatamente a seguir. Quando se quer aceder aos elementos de um ficheiro de forma não sequencial temos de reposicionar o apontador de ficheiro para a posição pretendida usando a função `fseek()`, que está declarada da seguinte forma:

```
int fseek(FILE *fp, long int num_bytes, int origem);
```

em que `fp` é o ponteiro de ficheiro devolvido por uma chamada à função `fopen()`. `fseek()` coloca o apontador de ficheiro na posição `num_bytes` a partir da origem. Esta origem pode ter o valor `0`, `1` ou `2` consoante a posição pretendida, estando definidas constantes para cada um dos valores.

Valor	Origem	Constante
0	início do ficheiro	SEEK_SET
1	posição actual do ficheiro	SEEK_CUR
2	fim do ficheiro	SEEK_END

Se o valor devolvido pela função `fseek()` for zero significa que o posicionamento do apontador foi executado com sucesso, no caso de ocorrer um erro é devolvido um valor não nulo. O uso de `fseek()` não é muito utilizado com ficheiros de texto sendo mais adequado para ficheiros binários. Por exemplo, para ler o 234º *byte* num ficheiro de nome `teste.dat` podemos usar o seguinte código

```
if ( (fp=fopen(“teste.dat”,“rb”)) == NULL)
  {
      printf(“\n Impossível abrir ficheiro!”);
  }
else   fseek(fp,234L,0);
```

De notar o uso do modificador `L` à constante 234 para forçar o compilador a tratá-la como `long int`. Poderia também usar-se um cast.

Para num determinado momento saber a posição do apontador de ficheiro está definida a função `ftell()`

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

No caso de ocorrer erro será devolvido `-1L`.

No programa apresentado a seguir ilustra-se a escrita e leitura de estruturas em ficheiros.

```
#include <stdio.h>
typedef struct pes
{
  char nome[80];
  float alt;
} pessoa;
void main()
{
  FILE *fp;
  pessoa p,r;

  fp = fopen("texto.txt","w");
  strcpy(p.nome,"jose manuel");
  p.alt=1.70;

  fwrite(&p,sizeof(p),1,fp);
  fclose(fp);

  fp = fopen("texto.txt","r+");
  fread(&r,sizeof(p),1,fp);
  fclose(fp);
  printf("\nNome --> %s\nAltura --> %f\n",r.nome,r.alt);
}
```