

# Programação em linguagem C

## Um curto resumo

Arquitectura de Computadores

9 de Outubro de 2011



## **Agradecimentos**

Ao Professor Doutor Miguel Pimenta Monteiro da FEUP, pela permissão concedida para reutilizar o seu tutorial sobre a linguagem C de 1996, do qual estes apontamentos são uma reedição.

# Índice

<b>1. Exemplos simples</b>	<b>6</b>
1.1. Um programa simples . . . . .	6
1.2. Compilação . . . . .	6
<b>2. Programação simples em C</b>	<b>7</b>
2.1. Variáveis . . . . .	7
2.2. Variáveis globais . . . . .	8
2.3. Funções de entrada/saída . . . . .	8
2.4. Operações aritméticas . . . . .	9
2.5. Operadores de comparação . . . . .	9
2.6. Operadores lógicos . . . . .	10
2.7. Operadores binários . . . . .	10
2.8. Precedência dos operadores . . . . .	10
<b>3. Controle de fluxo</b>	<b>12</b>
3.1. A instrução if . . . . .	12
3.2. A instrução switch . . . . .	13
<b>4. Repetições e ciclos</b>	<b>14</b>
4.1. A instrução while . . . . .	14
4.2. A instrução do-while . . . . .	15
4.3. Ciclos for . . . . .	15
4.4. As instruções break e continue . . . . .	16
4.5. Vetores unidimensionais e multidimensionais . . . . .	17
4.6. Strings . . . . .	17
<b>5. Funções</b>	<b>19</b>
5.1. Definição de funções . . . . .	19
5.2. Funções void . . . . .	20
<b>6. Outros Tipos de Dados</b>	<b>21</b>
6.1. Estruturas . . . . .	21
6.2. Uso de typedef . . . . .	21
6.3. Uniões . . . . .	22
6.4. Conversão entre tipos . . . . .	23
6.5. Variáveis estáticas . . . . .	24
<b>7. Apontadores</b>	<b>26</b>
7.1. O que são apontadores? . . . . .	26
7.2. Apontadores e funções . . . . .	29
7.3. Apontadores e vetores . . . . .	30
7.4. Vetores de apontadores . . . . .	32
7.5. Apontadores e estruturas . . . . .	32
<b>A. Exemplos e exercícios de programação</b>	<b>35</b>
A.1. Programas simples . . . . .	35

A.2. Ciclos e decisões . . . . .	37
A.3. Tipos de dados e atribuições . . . . .	42
A.4. Funções . . . . .	46
A.5. Macros e afins . . . . .	50
A.6. Caracteres e strings . . . . .	52
A.7. Apontadores . . . . .	55
A.8. Operações de entrada/saída . . . . .	57
A.9. Acesso a ficheiros . . . . .	61
A.10. Estruturas, uniões e campos de bits . . . . .	64
A.11. Memória dinâmica . . . . .	70
A.12. Operações diversas . . . . .	73
A.13. Erros mais comuns . . . . .	76

## 1. Exemplos simples

Vamos começar por ver programas simples para demonstrar a estrutura de um programa em C, e exemplificar como se pode compilar e correr um programa.

### 1.1. Um programa simples

O seguinte programa escreve algum texto no écran, e deve ser o programa mais divulgado:

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello world!\n");
5
6     return(0);
7 }
```

Veremos o que faz mais tarde o `printf` que é uma função existente na biblioteca standard da linguagem, ou então o leitor pode adivinhar o que a função faz.

### 1.2. Compilação

A compilação dos programas em C faz-se através da invocação de um compilador (p. ex. no UNIX, o comando `gcc`). O comando de compilação deverá ser seguido pelo nome do ficheiro que contém o código fonte (geralmente com a extensão `.c`). É também comum colocar como parâmetros de chamada do compilador, várias opções de compilação (é aconselhável como medida de prevenção a opção de ligar (quase) todos os avisos possíveis do compilador de C).

Assim uma compilação básica onde `prog1.c` seria o nome do ficheiro contendo o código fonte, pode ser executada no UNIX através do seguinte comando:

```
gcc -Wall prog1.c
```



O uso da opção `-Wall` é fortemente aconselhado como um procedimento saudável. Da mesma forma recomenda-se também uma atenção cuidadosa a todos os avisos do compilador! Só porque são avisos e não erros não se devem ignorar!

Se existirem erros de sintaxe no código fonte, o compilador detecta-os e indica a sua localização junto com uma breve descrição do erro. Erros na lógica do programa apenas poderão ser detectados durante a execução do mesmo. Se o programa não contiver erros de sintaxe o compilador produzirá código executável. Tratando-se de um programa completo o código executável é colocado, por defeito, num ficheiro chamado `a.out` (isto no UNIX). Se quisermos colocar o resultado da compilação noutra ficheiro deverá utilizar-se a opção `-o`:

```
gcc -o program -Wall prog1.c
```

Neste caso o código executável é colocado no ficheiro `program` que é já criado com as permissões de execução adequadas (no UNIX)<sup>1</sup>.

<sup>1</sup>Para correr o programa fazer `./a.out` ou `./program` na linha de comandos!

## 2. Programação simples em C

Vamos agora ver aspectos elementares dos programas em C, tais como a sua estrutura, os tipos de dados existentes, como se faz a declaração de variáveis, e como se usam os operadores. Assume-se a familiarização prévia com uma linguagem de programação de alto nível.

### 2.1. Variáveis

O C tem pré-definidos os seguintes tipos de dados simples:

Tabela 2.1.: Exemplos de tamanhos dos tipos dados em C

Tipo	Bytes	Bits	Valores de limite	
short int	2	16	-32,768	32,767
unsigned short int	2	16	0	65,535
unsigned int	4	32	0	4,294,967,295
int	4	32	-2,147,483,648	2,147,483,647
long int	4	32	-2,147,483,648	2,147,483,647
signed char	1	8	-128	127
unsigned char	1	8	0	+255
float	4	32		
double	8	64		
long double	12	96		

Geralmente nos sistemas UNIX os tipos `int` e `long int` são equivalentes (inteiros de 32 bits). No entanto noutros sistemas é possível que o tipo `int` seja equivalente a um `short int` (inteiro de 16 bits). É necessário consultar a documentação do compilador para o sistema em questão. O prefixo `unsigned` pode também ser usado com os tipos `int` e `long int`. O C não tem um tipo booleano pré-definido, no entanto, poderá usar-se um `char` ou um `int` para o efeito.

⚠ Uma das vantagens do C é possibilitar a mistura entre inteiros e variáveis lógicas que permite muitas simplificações e optimizações na escrita de código, relativamente a outras linguagens. No entanto esses *truques* devem estar bem assinalados e comentados de forma a que o programa se mantenha legível. Para sabermos em concreto o tamanho dos tipos de dados podemos usar `sizeof()` ou consultar os ficheiros `limits.h` e `float.h`.

É usada a seguinte regra para declararmos variáveis em C de um determinado tipo:

```
tipo_das_variáveis lista_de_variáveis;
```

Por exemplo:

```
1 int i, j, k;
2 float x, y, z;
3 char ch;
```

## 2.2. Variáveis globais

As variáveis globais, visíveis em todas as funções de um programa, declaram-se fora e antes de todas as funções (só são visíveis a partir do local da declaração). Podemos ver um exemplo disso no código seguinte.

```
1 short numero, soma;
2 int numgrande, somagrande;
3 char letra;
4
5 int main(void)
6 {
7
8 return(0);
9 }
```

É também possível inicializar as variáveis globais no momento da declaração. Usa-se para isso o operador de atribuição =. Por exemplo:

```
1 float soma = 0.0;
2 int produto = 0;
3 char ch = 'A';
4
5 int main(void)
6 {
7 return(0);
8 }
```

O C possibilita colocar o mesmo valor em várias variáveis ao mesmo tempo, usando múltiplas atribuições :

```
1 x = y = z = t = 3;
```

Para a definição de novos tipos em C usamos um typedef, da seguinte forma:

```
typedef tipo_já_definido novo_tipo;
```

Concretizando:

```
1 typedef char letra;
2 typedef int num;
3
4
5 num x = 0;           /* equivalente a int */
6 letra ch = 'A';      /* equivalente a char */
```

## 2.3. Funções de entrada/saída

As funções `printf()` e `scanf()` (existentes na biblioteca standard de Input/Output `stdio.h`) permitem escrever no écran e ler do teclado, respectivamente, o valor de variáveis. Estas funções têm como primeiro parâmetro uma string especificando o formato e a ordem das variáveis



a escrever ou a ler. Seguem-se como parâmetros as próprias variáveis pela ordem especificada. Na string de formatação indica-se o local e o tipo de um valor de variável através do carácter % seguido de uma letra indicadora do tipo. Alguns dos tipos suportados são:

%c – char    %d – int's    %f – float's    %s – string's    %p – pointer's

Um exemplo:

```
1 printf("Os valores das três variáveis são: %c, %d, %f\n", ch, i, x);
```

A função `scanf()` lê valores do teclado para variáveis. A sua estrutura é semelhante a `printf()`. Por exemplo:

```
1 scanf("%c %d %f", &ch, &i, &x);
```



Na lista de variáveis passada à função `scanf()` é obrigatório usar o operador `&` antes de cada variável; veremos mais tarde porquê. Outra característica saliente destas funções (comum ao `scanf()` e ao `printf()`) é o facto de possuírem um número variável de argumentos. Algo que é possível e legal em C, mas impossível noutras linguagens de programação.

## 2.4. Operações aritméticas

As quatro operações aritméticas standard são representadas em C pelos símbolos habituais (+, -, \*, /), da mesma forma que na maioria das linguagens de programação. Além destas operações aritméticas são suportadas outras.

A atribuição é representada no C pelo operador `=`.

Exemplos:

```
1 resposta = 's'; num = 4;
```

Um outro operador aritmético do C é o operador módulo `%`. Este operador tem como resultado o resto da divisão inteira. Só pode ser utilizado com valores inteiros como é óbvio. O operador de divisão `/`, pode ser usado com inteiros e reais.



A divisão inteira em C tem uma descontinuidade à volta do zero. Isto é: fazendo a correspondência entre o resultado da operação como um número real e o número inteiro que é produzido pela operação, este será zero no intervalo  $] - 1.0, 1.0[$ .

## 2.5. Operadores de comparação

O operador de teste de igualdade no C é `==`, à semelhança do que se passa na linguagem Java.



Note-se que é muito fácil trocar o teste de igualdade pelo operador de atribuição, o que conduz invariavelmente a erros difíceis de detectar (só um `=` em vez de `==`). A seguinte instrução condicional está sintacticamente correcta: `if (x = y) ...`, no entanto o que faz é copiar o valor de `y` para `x` e tem como resultado o valor de `y`, que é interpretado como `TRUE` se for diferente de 0 e `FALSE` no caso contrário. Se calhar o programador pretendia comparar `x` com `y` com resultado `TRUE` se fossem iguais. Esta é uma das razões pelas quais se deve prestar atenção a todos os avisos que o compilador dá. O C permite fazer muitos *truques* de programação, mas convém ter a certeza que era isso que o programador pretendia.

Outro erro igualmente perigoso é escrever `x==y`; em vez de `x=y`; . Neste caso a variável `x` não recebe nenhum valor ficando o valor que tinha inicialmente.

O operador de teste de desigualdade é em C `!=`. Os outros quatro operadores de comparação são: `<`, `<=`, `>` e `>=`.

Tabela 2.2.: Comparações

Condição	Operador
Igualdade	<code>==</code>
Desigualdade	<code>!=</code>
Maior	<code>&gt;</code>
Menor	<code>&lt;</code>
Maior ou igual	<code>&gt;=</code>
Menor ou igual	<code>&lt;=</code>

## 2.6. Operadores lógicos

Os operadores lógicos são utilizados para combinar resultados de comparações (valores lógicos) e são geralmente utilizados nas instruções condicionais. Os três operadores lógicos do C são:

Tabela 2.3.: Operadores lógicos

Operação	Operador
not	<code>!</code>
and	<code>&amp;&amp;</code>
or	<code>  </code>



Os operadores `and` e `or` funcionam em *curto circuito*. Isto é, uma expressão é avaliada termo a termo, da esquerda para a direita, e a partir do termo que é possível prever o resultado final de uma expressão, abandona-se a avaliação dos termos seguintes. Um valor lógico falso corresponde ao inteiro zero e o valor lógico verdadeiro a um, mas qualquer inteiro diferente de zero será interpretado como sendo verdadeiro.

## 2.7. Operadores binários

Os operadores binários efectuam operações bit a bit entre os termos. Não se devem confundir os operadores binários com os operadores lógicos, porque os programas podem até funcionar, mas só para alguns casos especiais!

## 2.8. Precedência dos operadores

A precedência dos operadores em C pode induzir o programador em erro, porque por exemplo, as operações lógicas possuem uma precedência menor do que as comparações. Assim, em caso de dúvida, e para tornar os programas mais claros, devemos usar parênteses, sempre que haja dúvidas.

Tabela 2.4.: Operadores binários

Operação	Operador
not	~
and	&
or	
xor	^

Tabela 2.5.: Precedência dos operadores

Operadores	Descrição	Associatividade
	Parênteses	Esquerda para a direita
[]	Índice de um vector	Esquerda para a direita
.	Seleccção de um membro de uma estrutura	Esquerda para a direita
->	Seleccção de um membro de uma estrutura usando um apontador	Esquerda para a direita
+ -	Mais e menos unários (sinal)	Direita para a esquerda
++ -	Prefixos de incremento e decremento	Direita para a esquerda
!	Não lógico e complemento bit a bit	Direita para a esquerda
*	Apontado por	Direita para a esquerda
&	Endereço de	Direita para a esquerda
sizeof	Tamanho de uma expressão ou tipo	Direita para a esquerda
(type)	Conversão explícita de tipos	Direita para a esquerda
* / %	Multiplicação divisão e módulo (resto da divisão)	Esquerda para a direita
+ -	Adição e subtracção	Esquerda para a direita
<< >>	Deslocamento bit a bit para a esquerda e para a direita	Esquerda para a direita
< <=	Menor e menor ou igual	Esquerda para a direita
> >=	Maior e maior ou igual	Esquerda para a direita
== !=	Igual e diferente	Esquerda para a direita
&	AND bit a bit	Esquerda para a direita
^	XOR bit a bit	Esquerda para a direita
	OR bit a bit	Esquerda para a direita
&&	AND lógico	Esquerda para a direita
	OR lógico	Esquerda para a direita
?:	Operador condicional	Direita para a esquerda
=	Atribuição	Direita para a esquerda
+= -=	Atribuição com soma e atribuição com subtracção	Direita para a esquerda
/= *=	Atribuição com divisão e atribuição com multiplicação	Direita para a esquerda
%=	Atribuição com módulo	Direita para a esquerda
<<= >>=	Atribuição com deslocamento para a esquerda e para a direita	Direita para a esquerda
&=    =	Atribuição com AND bit a bit e atribuição com OR bit a bit	Direita para a esquerda
^=	Atribuição com XOR bit a bit	Direita para a esquerda
,	Vírgula	Esquerda para a direita

## 3. Controle de fluxo

Através das instruções de controle de fluxo conseguimos transformar as expressões lógicas e condições em decisões, porque com as instruções de controle de fluxo conseguimos fazer testes que afectam o fluxo de execução de um programa

### 3.1. A instrução if

A instrução if em C tem um estrutura similar à existente em muitas outras linguagens de programação, sendo o else opcional:

```
if (expressão)
    instrução;
```

ou

```
if (expressão)
    instrução_1;
else
    instrução_2;
```

Como se vê no exemplo seguinte podemos usar vários ifs uns a seguir aos outros:

```
if (expressão)
    instrução_1;
else if (expressão)
    instrução_2;
else
    instrução_3;
```

Outro exemplo:

```
1 int main(void)
2 {
3     int a, b, max;
4
5     ...
6     if (a >= b) {
7         max = a;
8         ...
9     }
10    else {
11        max = a;
12        ...
13    }
14    ...
15 }
```

Note-se a colocação das chavetas.

### 3.2. A instrução switch

Em vez do nome `case` vulgar noutras linguagens de programação, a instrução similar em C recebeu o nome de `switch`, e tal como a anterior, esta permite a partir do cálculo de uma expressão, a escolha de um caminho a seguir na execução de um programa. A sua sintaxe normal é a seguinte:

```
switch (expressão) {
    case valor_1:
        instrução_1;
        break;
    case valor_2:
        instrução_2;
        break;
    ...
    case valor_n:
        instrução_n;
        break;
    default:
        instrução;
}
```

Os valores que aparecem a seguir à palavra `case` não podem ser expressões nem variáveis. São obrigatoriamente constantes. Se o cálculo da expressão inicial resultar num desses valores executa-se a instrução correspondente. Se o cálculo da expressão não tiver como resultado nenhum dos valores existentes, é executada a instrução correspondente ao `default`. Se não existir nenhum `default` o fluxo de execução continua na instrução seguinte ao instrução `switch`.



Num `switch` a execução de uma instrução (se um valor *bater certo* com o resultado da expressão) provoca não só o executar de um dos ramos do `case` mas também a execução das instruções correspondentes aos valores seguintes. Este comportamento não é o normal em outras linguagens de programa, e pode causar problemas se o programador não estiver atento. Para um comportamento mais *normal* devemos colocar um `break` em cada um dos ramos do `switch`.

Isso pode ser visto no seguinte exemplo:

```
1 resultado=0;
2 switch (letra) {
3     case 'A':
4         resultado=resultado+10;
5     case 'B':
6         resultado=resultado+10;
7     case 'C':
8         resultado=resultado+10;
9     case 'D':
10        resultado=resultado+10;
11    }
12 }
```

Neste caso se a variável `letra` for `'A'`, `'B'`, `'C'`, ou `'D'` a variável `resultado` terá respectivamente os valores 40, 30, 20 ou 10.

## 4. Repetições e ciclos

Neste capítulo vamos ver as várias instruções do C que podemos usar para fazer ciclos e repetir condicionalmente a execução de blocos de código.

### 4.1. A instrução `while`

A instrução `while` compreende-se ao traduzir o `while` para enquanto. Trata-se de enquanto uma determinada condição for verdadeira, repetir a execução de um determinado bloco de código. A sua sintaxe é a seguinte:

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

Um exemplo:

```
1 #include <stdio.h>
2
3 int y=20,x=3,quociente=0;
4
5 int main()
6 {
7     while (y > x) {
8         y=y-x;
9         quociente++;
10    }
11    printf(" %d \n",quociente);
12    return(0);
13 }
```

O exemplo anterior escreve no écran o resultado da divisão de 20 por 3.



Um dos sítios onde é usado normalmente o *truque* de usar inteiros como valores lógicos é a condição de um `while`. Assim poderemos ter também as instruções que o `while` vai repetir dentro da própria condição.

Exemplos:

```
1 while (x--);
```

```
1 while ((tecla = getchar()) != '0')
2     putchar(tecla);
```

Nos exemplos anteriores temos um ciclo que decreenta `x` até que este seja zero e outro ciclo que vai copiando caracteres da entrada standard para a saída até que apareça o carácter zero.

## 4.2. A instrução do-while

Enquanto um `while` testa primeiro e só executa (condicionalmente) depois, a instrução `do...while` executa primeiro e testa depois. Desta forma o ciclo é executado pelo menos uma vez, enquanto que no `while` o ciclo pode não ser executado nunca.

A sua sintaxe é:

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

Um exemplo:

```
1
2 int y = 20,x = 3;quociente=0;
3
4 do
5     { y=y-x;
6       quociente++;
7     };
8 while (y > x);
9 printf("quociente = %d\n", quociente);
10
```

O exemplo anterior faz na mesma a divisão de dois números através da repetição de subtrações, mas desta vez faz pelo menos uma subtração<sup>1</sup>. Um dos locais onde se usa normalmente um ciclo `do...while` é na entrada de dados. Isto porque é aconselhável verificar se os dados de entrada são válidos, mas isto só se pode fazer depois de os termos lido. Assim a entrada de dados num programa robusto deve estar dentro de um ciclo do estilo,

```
do
    ler_dados ;
while ( dados_estão_inválidos )
```

## 4.3. Ciclos for

A sintaxe da instrução `for` é diferente do habitual para outras linguagens. Uma instrução `for` é então definida como:

```
for (expressão1; expressão2; expressão3)
    instrução;
```

A expressão1 é o inicializador, a expressão2 constitui o teste de terminação, e a expressão3 é o modificador (é executada em cada ciclo e pode fazer mais do que um simples incremento ou decremento).

A instrução `for` é equivalente a um ciclo `while` com a seguinte construção:

```
expressão1;
while (expressão2) {
    instrução;
    expressão3;
}
```

---

<sup>1</sup>O que poder ser errado e/ou perigoso.

Por exemplo, o seguinte código

```
1 int x;
2 void main(void)
3 {
4     for (x=3; x>0; x--)
5         printf("x = %d\n", x);
6 }
```

escreve no écran estas mensagens:

```
x = 3
x = 2
x = 1
```

As próximas instruções for são todas legais:

```
1 for (x = 0; x <= 9 && x != 3; x = x + 1);
2
3 for (x = 0, y = 4; x <= 3 && y < 9; x = x + 1, y = y + 2);
4
5 for (x = 0, y = 4, z = 1000; z; z /= 10);
```

Repare-se no uso do operador `,` que serve para executar múltiplas acções, sendo o resultado da última acção o resultado da expressão. No terceiro ciclo, a execução continua até que `z` se torne 0 (FALSE).

#### 4.4. As instruções break e continue

As instruções `break` e `continue` permitem-nos controlar melhor a forma de execução dos ciclos:

- `break` – termina imediatamente a execução de um ciclo ou da instrução `switch`
- `continue` – salta imediatamente para a avaliação da expressão de controlo do ciclo

No exemplo seguinte pretende-se ler uma série de inteiros do teclado e fazer qualquer coisa com eles. No entanto se o valor lido for 0 terminamos a leitura, se o valor for negativo escrevemos uma mensagem de erro e terminamos o ciclo, e se for maior do que 100 ignoramos esse valor e passamos à leitura do seguinte.

```
1 while (scanf("%d", &value) && value != 0) {
2     if (value < 0) {
3         printf("Valor ilegal\n");
4         break;
5     }
6     if (value > 100)
7         continue;
8     ...;
9     /* processar value */
10    ...;
11 }
```

A função `scanf()` lê valores do teclado e coloca-os em variáveis. Retorna o número de caracteres lidos se tiver sucesso e 0 no caso contrário. Assim, se houver uma leitura, ela retorna um valor diferente de 0, que é interpretado como TRUE.



## 4.5. Vectors unidimensionais e multidimensionais

Os vectores declaram-se no C como se declaram as outras variáveis, acrescentando apenas um valor para a sua dimensão. Assim começa-se por indicar o tipo de valores que vão estar contidos em cada posição do vector, seguindo-se o nome do próprio, e terminando com a indicação da dimensão (nº de elementos) entre parêntesis rectos. Por exemplo, para definir um vector de 50 inteiros poderia usar-se a seguinte declaração:

```
1 int numbers[50];
```



Em C os vectores começam sempre no índice 0 e vão até ao valor da dimensão menos 1. Na definição de cima os índices válidos do vector `numbers` vão desde 0 até 49.

O acesso aos elementos individuais do vector faz-se utilizando também parêntesis rectos. Exemplos:

```
1 terceiro_valor = numbers[2];
2 numbers[5] = 100;
```

Os vectores multidimensionais declaram-se, indicando as várias dimensões umas a seguir às outras, da mesma forma que a primeira dimensão. Exemplo, para duas dimensões:

```
1 float matriz[50][50];
```

Para maiores dimensões basta acrescentar à declaração mais elementos [...]:

```
1 double big_array[10][10][22]...[8];
```

Os acessos aos elementos individuais faz-se da mesma forma:

```
1 valor = matriz[10][4];
2 matriz[0][1] = 100;
```

Nota: Em C os vectores são armazenados na memória por forma a que dois elementos consecutivos (na memória) correspondem prioritariamente a uma variação do último índice. Para matrizes bidimensionais, isso corresponde a um armazenamento linha a linha.

## 4.6. Strings

Na linguagem C as strings são simplesmente vectores de caracteres. No entanto a linguagem propriamente dita não tem quaisquer facilidades para o manuseamento de strings (excepto para a primeira inicialização). Apenas na biblioteca standard se encontra uma vasta gama de funções de manuseamento de strings<sup>2</sup>.

Uma declaração de uma string capaz de conter 50 caracteres:

```
1 char nome[51];
```

Como o C não manuseia strings directamente todas as seguintes instruções de atribuição são ilegais:

<sup>2</sup>Ver <string.h>.

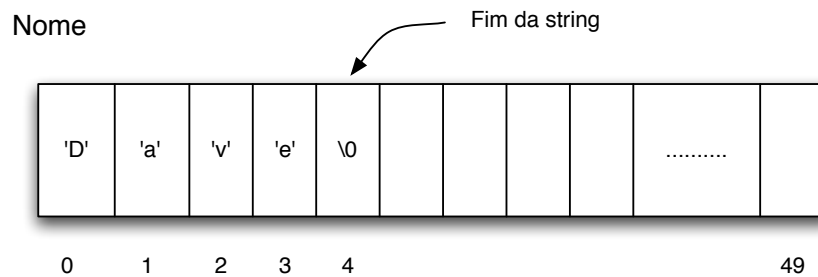
```
1 char nome[50], apelido[50], nome_completo[100];
2 nome = "Arnold";                               /* Illegal */
3 apelido = "Schwarznegger";                       /* Illegal */
4 nome_completo = "Mr. " + nome + ' ' + apelido;   /* Illegal */
```

No entanto a seguinte declaração com inicialização é válida:

```
1 char nome[50] = "Dave";
```

Para permitir a existência de strings de tamanho variável, quer a inicialização anterior, quer todas as funções de strings da biblioteca standard acrescentam um carácter final a todos os strings. Esse carácter (que marca o fim de um string) tem o valor 0 (código 0, não o carácter '0'). Se quisermos representar esse carácter podemos escrever '\0'.

A inicialização anterior produz a seguinte imagem na memória:



Para imprimir um string no écran pode usar-se a função `printf()`, com um especificador, no 1º parâmetro igual a `%s`, no local da variável do tipo string (`char []`):

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

## 5. Funções

Um programa em linguagem C é, como já se disse, essencialmente uma colecção de funções. Essas funções são muito semelhantes às que são possíveis de definir noutras linguagens como o Pascal. O próprio programa principal em C é uma função — a função `main()`. No C não existe o conceito de procedimento, como existia no Pascal. Os procedimentos em C são também funções, com a particularidade de não retornarem coisa alguma.

### 5.1. Definição de funções

A sintaxe geral para a definição de uma função é a seguinte:

```
tipo_de_retorno nome_da_função (def_parâmetro, def_parâmetro, ...)
{
    variáveis_locais

    instruções
}
```

Por exemplo, uma função para calcular o valor médio de dois números, poderia ser definida da seguinte forma:

```
1 float average(float a, float b)
2 {
3     float ave;
4
5     ave = (a + b) / 2;
6
7     return (ave);
8 }
```

Esta função poderia depois ser chamada da função `main()` como se mostra no exemplo seguinte:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     float a=5, b=15, result;
5
6     result = average(a, b);
7     printf("Média = %f\n", (double)result);
8     return (0);
9 }
```

Repare-se na instrução de `return` na função, que além de a terminar é também a responsável pela definição do valor de retorno da mesma.

## 5.2. Funções void

As funções void são funções que não retornam qualquer valor. Estas funções são definidas tendo como tipo de retorno a palavra void.

A palavra void também deverá ser usada no lugar dos parâmetros se a função não tiver nenhum. Um exemplo:

```
1 #include <stdio.h>
2 void squares(void)
3 {
4     int k;
5
6     for (k=1; k<=10; k++)
7         printf("%d\n", k*k);
8 }
9
10 int main(void)
11 {
12     squares();
13     return(0);
14 }
```

Na chamada de funções sem argumentos é sempre obrigatório utilizar parênteses, sem nada lá dentro, como se vê acima.

## 6. Outros Tipos de Dados

Neste capítulo discutimos como se podem criar em C tipos de dados mais avançados e estruturados.

### 6.1. Estruturas

As estruturas em C são muito semelhantes aos registos em Pascal. Agrupam num único tipo vários valores (campos), que podem ser de tipos diferentes.

Exemplo:

```
1 struct people {  
2     char name[50];  
3     int age;  
4     float salary;  
5 };  
6 struct people John;
```

As declarações anteriores definem uma estrutura chamada `people` e uma variável (`John`) desse tipo. Note-se que a estrutura definida tem um nome (*tag*) que é opcional. Quando as estruturas são definidas com *tag* podemos declarar posteriormente variáveis, argumentos de funções, e também tipos de retorno, usando esse *tag*, como se mostrou no exemplo anterior. É também possível definir estruturas sem *tag* (anónimas).

Neste último caso as variáveis terão de ser nomeadas entre o último `}` e `;` da forma habitual.

Por exemplo:

```
1 struct {  
2     char name[50];  
3     int age;  
4     float salary;  
5 } John;
```

Podemos ainda inicializar as variáveis do tipo estrutura quando da sua declaração, colocando os valores pela ordem dos campos definidos na estrutura entre chavetas:

```
1 struct people John = { "John Smith", 26, 124.5 };
```

Para aceder um campo, ou membro, de uma estrutura utiliza-se, à semelhança do Pascal, o operador `.`

Para aumentar o salário do Sr. John Smith deveria escrever-se:

```
1 John.salary = 150.0;
```

### 6.2. Uso de typedef

A definição de novos tipos com `typedef` pode também ser efectuada com estruturas. A declaração seguinte define um novo tipo `person`, podendo posteriormente declarar-se e inicializar-se variáveis desse tipo:

```
1 typedef struct people {
2     char name[50];
3     int age;
4     float salary;
5 } person;
6
7 person John = { "John Smith", 26, 124.5 };
```

Neste exemplo o nome `people` também funciona como *tag* da estrutura e também é opcional. Nesta situação a sua utilidade é reduzida.

vectores e estruturas podem ser misturados (aliás como quaisquer outros tipos):

```
1 typedef struct people {
2     char name[50];
3     int age;
4     float salary;
5 } person;
6
7 person team[1000];
```

Aqui a variável `team` é composta por 1000 `person`'s, e poderíamos fazer os seguintes acessos, por exemplo:

```
1 team[41].salary = 150.0;
2 total_salaries += team[501].salary;
```

### 6.3. Uniões

Uma variável do tipo união pode conter (em instantes diferentes) valores de diferentes tipos e tamanhos. Na linguagem C a declaração de uniões é em tudo semelhante à declaração de estruturas, empregando-se a palavra `union` em vez de `struct`.

Por exemplo, podemos ter:

```
1 union number {
2     short sort_number;
3     long long_number;
4     double real_number;
5 } a_number;
```

Aqui declara-se uma união chamada `number` e uma variável desse tipo – `a_number`. Esta variável poderá conter, em instantes diferentes, um `short`, um `long` ou um `double`. A distinção faz-se pelo nome do campo respectivo. Quando se preenche um determinado campo, o que estiver noutro qualquer é destruído. Os vários campos têm todos o mesmo endereço na memória.

Os campos ou membros são acedidos da mesma forma do que nas estruturas:

```
1 printf("%ld\n", a_number.long_number);
```

Quando o compilador reserva memória para armazenar uma união apenas reserva o espaço suficiente para o membro maior (no exemplo de cima serão 8 bytes para o `double`). Os outros membros ficam sobrepostos, com o mesmo endereço inicial.

Uma forma comum de num programa se tomar nota do membro activo em cada instante é incluir a união dentro de uma estrutura, onde se acrescenta um outro membro com essa função. Examine atentamente o exemplo que se segue:

```

1 typedef struct {
2     int max_passengers;
3 } jet;
4
5 typedef struct {
6     int lift_capacity;
7 } helicopter;
8
9 typedef struct {
10    int max_payload;
11 } cargo_plane;
12
13 typedef union {
14     jet jet_unit;
15     helicopter heli_unit;
16     cargo_plane cargo_unit;
17 } aircraft;
18
19 typedef struct {
20     aircraft_type kind;
21     int speed;
22     aircraft description;
23 } an_aircraft;
```

No tipo `an_aircraft` inclui-se um membro `description` que é do tipo `aircraft` que é, por sua vez uma união de 3 estruturas diferentes (`jet`, `helicopter` e `cargo_plane`). Inclui-se ainda no tipo `an_aircraft` um membro (`kind`) que indica qual é o membro válido no momento para a união `description`.

## 6.4. Conversão entre tipos

O C é uma das poucas linguagens que permite a mudança de tipo das suas variáveis. Para isso usa-se o operador (`cast`), onde `cast` é o novo tipo pretendido. Assim é possível escrever:

```

1 int k;
2 float r = 9.87;
3
4 k = (int) r;
```

A variável `k` ficará com o valor 9, sendo a parte fraccionária de `r` descartada.

O *casting* pode ser usado com qualquer um dos tipos simples, e muitas vezes quando é omitido o compilador executa a conversão silenciosamente.



Este comportamento do C pode ser muito útil em certas alturas, mas também pode ser a origem de muitos erros!

Outro exemplo:

```
1 int k;  
2 char ch = 'A';  
3 k = (int) ch;
```

Aqui o valor de k será 65 (o código ascii do carácter 'A').

Outro uso frequente do *casting* consiste em assegurar que a divisão entre inteiros não seja uma divisão inteira. Basta para isso converter para real o numerador ou denominador. O outro operando é assim também automaticamente convertido, antes de se efectuar a operação:

```
1 int j = 2, k = 5;  
2 float r;  
3  
4 r = (float) j / k;  
5 /* r = 0.4 em vez de 0, se não se fizesse o cast de j */
```

## 6.5. Variáveis estáticas

É possível, quando da declaração de variáveis, locais ou não, prefixá-las com o qualificador *static*. As variáveis locais estáticas são inicializadas uma só vez e não desaparecem quando a função a que pertencem termina, podendo no entanto ser acedidas apenas dentro da função. As variáveis globais estáticas apenas podem ser acedidas no ficheiro onde são declaradas, não sendo exportadas para o exterior.

As variáveis locais estáticas também mantêm o seu valor de umas chamadas para as outras, como se vê no exemplo seguinte:

```
1 #include <stdio.h>  
2  
3 void stat(void);      /* Protótipo de stat() */  
4  
5 int main(void)  
6 {  
7     int k;  
8  
9     for (k=0; k<5; k++)  
10         stat();  
11     return(0);  
12 }  
13  
14 void stat(void)  
15 {  
16     int var = 0;  
17     static int st_var = 0;  
18  
19     printf("var = %d, auto_var = %d\n", var++, auto_var++);  
20 }
```



A saída deste programa será:

```
var = 0, auto_var = 0  
var = 0, auto_var = 1  
var = 0, auto_var = 2  
var = 0, auto_var = 3  
var = 0, auto_var = 4
```

A variável `var` é criada e inicializada de cada vez que a função é chamada. A variável `st_var` só é inicializada da primeira vez e lembra-se do valor que tinha na última vez que a função terminou.

## 7. Apontadores

Os apontadores são uma parte fundamental da linguagem C. O C utiliza os apontadores de forma bastante intensa.

Algumas razões:

- São a única forma de exprimir certas operações;
- Produzem código compacto e eficiente;
- Constituem uma ferramenta bastante poderosa para a manipulação da informação.

O C utiliza apontadores explicitamente com certas construções:

- vectores;
- Estruturas;
- Funções



Os apontadores são talvez a parte mais difícil de dominar na linguagem C. A implementação no C é algo diferente das outras linguagens.

### 7.1. O que são apontadores?

Um apontador é uma variável que contém o endereço de memória de outra variável. É possível ter um apontador para qualquer tipo de variável. O operador unário `&` dá o «endereço de uma variável». Por sua vez, o operador unário `*` dá-nos o «conteúdo de um objecto apontado por um apontador». Para definir um apontador para uma variável de um determinado tipo, pode, por exemplo, usar-se a seguinte declaração:

```
1 int *pointer;
```



É sempre necessário associar um tipo a um apontador. Assim, por exemplo, um apontador para um inteiro fica diferente de um apontador para um longo.

Considere-se o efeito do código seguinte:

```
1 int x = 1, y = 2;  
2 int *ip;  
3  
4 ip = &x;  
5 y = *ip;  
6 x = (int) ip;  
7 *ip = 3;
```

Vale a pena considerar o que passa ao nível da memória da máquina onde é executado. Vamos assumir que a variável `x` se encontra armazenada na posição de memória 100, `y` na posição 200 e `ip` na posição 1000. Note-se que um apontador é uma variável como as outras e os valores

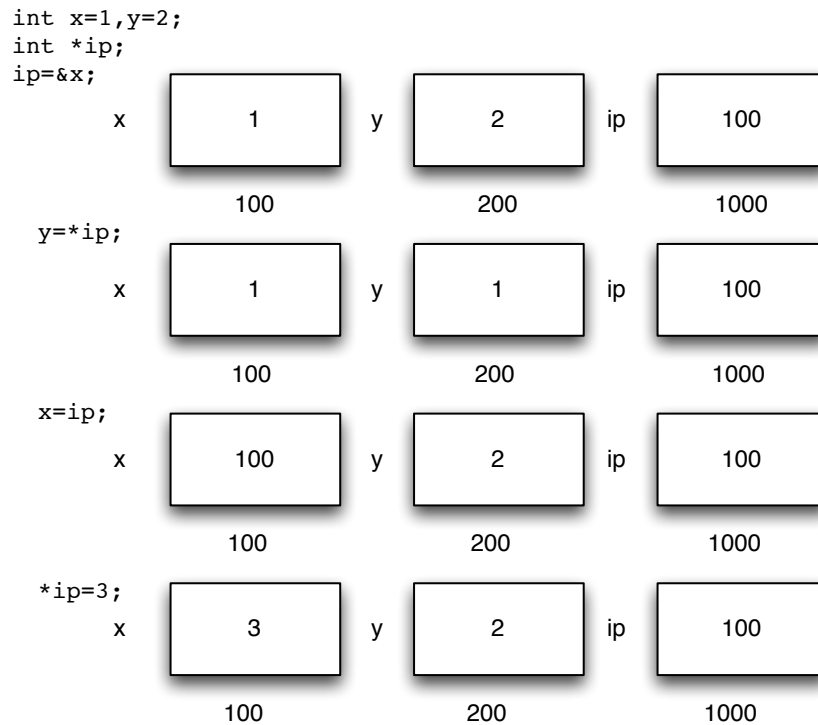


Figura 7.1.: Ilustração do uso de apontadores

que lhe estão associados têm de ser armazenados algures na memória. É a natureza dos valores que se armazenam nos apontadores que é novo.

Veja-se a figura 7.1:

As atribuições `x=1` e `y=2` obviamente colocam esses valores nas posições de memória das variáveis `x` e `y`. `ip` é declarado como sendo um apontador para um inteiro e aí é colocado o endereço da variável `x` (`&x`).

Assim `ip` fica com o valor 100. A seguir, a variável `y` fica com o valor da posição de memória cujo endereço se encontra em `ip`. No exemplo `ip` aponta para a posição de memória 100 - que é o endereço de `x`. Assim `y` fica com o valor de `x`, que é 1.

Já vimos que o C permite conversões de tipos. Assim é possível converter um apontador num inteiro (se tiverem o mesmo tamanho - em geral 32 bits). O valor de `ip` na 3ª instrução é transferido para `x`. Finalmente transfere-se para o local apontado por `ip` o valor 3. Como `ip` contém o endereço de `x` é para aí que vai parar o valor 3.



Quando se declara um apontador ele não aponta para lugar nenhum. Compete ao programador colocar no apontador um endereço válido, antes de o poder utilizar.

Assim o código seguinte poderá gerar um erro. (Um *crash* do programa!).

```

1 int *ip;
2 *ip = 100;

```

Um uso correcto, poderia ser, por exemplo:

```
1 int *ip, x;
2
3 ip = &x;
4 *ip = 100;
```

O conteúdo da posição de memória apontada pelo apontador pode ser usado normalmente em expressões aritméticas:

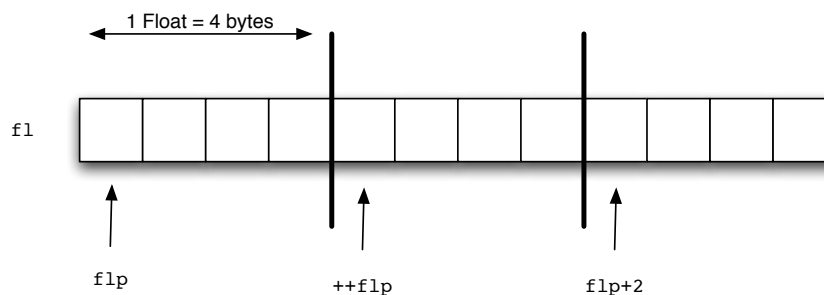
```
1 float *flp, *flq;
2
3 *flp = *flq + 10;
4 *flp = *flq + 1;          /* Incremento de (*flp) */
5 (*flq)++;                /* Diferente de *flq++ */
6 flq = flp;               /* Aqui transfere-se o conteúdo de um apontador
7                           (endereço de float) para o outro */
```

Como já se disse um apontador é um endereço de memória, que é representado por um valor com o mesmo tamanho de um inteiro na maioria das implementações do C, mas um apontador NÃO é um inteiro (int). Ainda assim são possíveis certas operações aritméticas directamente com os apontadores. Uma das razões porque é necessário indicar o tipo da variável apontada por um apontador é permitir essas operações. Por exemplo, quando se incrementa um apontador, isso, em geral, não corresponde a somar 1 ao endereço que ele contém, mas sim o número de bytes que ocupa o tipo de valor para onde ele aponta. O incremento corresponde à diferença entre dois endereços de memória consecutivos de duas variáveis do tipo declarado para o apontador.

Apenas para um apontador para char corresponde a operação ++char\_ptr a incrementar 1 ao valor que está em char\_ptr, mas apenas se os chars tiverem apenas 8 bits, e o endereçamento funcionar ao byte.

Para apontadores para int's (de 32 bits) e float's também de 32 bits, ++ptr corresponde a somar 4 ao valor contido em ptr.

Considere-se um float (f1) e um apontador para float (flp), como se mostra na próxima figura.



Se flp apontar para f1, quando se incrementa flp (++flp) ele passa a apontar para uma posição 4 bytes mais à frente (assumindo que 4 bytes é o tamanho de um float). Se se adicionasse 2 (flp + 2) ao valor original de flp (&f1), ele passaria a apontar para um endereço 8 bytes mais à frente (ou seja 2 float's mais à frente). Isto quer dizer que o C ao saber para que variáveis um apontador aponta, serve-se disso para determinar automaticamente a escala dos incrementos (ou decrementos) que aplica a esse apontador.

## 7.2. Apontadores e funções

Examinemos agora a relação próxima entre apontadores e outras construções do C.

Quando se passam argumentos para funções em C, estes são passados por valor (excepto os vectores, como veremos); ou seja, é criada uma cópia do argumento no stack do processador e é essa cópia que chega à função; quando a função termina essa cópia desaparece. No entanto há situações em que a passagem de argumentos por valor não é muito conveniente; por exemplo, quando queremos que modificações feitas aos argumentos no interior das funções cheguem a quem as chamou, ou quando necessitamos de passar argumentos de tamanho muito elevado (estruturas com muitos membros). Certas linguagens permitem então a passagem de parâmetros de outra forma — passagem por referência — em que a informação que chega à função é apenas o endereço do local na memória onde se encontra o valor do argumento (por exemplo, argumentos declarados como `var` no Pascal). No entanto o C não permite esse mecanismo sendo necessário o uso explícito de apontadores para o implementar<sup>1</sup>.

Por exemplo, se quiséssemos escrever uma função para trocar o conteúdo de duas variáveis, a forma habitual com a chamada `swap(i, j)`; não funcionaria:

```
1 void swap(int a, int b)
2 {
3     int t;
4
5     t = a; a = b; b = t;
6 }
```

Para funcionar teríamos de recorrer a apontadores e ao operador `&` de obtenção do endereço de uma variável com a chamada `swap(&i, &j)`;

```
1 void swap(int *a, int *b)
2 {
3     int t;
4
5     t = *a; *a = *b; *b = t;
6 }
```

É possível também retornar apontadores como resultado de uma função, usado geralmente quando se pretende retornar informação que ocupe um grande espaço, como as estruturas:

```
1 typedef struct { float x, y, z; } coord;
2
3 coord *coord_fn(void);
4
5 int main(void)
6 {
7     coord p1;
8     ...
9     p1 = *coord_fn();
10    ...
11 }
12
13 coord *coord_fn(void)
```

<sup>1</sup>O C++ já contém esse mecanismo.

```

14 {
15     coord p;
16     p = ... ;
17     return &p;
18 }

```

Aqui retorna-se um apontador cujo conteúdo é imediatamente usado para copiar para uma variável do tipo correspondente, o valor para o qual o apontador «está a apontar». Deve fazer-se esta transferência de imediato uma vez que a variável *p* é local à função e desaparece quando a função retorna, podendo a memória que lhe corresponde ser reutilizada.

### 7.3. Apontadores e vectores

Os apontadores e os vectores estão numa relação muito próxima na linguagem C. Um vector não é mais do que um bloco de memória onde são armazenados em posições sucessivas valores do mesmo tipo (ocupando por isso o mesmo número de bytes). Podemos pensar no nome do vector como representando o endereço desse bloco de memória. Realmente o C considera quase equivalentes esses dois pontos de vista.

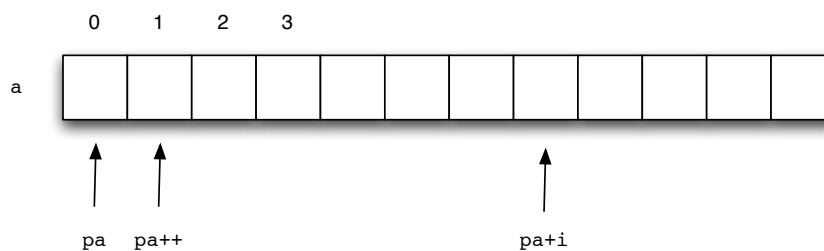
Vejamos o seguinte código:

```


1 int a[10], x;
2 int *pa;
3
4 pa = &a[0];      /* pa fica a apontar para a[0] - endereço inicial do
5                  vector a[] */
6
7 x = *pa;         /* x passa a ser igual a a[0] */

```

Observe-se agora a figura seguinte:



Para se aceder à posição *i* do vector podemos escrever  $*(pa+i)$ , que é equivalente a  $a[i]$ .

 O C não testa de maneira alguma a validade de acessos fora do bloco de memória previamente declarado ou alocado. É da inteira responsabilidade do programador assegurar que esses acessos não são feitos.

No entanto a identificação entre apontadores e vectores vai mais longe. Por exemplo, é válido escrever  $pa = a$ ; em vez de  $pa = \&a[0]$ ; e como vimos  $a[i] \Leftrightarrow *(pa + i)$ , ou seja  $\&a[i] \Leftrightarrow pa + i$ .

Outras construções equivalentes são  $pa[i]$  e  $*(pa + i)$ .

No entanto existe uma diferença entre vectores e apontadores:

- Um apontador é uma variável simples. Podem executar-se as seguintes instruções: `pa = a` e `pa++`
- Um vector não é uma variável simples pelo que as seguintes construções são inválidas: `a = pa` e `a++`

A passagem de um vector como argumento de uma função faz-se sempre por referência, ou seja o que realmente é passado à função é o endereço da primeira posição do vector, o que é equivalente a passar um apontador para essa posição de memória.

Assim a chamada à função `int strlen(char s[]);` pode ser feita de forma equivalente das seguintes maneiras:

```
strlen(s)
```

```
strlen(&s[0])
```

Isto se admitirmos que `s` é um vector de caracteres terminado com o carácter de código 0 (string). Da mesma forma a declaração da função `strlen` também poderia ser feita como<sup>2</sup>:

```
int strlen(char *s);
```

Esta função poderia ter sido escrita da forma seguinte:

```
1 int strlen(char *s)
2 {
3     char *p = s;
4
5     while (*p != '\0')    /* ou mais simplesmente while (*p++); */
6         p++;              /* uma vez que qualquer valor != de 0 é
7                             considerado true */
8     return (p-s);         /* número de caracteres que cabem entre os
9                             endereços */
10 }                          /* representados por p e s */
```

Outro exemplo: uma função que copia uma string para outra; `strcpy()` é também uma função da biblioteca standard do C.

```
1 void strcpy(char *s, char *t)
2 {
3     while (*t++ = *s++);
4 }
```

É de lembrar que o último carácter de uma string em C deve ter o código 0 que faz com que o ciclo `while` termine.

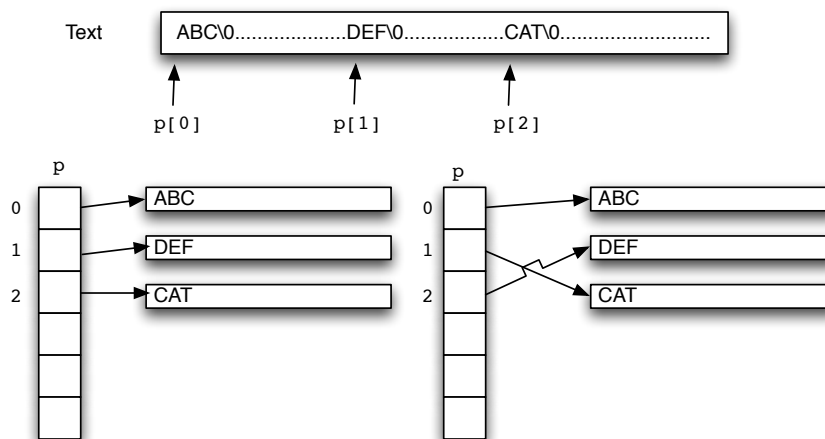
<sup>2</sup>`strlen()` é uma função da biblioteca standard do C que retorna o tamanho de uma string.

## 7.4. Vectores de apontadores

Uma vez que os apontadores são valores de um tipo definido no C, nada nos impede de declararmos e usarmos vectores de apontadores. Para certas situações eles são até muito úteis.

Imaginemos que queríamos ordenar linhas de texto (strings) que se apresentam com comprimentos muito diferentes. Uma forma bastante eficiente de fazer isso em memória poder ser a que se esquematiza a seguir:

1. Armazenar todas as strings num único vector de caracteres em que se usa o carácter '\n' (newline) para separar as várias linhas;
2. Armazenar apontadores num vector diferente onde cada um deles aponta para o 1º carácter de cada linha;
3. Comparar as linhas usando a função standard `strcmp()`;
4. Se duas linhas estiverem fora de ordem trocam-se apenas os apontadores e não os caracteres de cada linha;
5. Quando tudo estiver ordenado escrevem-se as linhas pela ordem com aparecem no vector de apontadores.



Este esquema elimina:

- estratégias complicadas de armazenamento;
- a grande ineficiência de movimentar os caracteres individuais de cada linha de texto.

## 7.5. Apontadores e estruturas

A combinação destes 2 tipos é simples e geralmente sem problemas. Considere-se o seguinte exemplo:

```
1 struct coord {float x, y, z; } pt;
2 struct coord *ppt;
3 ppt = &pt;
```



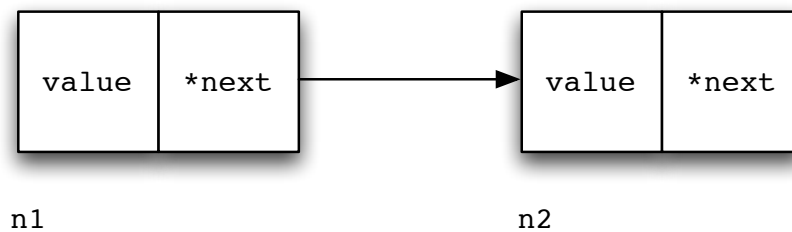
Em C existe o operador `->` que nos permite aceder directamente a membros de uma estrutura apontada por um apontador, como nas 2 linhas seguintes:

```
1 ppt->x = 1.0;  
2 ppt->y = ppt->z - 3.1;
```

Outro exemplo: listas ligadas

```
1 typedef struct {  
2     int value;  
3     element *next;  
4 } element;  
5  
6 element n1, n2;  
7  
8 n1.next = &n2;  
9
```

O código de cima liga o elemento `n1` a `n2` através do seu membro `next`, como se mostra na figura:



## Bibliografia

- BANAHAN, Mike; BRADY, Declan; DORAN, Mark – *The C book*. 2.<sup>a</sup> edição. New York: Addison-Wesley, 1991 (URL: [http://publications.gbdirect.co.uk/c\\_book/](http://publications.gbdirect.co.uk/c_book/)) – Acedido em Março de 2007. ISBN 0201544334
- DEITEL, Harvey M.; DEITEL, Paul J. – *C: how to program*. 4.<sup>a</sup> edição. Upper Saddle River, New Jersey: Prentice Hall, 2004. ISBN 0-13-122543-X
- ECKEL, Bruce – *Thinking in C, beta 3*. 2007 (URL: <http://mindview.net/CDs/ThinkingInC/beta3>) – Acedido em Março de 2007
- GOUGH, Brian – *An Introduction to GCC*. Bristol, United Kingdom: Network Theory Limited, 2004 (URL: <http://www.network-theory.co.uk/gcc/intro/>). ISBN 0-9541617-9-3
- GRÖTKER, Thorsten *et al.* – *The Developer's Guide to Debugging*. Springer, 2008 (URL: <http://www.debugging-guide.com>). ISBN 978-1-4020-5540-9
- HAGEN, William von – *The Definitive guide to GCC*. 2.<sup>a</sup> edição. Berkeley, CA: Apress, 2006 (URL: <http://www.apress.com/book/downloadfile/2932>). ISBN 1-59059-585-8
- HORTON, Ivor – *Beginning C: from novice to professional*. 4.<sup>a</sup> edição. Berkeley, CA: Apress, 2006 (URL: <http://www.apress.com>), p. 640. ISBN 1590597354
- JONES, Derek M. – *The new C standard*. New York: Addison-Wesley Professional, 2003 (URL: <http://www.knosof.co.uk/cbook/cbook.html>) – Acedido em Março de 2007. ISBN 0201709171
- KERNIGHAN, Brian W.; RITCHIE, Dennis M. – *The C programming language*. 2.<sup>a</sup> edição. Englewood Cliffs: Prentice Hall, 1988. ISBN 0-13-110362-8
- KOENIG, Andrew – *C Traps and Pitfalls*. Reading, Massachusetts: Addison-Wesley, 1989. ISBN 0-201-17928-8
- LINDEN, Peter van der – *Expert C Programming: Deep C Secrets*. Englewood Cliffs, NJ: Prentice Hall, 1994. ISBN 0-13-177429-8
- OUALLINE, Steve – *Practical C programming*. 3.<sup>a</sup> edição. Sebastopol, CA: O'Reilly & Associates, 1987. ISBN 1-56592-306-5
- SAMPAIO, Isabel; SAMPAIO, Alberto – *Fundamental da programação em C*. 3.<sup>a</sup> edição. Lisboa: FCA – Editora de informática, 1988. ISBN 972-722-130-0
- SEEBACH, Peter – *Infrequently asked C questions in comp.lang.c*. 1995 (URL: <http://www.plethora.net/~seebs/faqs/c-iaq.html>) – Acedido em Março de 2007

## A. Exemplos e exercícios de programação

De seguida apresentam-se exemplos de programas simples, retirados de um Tutorial para C em MS-DOS, da Coronado Enterprises, e adaptados para Unix, de forma estarem compatíveis com as novas normas da linguagem C.

### A.1. Programas simples

**ex001.c** – Não faz nada.

```
1 int main()
2 {
3     return(0);
4 }
```

**ex002.c** – Não faz nada

```
1 int main()
2 {
3     return(1);
4 }
```

Estes dois primeiros exemplos mostram dois programas simples que não fazem nada, excepto o básico e fundamental que é devolver um código de saída ao sistema operativo. O primeiro simula um funcionamento sem erros, e o segundo um funcionamento com erros. Se o funcionamento for normal o programa deve devolver zero ao sistema operativo, e um valor diferente de zero se acontecer qualquer erro.



A função `main` em C deve ser sempre do tipo `int` e devolver um valor com `return`, excepto quando saímos antecipadamente usando a função `exit`.

**ex003.c** – Um exemplo de impressão em C.

```
1 #include<stdio.h>
2 int main()
3 {
4     printf("This is a line of text to output.");
5     return(0);
6 }
```

**ex004.c** – Um exemplo de várias impressões em C.

```
1 #include<stdio.h>
2 int main()
3 {
4     printf("This is a line of text to output.\n");
```

```
5   printf("And this is another ");
6   printf("line of text.\n\n");
7   printf("This is a third line.\n");
8   return(0);
9 }
```

**ex005.c** – Como alterar o valor de uma variável.

```
1  #include<stdio.h>
2  int main()
3  {
4  int index;
5      index = 13;
6      printf("The value of the index is %d\n",index);
7      index = 27;
8      printf("The value of the index is %d\n",index);
9      index = 10;
10     printf("The value of the index is %d\n",index);
11     return(0);
12 }
```

**ex006.c** – Como inserir comentários num programa.

```
1      /* This is a comment ignored by the compiler */
2  #include<stdio.h>
3  int main() /* This is another comment ignored by the compiler */
4  {
5      printf("We are looking at how comments are ");
6                                     /* A comment is
7                                     allowed to be
8                                     continued on
9                                     another line */
10     printf("used in C.\n");
11     return(0);
12 }
13     /* One more comment for effect */
```

**ex007.c** – Um programa bem formatado

```
1  #include<stdio.h>
2  int main() /* Main program starts here */
3  {
4      printf("Good form ");
5      printf          ("can aid in ");
6      printf          ("understanding a program.\n");
7      printf("And bad form ");
8      printf          ("can make a program ");
9      printf          ("unreadable.\n");
10     return(0);
11 }
```

**ex008.c** – Um programa mal formatado

```
1 #include<stdio.h>
2 int main() /* Main program starts here */{printf("Good form ")
3 ;printf
4 ("can aid in ");printf("understanding a program.\n")
5 ;printf("And bad form ");printf("can make a program ");
6 printf("unreadable.\n"); return(0);}
```

**Exercícios**

1. Escreva um programa que mostre o seu nome.
2. Modifique o programa anterior para mostrar também o seu endereço e o seu numero de telefone (podem ser imaginários) adicionando dois printf's ao programa anterior.

**A.2. Ciclos e decisões**

**ex009.c** – Exemplo de utilização de um ciclo while.

```
1 #include<stdio.h>
2 /* This is an example of a "while" loop */
3
4 int main()
5 {
6     int count;
7
8     count = 0;
9     while (count < 6) {
10         printf("The value of count is %d\n",count);
11         count = count + 1;
12     }
13     return(0);
14 }
```

**ex010.c** – Exemplo de utilização de um ciclo do...while.

```
1 #include<stdio.h>
2 /* This is an example of a do-while loop */
3
4 int main()
5 {
6     int i;
7
8     i = 0;
9     do {
10         printf("The value of i is now %d\n",i);
11         i = i + 1;
12     } while (i < 5);
13     return(0);
14 }
```

**ex011.c** – Exemplo de utilização de um ciclo for.

```
1 #include<stdio.h>
2 /* This is an example of a for loop */
3
4 int main()
5 {
6     int index;
7
8     for(index = 0;index < 6;index = index + 1)
9         printf("The value of the index is %d\n",index);
10    return(0);
11 }
```

**ex012.c** – Exemplo de utilização de um if...else.

```
1 #include<stdio.h>
2 /* This is an example of the if and the if-else statements */
3
4 int main()
5 {
6     int data;
7
8     for(data = 0;data < 10;data = data + 1) {
9
10        if (data == 2)
11            printf("Data is now equal to %d\n",data);
12
13        if (data < 5)
14            printf("Data is now %d, which is less than 5\n",data);
15        else
16            printf("Data is now %d, which is greater than 4\n",data);
17
18    } /* end of for loop */
19    return(0);
20 }
```

**ex013.c** – Exemplo de utilização de break e continue.

```
1 #include<stdio.h>
2 int main()
3 {
4     int xx;
5
6     for(xx = 5;xx < 15;xx = xx + 1){
7         if (xx == 8)
8             break;
9         printf("In the break loop, xx is now %d\n",xx);
10    }
11
12 }
```

```

13     for(xx = 5;xx < 15;xx = xx + 1){
14         if (xx == 8)
15             continue;
16         printf("In the continue loop, xx is now %d\n",xx);
17     }
18     return(0);
19 }

```

**ex014.c** – Exemplo de utilização de um switch.

```

1  #include<stdio.h>
2  int main()
3  {
4      int truck;
5
6      for (truck = 3;truck < 13;truck = truck + 1) {
7
8          switch (truck) {
9              case 3 : printf("The value is three\n");
10                 break;
11              case 4 : printf("The value is four\n");
12                 break;
13              case 5 :
14              case 6 :
15              case 7 :
16              case 8 : printf("The value is between 5 and 8\n");
17                 break;
18              case 11 : printf("The value is eleven\n");
19                 break;
20              default : printf("It is one of the undefined values\n");
21                 break;
22          } /* end of switch */
23
24      } /* end of for loop */
25      return(0);
26  }

```

**ex015.c** – Exemplo de utilização de um goto.

```

1  #include<stdio.h>
2  int main()
3  {
4      int dog,cat,pig;
5
6      goto real_start;
7
8      some_where:
9      printf("This is another line of the mess.\n");
10     goto stop_it;
11

```

```
12 /*the following section is the only section with a useable goto*/
13     real_start:
14     for(dog = 1;dog < 6;dog = dog + 1) {
15         for(cat = 1;cat < 6;cat = cat + 1) {
16             for(pig = 1;pig < 4;pig = pig + 1) {
17                 printf("Dog = %d  Cat = %d  Pig = %d\n",dog,cat,pig);
18                 if ((dog + cat + pig) > 8 ) goto enough;
19             };
20         };
21     };
22     enough: printf("Those are enough animals for now.\n");
23 /*this is the end of the section with a useable goto statement*/
24
25     printf("\nThis is the first line of the spaghetti code.\n");
26     goto there;
27
28     where:
29     printf("This is the third line of spaghetti.\n");
30     goto some_where;
31
32     there:
33     printf("This is the second line of the spaghetti code.\n");
34     goto where;
35
36     stop_it:
37     printf("This is the last line of this mess.\n");
38
39     return(0);
40 }
```

**ex016.c** – Programa que imprime uma tabela de conversão de temperaturas.

```
1  /*****
2  /*
3  /*   This is a temperature conversion program written in   */
4  /*   the C programming language. This program generates   */
5  /*   and displays a table of fahrenheit and centigrade     */
6  /*   temperatures, and lists the freezing and boiling      */
7  /*   of water.                                              */
8  /*
9  /*****
10 #include<stdio.h>
11 int main()
12 {
13     int count;          /* a loop control variable          */
14     int fahrenheit;     /* the temperature in fahrenheit degrees */
15     int centigrade;     /* the temperature in centigrade degrees */
16
17     printf("Centigrade to Farenheit temperature table\n\n");
18
19 }
```



```

20     for(count = -2;count <= 12;count = count + 1){
21         centigrade = 10 * count;
22         fahrenheit = 32 + (centigrade * 9)/5;
23         printf(" C =%4d   F =%4d   ",centigrade,fahrenheit);
24         if (centigrade == 0)
25             printf(" Freezing point of water");
26         if (centigrade == 100)
27             printf(" Boiling point of water");
28         printf("\n");
29     } /* end of for loop */
30     return(0);
31 }

```

**ex017.c** – Programa que imprime uma tabela de conversão de temperaturas, mas de uma forma menos inteligível.

```

1  #include<stdio.h>
2  int main()
3  {
4      int x1,x2,x3;
5
6      printf("Centigrade to Farenheit temperature table\n\n");
7
8      for(x1 = -2;x1 <= 12;x1 = x1 + 1){
9          x3 = 10 * x1;
10         x2 = 32 + (x3 * 9)/5;
11         printf(" C =%4d   F =%4d   ",x3,x2);
12         if (x3 == 0)
13             printf(" Freezing point of water");
14         if (x3 == 100)
15             printf(" Boiling point of water");
16         printf("\n");
17     }
18     return(0);
19 }

```

## Exercícios

1. Escreva um programa que mostre o seu nome no écran dez vezes. Escreva três versões deste programa, cada uma com um tipo de ciclo diferente.
2. Escreva um programa que conte de um a dez, mostre os valores no écran, cada um na sua linha, e tenha uma mensagem escolhida por si quando passar pelo número 3 (três) e outra mensagem diferente quando passar pelo número 7 (sete) .

### A.3. Tipos de dados e atribuições

**ex018.c** – Atribuições e operações com inteiros.

```
1  /* This program will illustrate the assignment statements */
2  #include<stdio.h>
3
4
5  int main()
6  {
7      int a,b,c;    /* Integer variables for examples */
8
9      a = 12;
10     b = 3;
11     c = a + b;      /* simple addition */
12     c = a - b;      /* simple subtraction */
13     c = a * b;      /* simple multiplication */
14     c = a / b;      /* simple division */
15     c = a % b;      /* simple modulo (remainder) */
16     c = 12*a + b/2 - a*b*2/(a*c + b*2);
17     c = c/4+13*(a + b)/3 - a*b + 2*a*a;
18     a = a + 1;      /* incrementing a variable */
19     b = b * 5;
20
21     a = b = c = 20;  /* multiple assignment */
22     a = b = c = 12*13/4;
23     return(0);
24 }
```

**ex019.c** – Atribuições e operações com outros tipos de dados.

```
1  /* The purpose of this file is to */
2  /* introduce additional data types */
3
4  #include<stdio.h>
5  int main()
6  {
7      int a,b,c;
8      char x,y,z;
9      float num,toy,thing;
10
11     a = b = c = -27;
12     x = y = z = 'A';
13     num = toy = thing = 3.6792;
14
15     a = y;          /* a is now 65 (character A) */
16     x = b;          /* x is now -27 */
17     num = b;        /* num will now be -27.00 */
18     a = toy;        /* a will now be 3 */
19     return(0);
20 }
```

**ex020.c** – Ainda mais tipos de dados.

```

1  #include<stdio.h>
2  int main()
3  {
4      int a;                /* simple integer type          */
5      long int b;           /* long integer type       */
6      short int c;          /* short integer type      */
7      unsigned int d;       /* unsigned integer type   */
8      char e;               /* character type          */
9      float f;              /* floating point type     */
10     double g;              /* double precision floating point */
11
12     a = 1023;
13     b = 2222;
14     c = 123;
15     d = 1234;
16     e = 'X';
17     f = 3.14159;
18     g = 3.1415926535898;
19
20     printf("a = %d\n",a);    /* decimal output          */
21     printf("a = %o\n",a);    /* octal output             */
22     printf("a = %x\n",a);    /* hexadecimal output       */
23     printf("b = %ld\n",b);    /* decimal long output      */
24     printf("c = %d\n",c);    /* decimal short output     */
25     printf("d = %u\n",d);    /* unsigned output          */
26     printf("e = %c\n",e);    /* character output         */
27     printf("f = %f\n",f);    /* floating output          */
28     printf("g = %f\n",g);    /* double float output      */
29     printf("\n");
30     printf("a = %d\n",a);    /* simple int output        */
31     printf("a = %7d\n",a);    /* use a field width of 7   */
32     printf("a = %-7d\n",a);   /* left justify in field of 7 */
33     c = 5;
34     d = 8;
35     printf("a = %*d\n",c,a);  /* use a field width of 5    */
36     printf("a = %*d\n",d,a);  /* use a field width of 8    */
37     printf("\n");
38     printf("f = %f\n",f);     /* simple float output       */
39     printf("f = %12f\n",f);   /* use field width of 12     */
40     printf("f = %12.3f\n",f); /* use 3 decimal places     */
41     printf("f = %12.5f\n",f); /* use 5 decimal places     */
42     printf("f = %-12.5f\n",f); /* left justify in field    */
43     return(0);
44 }

```

**ex021.c** – Como fazer comparações.

```

1 #include<stdio.h>
2
3 int main() /* This file will illustrate logical compares */
4 {
5     int x = 11,y = 11,z = 11;
6     char a = 40,b = 40,c = 40;
7     float r = 12.987,s = 12.987,t = 12.987;
8
9     /* First group of compare statements */
10
11     if (x == y) z = -13; /* This will set z = -13 */
12     if (x > z) a = 'A'; /* This will set a = 65 */
13     if (!(x > z)) a = 'B'; /* This will change nothing */
14     if (b <= c) r = 0.0; /* This will set r = 0.0 */
15     if (r != s) t = c/2; /* This will set t = 20 */
16
17     /* Second group of compare statements */
18
19     if (x = (r != s)) z = 1000; /* This will set x = some positive
20                                number and z = 1000 */
21     if (x = y) z = 222; /* This sets x = y, and z = 222 */
22     if (x != 0) z = 333; /* This sets z = 333 */
23     if (x) z = 444; /* This sets z = 444 */
24
25     /* Third group of compare statements */
26
27     x = y = z = 77;
28     if ((x == y) && (x == 77)) z = 33; /* This sets z = 33 */
29     if ((x > y) || (z > 12)) z = 22; /* This sets z = 22 */
30     if (x && y && z) z = 11; /* This sets z = 11 */
31     if ((x = 1) && (y = 2) && (z = 3)) r = 12.00; /* This sets
32                                                    x = 1, y = 2, z = 3, r = 12.00 */
33     if ((x == 2) && (y = 3) && (z = 4)) r = 14.56; /* This doesn't
34                                                    change anything */
35
36     /* Fourth group of compares */
37
38     if (x == x); z = 27.345; /* z always gets changed */
39     if (x != x) z = 27.345; /* Nothing gets changed */
40     if (x = 0) z = 27.345; /* This sets x = 0, z is unchanged */
41
42     return(0);
43 }

```



Ao compilar este programa o compilador avisa que estamos a utilizar o resultado de uma atribuição como um valor lógico. Os três avisos resultam desse facto.

**ex022.c** – Coisas mesmo *estranhas* do C.

```

1  #include<stdio.h>
2
3  int main()
4  {
5      int x = 0,y = 2,z = 1025;
6      float a = 0.0,b = 3.14159,c = -37.234;
7
8                                     /* incrementing */
9      x = x + 1;      /* This increments x */
10     x++;           /* This increments x */
11     ++x;           /* This increments x */
12     z = y++;        /* z = 2, y = 3 */
13     z = ++y;        /* z = 4, y = 4 */
14
15                                     /* decrementing */
16     y = y - 1;      /* This decrements y */
17     y--;           /* This decrements y */
18     --y;           /* This decrements y */
19     y = 3;
20     z = y--;        /* z = 3, y = 2 */
21     z = --y;        /* z = 1, y = 1 */
22
23                                     /* arithmetic op */
24     a = a + 12;      /* This adds 12 to a */
25     a += 12;        /* This adds 12 more to a */
26     a *= 3.2;        /* This multiplies a by 3.2 */
27     a -= b;          /* This subtracts b from a */
28     a /= 10.0;       /* This divides a by 10.0 */
29
30                                     /* conditional expression */
31     a = (b >= 3.0 ? 2.0 : 10.5 ); /* This expression */
32
33     if (b >= 3.0)     /* And this expression */
34         a = 2.0;      /* are identical, both */
35     else              /* will cause the same */
36         a = 10.5;     /* result. */
37
38     c = (a > b?a:b);   /* c will have the max of a or b */
39     c = (a > b?b:a);   /* c will have the min of a or b */
40
41     return(0);
42 }

```

**Exercícios**

1. Escreva um programa que conte de 1 (um) até 12 (doze) e imprima o número e o seu quadrado para cada um dos valores percorridos pelo ciclo.
2. Escreva um programa que conte de 1 (um) até 12 (doze) e imprima o número e o seu inverso ( $1/x$ ) para cada um dos valores do ciclo.

## A.4. Funções

**ex023.c** – Exemplo de definição e uso de uma função.

```
1 #include<stdio.h>
2 int sum; /* This is a global variable */
3
4 void header();
5 void square( int number);    /* function prototype */
6 void ending();
7
8
9 int main()
10 {
11     int index;
12
13     header();    /* This calls the function named header */
14     for (index = 1;index <= 7;index++)
15         square(index); /* This calls the square function */
16     ending();    /* This calls the ending function */
17     return(0);
18 }
19
20 void header()    /* This is the function named header */
21 {
22     sum = 0;    /* Initialize the variable "sum" */
23     printf("This is the header for the square program\n\n");
24 }
25
26 void square( int number)    /* This is the square function */
27 {
28     int numsq;
29
30     numsq = number * number; /* This produces the square */
31     sum += numsq;
32     printf("The square of %d is %d\n",number,numsq);
33 }
34
35 void ending()    /* This is the ending function */
36 {
37     printf("\nThe sum of the squares is %d\n",sum);
38 }
```

**ex024.c** – Exemplo de definição e uso de várias funções.

```

1  #include<stdio.h>
2
3  int squ(int number);
4
5  int main()  /* This is the main program */
6  {
7  int x,y;
8
9      for(x = 0;x <= 7;x++) {
10         y = squ(x); /* go get the value of x*x */
11         printf("The square of %d is %d\n",x,y);
12     }
13
14     for (x = 0;x <= 7;++x)
15         printf("The value of %d is %d\n",x,squ(x));
16     return(0);
17 }
18
19 int squ(int in) /* function to get the value of in squared */
20 {
21     int square;
22
23     square = in * in;
24     return(square); /* This sets squ() = square */
25 }

```

**ex025.c** – Exemplo de definição e uso de uma função com variáveis e resultados do tipo float.

```

1  #include<stdio.h>
2
3  float z; /* This is a global variable */
4
5
6  int main()
7  {
8  int index;
9  float x,y,sqr(float inval),glsqr();
10
11     for (index = 0;index <= 7;index++){
12         x = index; /* convert int to float */
13         y = sqr(x); /* square x to a floating point variable */
14         printf("The square of %d is %10.4f\n",index,y);
15     }
16
17     for (index = 0; index <= 7;index++) {
18         z = index;
19         y = glsqr();
20         printf("The square of %d is %10.4f\n",index,y);
21     }

```

```
22     return(0);
23 }
24
25 float sqr(float inval) /* square a float, return a float */
26 {
27     float square;
28
29     square = inval * inval;
30     return(square);
31 }
32
33 float glsqr() /* square a float, return a float */
34 {
35     return(z*z);
36 }
```

**ex026.c** – Exemplo do âmbito das variáveis dentro das funções.

```
1  #include <stdio.h> /* Prototypes for Input/Output */
2  void head1(void); /* Prototype for head1 */
3  void head2(void); /* Prototype for head2 */
4  void head3(void); /* Prototype for head3 */
5
6  int count; /* This is a global variable */
7
8  int main()
9  {
10     register int index; /* This variable is available only in main */
11
12     head1();
13     head2();
14     head3();
15
16     /* main "for" loop of this program */
17     for (index = 8; index > 0; index--) {
18         int stuff;
19         /* This variable is only available in these braces */
20         for (stuff = 0; stuff <= 6; stuff++)
21             printf("%d ", stuff);
22         printf(" index is now %d\n", index);
23     }
24     return(0);
25 }
26
27 int counter; /* This is available from this point on */
28 void head1(void)
29 {
30     int index; /* This variable is available only in head1 */
31
32     index = 23;
33     printf("The header1 value is %d\n", index);
34 }
```



```

34 |
35 | void head2(void)
36 | {
37 |   int count; /* This variable is available only in head2 */
38 |              /* and it displaces the global of the same name */
39 |
40 |   count = 53;
41 |   printf("The header2 value is %d\n",count);
42 |   counter = 77;
43 | }
44 |
45 | void head3(void)
46 | {
47 |   printf("The header3 value is %d\n",counter);
48 | }

```

**ex027.c** – Exemplo do uso de funções recursivas.

```

1 | #include<stdio.h>
2 |
3 | void count_dn(int count);
4 | int main()
5 | {
6 |   int index;
7 |
8 |   index = 8;
9 |   count_dn(index);
10 |  return(0);
11 | }
12 |
13 | void count_dn(int count)
14 | {
15 |   count--;
16 |   printf("The value of the count is %d\n",count);
17 |   if (count > 0)
18 |     count_dn(count);
19 |   printf("Now the count is %d\n",count);
20 | }

```

**ex028.c** – Outro exemplo do uso de funções recursivas.

```

1 | #include <stdio.h> /* Prototypes for standard Input/Output */
2 | #include <string.h> /* Prototypes for string operations */
3 |
4 | void forward_and_backwards(char line_of_char[],int index);
5 |
6 | int main()
7 | {
8 |   char line_of_char[80];
9 |   int index = 0;

```

```
10
11     strcpy(line_of_char,"This is a string.\n");
12
13     forward_and_backwards(line_of_char,index);
14     return(0);
15 }
16
17 void forward_and_backwards(char line_of_char[],int index)
18 {
19     if (line_of_char[index]) {
20         printf("%c",line_of_char[index]);
21         index++;
22         forward_and_backwards(line_of_char,index);
23     }
24     printf("%c",line_of_char[index]);
25 }
```

### Exercícios

1. Re-escreva o programa de conversão de temperatura (ex016.c) de modo a que a conversão de temperatura seja efectuada numa função.
2. Escreva um programa que mostre o seu nome dez vezes no écran, chamando uma função para fazer essa escrita.

## A.5. Macros e afins

**ex029.c** – Exemplos do uso de define.

```
1 #include<stdio.h>
2
3 #define START 0 /* Starting point of loop */
4 #define ENDING 9 /* Ending point of loop */
5 #define MAX(A,B) ((A)>(B)?(A):(B)) /* Max macro definition */
6 #define MIN(A,B) ((A)>(B)?(B):(A)) /* Min macro definition */
7
8 int main()
9 {
10     int index,mn,mx;
11     int count = 5;
12
13     for (index = START;index <= ENDING;index++) {
14         mx = MAX(index,count);
15         mn = MIN(index,count);
16         printf("Max is %d and min is %d\n",mx,mn);
17     }
18     return(0);
19 }
```

**ex030.c** – Exemplos da definição e uso de macros.

```

1  #include<stdio.h>
2  #define WRONG(A) A*A*A          /* Wrong macro for cube    */
3  #define CUBE(A) (A)*(A)*(A)     /* Right macro for cube   */
4  #define SQUR(A) (A)*(A)         /* Right macro for square */
5  #define ADD_WRONG(A) (A)+(A)    /* Wrong macro for addition */
6  #define ADD_RIGHT(A) ((A)+(A)) /* Right macro for addition */
7  #define START 1
8  #define STOP 7
9
10 int main()
11 {
12     int i,offset;
13     offset = 5;
14     for (i = START;i <= STOP;i++) {
15         printf("The square of %3d is %4d, and its cube is %6d\n",
16             i+offset,SQUR(i+offset),CUBE(i+offset));
17         printf("The wrong of %3d is %6d\n",i+offset,WRONG(i+offset));
18     }
19     printf("\nNow try the addition macro's\n");
20     for (i = START;i <= STOP;i++) {
21         printf("Wrong addition macro = %6d, and right = %6d\n"
22             ,5*ADD_WRONG(i),5*ADD_RIGHT(i));
23     }
24     return(0);
25 }

```

**ex031.c** – Exemplo do uso do tipo de dados enum.

```

1  #include <stdio.h>
2  int main()
3  {
4      enum {win,tie,bye,lose,no_show} result;
5      enum {sun,mon,tues,wed,thur,fri,sat} days;
6
7      result = win;
8      printf("    win = %d\n",result);
9      result = lose;
10     printf("    lose = %d\n",result);
11     result = tie;
12     printf("    tie = %d\n",result);
13     result = bye;
14     printf("    bye = %d\n",result);
15     result = no_show;
16     printf("no show = %d\n\n",result);
17
18     for(days = mon;days < fri;days++)
19         printf("The day code is %d\n",days);
20     return(0);
21 }

```

**Exercícios**

1. Escreva um programa que conte de 7 (sete) até -5 (menos cinco) decrementando uma variável, e usando o `define` para definir os limites.
2. Adicione instruções `printf` ao ficheiro `ex030.c` de forma a ver os problemas dos macros errados.
3. Use a opção `-E` na compilação do programa anterior para ver melhor a expansão dos macros.

**A.6. Caracteres e strings**

**ex032.c** – Exemplo do uso do caracteres.

```
1 #include<stdio.h>
2 int main()
3 {
4     char name[5];          /* define a string of characters */
5
6     name[0] = 'D';
7     name[1] = 'a';
8     name[2] = 'v';
9     name[3] = 'e';
10    name[4] = 0;          /* Null character - end of text */
11
12    printf("The name is %s\n",name);
13    printf("One letter is %c\n",name[2]);
14    printf("Part of the name is %s\n",&name[1]);
15    return(0);
16 }
```

**ex033.c** – Exemplo do uso de *strings*.

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char name1[12],name2[12],mixed[25];
6     char title[20];
7
8     strcpy(name1,"Rosalinda");
9     strcpy(name2,"Zeke");
10    strcpy(title,"This is the title.");
11    printf("    %s\n\n",title);
12    printf("Name 1 is %s\n",name1);
13    printf("Name 2 is %s\n",name2);
14
15    if(strcmp(name1,name2)>0) /* returns 1 if name1 > name2 */
16        strcpy(mixed,name1);
17    else
18        strcpy(mixed,name2);
```

```

19
20     printf("The biggest name alphabetically is %s\n",mixed);
21
22     strcpy(mixed,name1);
23     strcat(mixed," ");
24     strcat(mixed,name2);
25     printf("Both names are %s\n",mixed);
26     return(0);
27 }

```

**ex034.c** – Exemplo do uso de vectores de inteiros.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int values[12];
6      int index;
7
8      for (index = 0;index < 12;index++)
9          values[index] = 2 * (index + 4);
10
11     for (index = 0;index < 12;index++)
12         printf("The value at index = %2d is %3d\n",index,values[index]);
13     return(0);
14 }

```

**ex035.c** – Exemplo do uso de números em vírgula flutuante.

```

1  #include <stdio.h>
2  char name1[] = "First Program Title";
3
4  int main()
5  {
6      int index;
7      int stuff[12];
8      float weird[12];
9      static char name2[] = "Second Program Title";
10
11     for (index = 0;index < 12;index++) {
12         stuff[index] = index + 10;
13         weird[index] = 12.0 * (index + 7);
14     }
15
16     printf("%s\n",name1);
17     printf("%s\n\n",name2);
18     for (index = 0;index < 12;index++)
19         printf("%5d %5d %10.3f\n",index,stuff[index],weird[index]);
20     return(0);
21 }

```

**ex036.c** – Exemplo de uma função com retorno do resultado.

```
1 #include <stdio.h>
2 void dosome(int list[]);
3
4 int main()
5 {
6     int index;
7     int matrix[20];
8
9     for (index = 0; index < 20; index++)          /* generate data */
10         matrix[index] = index + 1;
11
12     for (index = 0; index < 5; index++)          /* print original data */
13         printf("Start  matrix[%d] = %d\n", index, matrix[index]);
14
15     dosome(matrix);                             /* go to a function & modify matrix */
16
17     for (index = 0; index < 5; index++)          /* print modified matrix */
18         printf("Back   matrix[%d] = %d\n", index, matrix[index]);
19     return(0);
20 }
21
22 void dosome(int list[])          /* This will illustrate returning data */
23 {
24     int i;
25
26     for (i = 0; i < 5; i++)          /* print original matrix */
27         printf("Before matrix[%d] = %d\n", i, list[i]);
28
29     for (i = 0; i < 20; i++)          /* add 10 to all values */
30         list[i] += 10;
31
32     for (i = 0; i < 5; i++)          /* print modified matrix */
33         printf("After  matrix[%d] = %d\n", i, list[i]);
34 }
```

**ex037.c** – Exemplo do uso de matrizes.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i, j;
6     int big[8][8], large[25][12];
7
8     for (i = 0; i < 8; i++)
9         for (j = 0; j < 8; j++)
10             big[i][j] = i * j;          /* This is a multiplication table */
11
12 }
```

```

13   for (i = 0; i < 25; i++)
14       for (j = 0; j < 12; j++)
15           large[i][j] = i + j;           /* This is an addition table */
16
17   big[2][6] = large[24][10]*22;
18   big[2][2] = 5;
19   big[big[2][2]][big[2][2]] = 177;    /* this is big[5][5] = 177; */
20
21   for (i = 0; i < 8; i++) {
22       for (j = 0; j < 8; j++)
23           printf("%5d ", big[i][j]);
24       printf("\n");                  /* newline for each increase in i */
25   }
26   return(0);
27 }

```

## Exercícios

1. Escreva um programa com três *strings* e use a função `strcpy` para colocar os valores "one", "two", e "three" nas três *strings* (um valor em cada uma delas). Concatene as três *strings* numa única e imprima o resultado 10 vezes.
2. Defina dois vectores de inteiros, cada um com 10 (dez) elementos com os nomes de `array1` e `array2`. Preencha-os com valores à sua escolha, e some-os (elemento a elemento) colocando o resultado no vector `arraysum`. Finalmente imprima todos os dados envolvidos da seguinte forma: em cada linha deve imprimir o índice, e o valor dos elementos correspondentes de cada uma das matrizes.

## A.7. Apontadores

**ex038.c** – Exemplo do uso de apontadores.

```

1   #include <stdio.h>
2   int main()                               /* illustration of pointer use */
3   {
4       int index,*pt1,*pt2;
5
6       index = 39;                           /* any numerical value */
7       pt1 = &index;                         /* the address of index */
8       pt2 = pt1;
9       printf("The value is %d %d %d\n",index,*pt1,*pt2);
10      *pt1 = 13;                             /* this changes the value of index */
11      printf("The value is %d %d %d\n",index,*pt1,*pt2);
12      return(0);
13  }

```

**ex039.c** – Outro exemplo do uso de apontadores.

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char strg[40],*there,one,two;
6     int *pt,list[100],index;
7
8     strcpy(strg,"This is a character string.");
9
10    one = strg[0];                /* one and two are identical */
11    two = *strg;
12    printf("The first output is %c %c\n",one,two);
13
14    one = strg[8];                /* one and two are identical */
15    two = *(strg+8);
16    printf("the second output is %c %c\n",one,two);
17
18    there = strg+10;              /* strg+10 is identical to strg[10] */
19    printf("The third output is %c\n",strg[10]);
20    printf("The fourth output is %c\n",*there);
21
22    for (index = 0;index < 100;index++)
23        list[index] = index + 100;
24    pt = list + 27;
25    printf("The fifth output is %d\n",list[27]);
26    printf("The sixth output is %d\n",*pt);
27    return(0);
28 }
```

**ex040.c** – Como devolver valores usando apontadores.

```
1 #include <stdio.h>
2 void fixup(int nuts,int * fruit);
3
4 int main()
5 {
6     int pecans,apples;
7
8     pecans = 100;
9     apples = 101;
10    printf("The starting values are %d %d\n",pecans,apples);
11
12    /* when we call "fixup" */
13    fixup(pecans,&apples); /* we take the value of pecans */
14    /* we take the address of apples */
15
16    printf("The ending values are %d %d\n",pecans,apples);
17    return(0);
18 }
```



```

19
20 void fixup(int nuts,int *fruit) /* nuts is an integer value */
21                                /* fruit points to an integer */
22 {
23     printf("The values are %d %d\n",nuts,*fruit);
24     nuts = 135;
25     *fruit = 172;
26     printf("The values are %d %d\n",nuts,*fruit);
27 }

```

### Exercícios

1. Defina um vector de caracteres e use a função `strcpy` para copiar uma *string* para dentro dela. Imprima a *string* usando um ciclo com um apontador para a imprimir, um carácter de cada vez.
2. Modifique o programa para imprimir a *string* em sentido contrário, do fim para o início.

## A.8. Operações de entrada/saída

**ex041.c** – Exemplo de entrada/saída simples.

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     char c;
7
8     printf("Enter any characters, X = halt program.\n");
9
10    do {
11        c = getchar(); /* get a single character from the kb */
12        putchar(c);    /* display the character on the monitor */
13    } while (c != 'X'); /* until an X is hit */
14
15    printf("\nEnd of program.\n");
16    return(0);
17 }

```



Este programa não se porta da forma esperada, devido ao comportamento *normal* da leitura de caracteres em UNIX. Os caracteres só são normalmente passados ao programa, quando o utilizador *acaba* a linha, para dar a hipótese a este de corrigir erros. Vamos ver nos exemplos seguintes como evitar este comportamento.

**ex042.c** – Como ler carácter a carácter sem imprimir os caracteres.

```
1 #include <stdio.h>
2 #include <termios.h>
3 #include <unistd.h>
4
5 /* mygetch reads a char without echo */
6 int mygetch( ) {
7     struct termios oldt, newt;
8     int ch;
9     tcgetattr(STDIN_FILENO, &oldt);
10    newt = oldt;
11    newt.c_lflag &= ~( ICANON | ECHO );
12    tcsetattr( STDIN_FILENO, TCSANOW, &newt );
13    ch = getchar();
14    tcsetattr( STDIN_FILENO, TCSANOW, &oldt );
15    return (ch);
16 }
17
18 int main()
19 {
20     char c;
21     printf("Enter any characters, terminate program with X\n");
22
23     do {
24         c = mygetch();           /* get a character */
25         putchar(c);             /* display the hit key */
26     } while (c != 'X');
27
28     printf("\nEnd of program.\n");
29     return(0);
30 }
```

**ex043.c** – Como ler carácter a carácter imprimindo os caracteres.

```
1 #include <stdio.h>
2 #include <termios.h>
3 #include <unistd.h>
4
5 /* mygetche reads a char with echo */
6 int mygetche( ) {
7     struct termios oldt, newt;
8     int ch;
9     tcgetattr(STDIN_FILENO, &oldt);
10    newt = oldt;
11    newt.c_lflag &= ~( ICANON );
12    tcsetattr( STDIN_FILENO, TCSANOW, &newt );
13    ch = getchar();
14    tcsetattr( STDIN_FILENO, TCSANOW, &oldt );
15    return (ch);
16 }
```

```

17 |
18 | int main()
19 | {
20 |     char c;
21 |
22 |     printf("Enter any characters, terminate program with X\n");
23 |
24 |     do {
25 |         c = mygetche();           /* get a character */
26 |         putchar(c);              /* display the hit key */
27 |     } while (c != 'X');
28 |
29 |     printf("\nEnd of program.\n");
30 |     return(0);
31 | }

```

**ex044.c** – Exemplo de leitura de inteiros.

```

1 | #include <stdio.h>
2 |
3 | int main()
4 | {
5 |     int valin;
6 |
7 |     printf("Input a number from 0 to 2147483647, stop with 100.\n");
8 |
9 |     do {
10 |         scanf("%d",&valin); /* read a single integer value in */
11 |         printf("The value is %d\n",valin);
12 |     } while (valin != 100);
13 |
14 |     printf("End of program\n");
15 |     return(0);
16 | }

```

**ex045.c** – Exemplo de leitura de *strings*.

```

1 | #include <stdio.h>
2 |
3 | int main()
4 | {
5 |     char big[25];
6 |
7 |     printf("Input a character string, up to 25 characters.\n");
8 |     printf("An X in column 1 causes the program to stop.\n");
9 |
10 |    do {
11 |        scanf("%s",big);
12 |        printf("The string is -> %s\n",big);
13 |    } while (big[0] != 'X');

```

```
14
15     printf("End of program.\n");
16     return(0);
17 }
```

**ex046.c** – Exemplo de formatação para uma *string*.

```
1  #include <stdio.h>
2  int main()
3  {
4  int numbers[5], result[5], index;
5  char line[80];
6
7  numbers[0] = 74;
8  numbers[1] = 18;
9  numbers[2] = 33;
10 numbers[3] = 30;
11 numbers[4] = 97;
12
13 sprintf(line,"%d %d      %d %d %d\n",numbers[0],numbers[1],
14         numbers[2],numbers[3],numbers[4]);
15
16 printf("%s",line);
17
18 sscanf(line,"%d %d %d %d      %d",&result[4],&result[3],
19        (result+2),(result+1),result);
20
21
22 for (index = 0;index < 5;index++)
23     printf("The final result is %d\n",result[index]);
24 return(0);
25 }
```

**ex047.c** – Exemplo de uso do `stderr`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6  int index;
7
8  for (index = 0;index < 6;index++) {
9      printf("This line goes to the standard output.\n");
10     fprintf(stderr,"This line goes to the error device.\n");
11 }
12 exit(4); /* This can be tested with the
13          unix command line
14          ./ex047 >lixo ; echo $? */
15 }
```

## Exercícios

1. Escreva um programa que em ciclo leia caracteres e os mostre na forma *normal* e também mostre o seu valor decimal. Use o carácter f como sinal para finalizar o ciclo.

## A.9. Acesso a ficheiros

**ex048.c** – Exemplo de como escrever dados para um ficheiro.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     FILE *fp;
7     char stuff[25];
8     int index;
9
10    fp = fopen("tenlines.txt","w"); /* open for writing */
11    strcpy(stuff,"This is an example line.");
12
13    for (index = 1;index <= 10;index++)
14        fprintf(fp,"%s Line number %d\n",stuff,index);
15
16    fclose(fp); /* close the file before ending program */
17    return(0);
18 }
```

**ex049.c** – Exemplo de como escrever dados para um ficheiro, um carácter de cada vez.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     FILE *point;
7     char others[35];
8     int indexer,count;
9
10    strcpy(others,"Additional lines.");
11    point = fopen("tenlines.txt","a"); /* open for appending */
12
13    for (count = 1;count <= 10;count++) {
14        for (indexer = 0;others[indexer];indexer++)
15            putchar(others[indexer],point); /* output a single character */
16        putchar('\n',point); /* output a linefeed */
17    }
18    fclose(point);
19    return(0);
20 }
```

**ex050.c** – Exemplo de como ler caracteres de um ficheiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     FILE *funny;
7     char c;
8
9     funny = fopen("tenlines.txt","r");
10
11     if (funny == NULL) {
12         printf("File doesn't exist\n");
13         exit(-1); }
14     else {
15         do {
16             c = getc(funny);    /* get one character from the file */
17             putchar(c);        /* display it on the monitor */
18         } while (c != EOF);    /* repeat until EOF (end of file) */
19     }
20     fclose(funny);
21     return(0);
22 }
```

**ex051.c** – Exemplo de como ler texto de um ficheiro.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     FILE *fp1;
6     char oneword[100];
7     char c;
8
9     fp1 = fopen("tenlines.txt","r");
10
11     do {
12         c = fscanf(fp1,"%s",oneword); /* got one word from the file */
13         printf("%s\n",oneword);      /* display it on the monitor */
14     } while (c != EOF);              /* repeat until EOF */
15
16     fclose(fp1);
17     return(0);
18 }
```

**ex052.c** – Exemplo de como ler texto de um ficheiro (corrigido).

```
1 #include <stdio.h>
2
3 int main()
```

```

4 {
5 FILE *fp1;
6 char oneword[100];
7 char c;
8
9     fp1 = fopen("tenlines.txt","r");
10
11     do {
12         c = fscanf(fp1,"%s",oneword); /* got one word from the file */
13         if (c != EOF)
14             printf("%s\n",oneword);    /* display it on the monitor */
15     } while (c != EOF);                /* repeat until EOF      */
16
17     fclose(fp1);
18     return(0);
19 }

```

**ex053.c** – Exemplo de como ler um ficheiro linha a linha.

```

1 #include "stdio.h"
2
3 int main()
4 {
5     FILE *fp1;
6     char oneword[100];
7     char *c;
8
9     fp1 = fopen("tenlines.txt","r");
10
11     do {
12         c = fgets(oneword,100,fp1); /* get one line from the file */
13         if (c != NULL)
14             printf("%s",oneword);    /* display it on the monitor */
15     } while (c != NULL);            /* repeat until NULL      */
16
17     fclose(fp1);
18     return(0);
19 }

```

**ex054.c** – Exemplo de como ler de qualquer ficheiro.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     FILE *fp1;
6     char oneword[100],filename[25];
7     char *c;
8
9     printf("Enter filename -> ");
10    scanf("%s",filename);      /* read the desired filename */
11    fp1 = fopen(filename,"r");
12
13    do {
14        c = fgets(oneword,100,fp1); /* get one line from the file */
15        if (c != NULL)
16            printf("%s",oneword);    /* display it on the monitor */
17    } while (c != NULL);            /* repeat until NULL */
18
19    fclose(fp1);
20    return(0);
21 }
22
```

### Exercícios

1. Faça um programa que peça ao utilizador o nome de um ficheiro e mostre esse ficheiro no écran, linha a linha, sendo cada linha precedida pelo seu número.
2. Modifique o programa ex054.c para ver se o ficheiro a ler existe ou não, de uma forma similar ao que está feito em ex050.c.

## A.10. Estruturas, uniões e campos de bits

**ex055.c** – Exemplo de estruturas.

```
1 #include <stdio.h>
2 int main()
3 {
4
5     struct {
6         char initial;    /* last name initial */
7         int age;         /* childs age */
8         int grade;       /* childs grade in school */
9     } boy,girl;
10
11     boy.initial = 'R';
12     boy.age = 15;
13     boy.grade = 75;
14
15
```



```

16  girl.age = boy.age - 1;  /* she is one year younger */
17  girl.grade = 82;
18  girl.initial = 'H';
19
20  printf("%c is %d years old and got a grade of %d\n",
21         girl.initial, girl.age, girl.grade);
22
23  printf("%c is %d years old and got a grade of %d\n",
24         boy.initial, boy.age, boy.grade);
25  return(0);
26  }

```

**ex056.c** – Exemplo de um vector de estruturas.

```

1  #include <stdio.h>
2  int main()
3  {
4  struct {
5      char initial;
6      int age;
7      int grade;
8  } kids[12];
9
10 int index;
11
12 for (index = 0; index < 12; index++) {
13     kids[index].initial = 'A' + index;
14     kids[index].age = 16;
15     kids[index].grade = 84;
16 }
17
18 kids[3].age = kids[5].age = 17;
19 kids[2].grade = kids[6].grade = 92;
20 kids[4].grade = 57;
21
22 kids[10] = kids[4];          /* Structure assignment */
23
24 for (index = 0; index < 12; index++)
25     printf("%c is %d years old and got a grade of %d\n",
26            kids[index].initial, kids[index].age,
27            kids[index].grade);
28 return(0);
29 }

```

**ex057.c** – Exemplo de como usar apontadores com estruturas.

```
1 #include <stdio.h>
2 int main()
3 {
4     struct {
5         char initial;
6         int age;
7         int grade;
8     } kids[12], *point, extra;
9
10    int index;
11
12    for (index = 0; index < 12; index++) {
13        point = kids + index;
14        point->initial = 'A' + index;
15        point->age = 16;
16        point->grade = 84;
17    }
18    kids[3].age = kids[5].age = 17;
19    kids[2].grade = kids[6].grade = 92;
20    kids[4].grade = 57;
21
22    for (index = 0; index < 12; index++) {
23        point = kids + index;
24        printf("%c is %d years old and got a grade of %d\n",
25              (*point).initial, kids[index].age,
26              point->grade);
27    }
28    extra = kids[2];           /* Structure assignment */
29    extra = *point;           /* Structure assignment */
30    return(0);
31 }
```

**ex058.c** – Exemplo de como colocar estruturas dentro de estruturas.

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     struct person {
6         char name[25];
7         int age;
8         char status;           /* M = married, S = single */
9     } ;
10
11    struct alldat {
12        int grade;
13        struct person descrip;
14        char lunch[25];
15    } student[53];
```

```

16 |
17 | struct alldat teacher,sub;
18 |
19 | teacher.grade = 94;
20 | teacher.descrip.age = 34;
21 | teacher.descrip.status = 'M';
22 | strcpy(teacher.descrip.name,"Mary Smith");
23 | strcpy(teacher.lunch,"Baloney sandwich");
24 |
25 | sub.descrip.age = 87;
26 | sub.descrip.status = 'M';
27 | strcpy(sub.descrip.name,"Old Lady Brown");
28 | sub.grade = 73;
29 | strcpy(sub.lunch,"Yogurt and toast");
30 |
31 | student[1].descrip.age = 15;
32 | student[1].descrip.status = 'S';
33 | strcpy(student[1].descrip.name,"Billy Boston");
34 | strcpy(student[1].lunch,"Peanut Butter");
35 | student[1].grade = 77;
36 |
37 | student[7].descrip.age = 14;
38 | student[12].grade = 87;
39 | return(0);
40 | }

```

#### ex059.c – Exemplo de uniões.

```

1 | #include <stdio.h>
2 | int main()
3 | {
4 |     union {
5 |         short int value;      /* This is the first part of the union */
6 |         struct {
7 |             char first;      /* These two values are the second */
8 |             char second;
9 |         } half;
10 |    } number;
11 |
12 |    long index;
13 |
14 |    for (index = 12; index < 30000; index += 1000) {
15 |        number.value = index;
16 |        printf("%8x %6x %6x\n",number.value, number.half.first,
17 |               number.half.second);
18 |    }
19 |    return(0);
20 | }

```

**ex060.c** – Outro exemplo de uniões.

```
1 #include <stdio.h>
2
3 #define AUTO 1
4 #define BOAT 2
5 #define PLANE 3
6 #define SHIP 4
7
8 int main()
9 {
10 struct automobile { /* structure for an automobile */
11     int tires;
12     int fenders;
13     int doors;
14 };
15
16 typedef struct { /* structure for a boat or ship */
17     int displacement;
18     char length;
19 } BOATDEF;
20
21 struct {
22     char vehicle; /* what type of vehicle? */
23     int weight; /* gross weight of vehicle */
24     union { /* type-dependent data */
25         struct automobile car; /* part 1 of the union */
26         BOATDEF boat; /* part 2 of the union */
27         struct {
28             char engines;
29             int wingspan;
30         } airplane; /* part 3 of the union */
31         BOATDEF ship; /* part 4 of the union */
32     } vehicle_type;
33     int value; /* value of vehicle in dollars */
34     char owner[32]; /* owners name */
35 } ford, sun_fish, piper_cub; /* three variable structures */
36
37 /* define a few of the fields as an illustration */
38
39 ford.vehicle = AUTO;
40 ford.weight = 2742; /* with a full gas tank */
41 ford.vehicle_type.car.tires = 5; /* including the spare */
42 ford.vehicle_type.car.doors = 2;
43
44 sun_fish.value = 3742; /* trailer not included */
45 sun_fish.vehicle_type.boat.length = 20;
46
47 piper_cub.vehicle = PLANE;
48 piper_cub.vehicle_type.airplane.wingspan = 27;
49
50
```

```

51     if (ford.vehicle == AUTO) /* which it is in this case */
52         printf("The ford has %d tires.\n",ford.vehicle_type.car.tires);
53
54     if (piper_cub.vehicle == AUTO) /* which it is not in this case */
55         printf("The plane has %d tires.\n",piper_cub.vehicle_type.
56             car.tires);
57     return(0);
58 }

```

#### ex061.c – Exemplo de campos de bits.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      union {
6          short int index;
7          struct {
8              unsigned int t : 11; /* fill the rest - not needed on Intel */
9              unsigned int x : 1;
10             unsigned int y : 2;
11             unsigned int z : 2;
12         } bits;
13     } number;
14
15     for (number.index = 0;number.index < 20;number.index++) {
16         printf("index = %3d, bits = %3d%3d%3d\n",number.index,
17             number.bits.z,number.bits.y,number.bits.x);
18     }
19     return(0);
20 }

```

#### Exercícios

- Defina uma estrutura, que contenha um campo do tipo *string* chamado nome, e dois do tipo inteiro chamados assentos e rodas. Use esse tipo de dados para definir um vector de cerca de 6 itens. Preencha os campos com dados e imprima-os da forma seguinte:
  - Carro tem 4 assentos e 4 rodas.
  - Bicicleta tem 1 assentos e 2 rodas.
  - Triciclo tem 1 assentos e 3 rodas.
  - Sidecar tem 3 assentos e 3 rodas.
  - Moto tem 2 assentos e 2 rodas.
  - Uniciclo tem 1 assentos e 1 rodas.
  - Motoquatro tem 2 assentos e 4 rodas.
- Re-escreva o exercício anterior de modo a usar um apontador para imprimir os dados.

## A.11. Memória dinâmica

**ex062.c** – Exemplo de alocação dinâmica da memória.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main()
5 {
6     struct animal {
7         char name[25];
8         char breed[25];
9         int age;
10    } *pet1, *pet2, *pet3;
11
12    pet1 = (struct animal *)malloc(sizeof(struct animal));
13    strcpy(pet1->name,"General");
14    strcpy(pet1->breed,"Mixed Breed");
15    pet1->age = 1;
16
17    pet2 = pet1;    /* pet2 now points to the above data structure */
18
19    pet1 = (struct animal *)malloc(sizeof(struct animal));
20    strcpy(pet1->name,"Frank");
21    strcpy(pet1->breed,"Labrador Retriever");
22    pet1->age = 3;
23
24    pet3 = (struct animal *)malloc(sizeof(struct animal));
25    strcpy(pet3->name,"Krystal");
26    strcpy(pet3->breed,"German Shepherd");
27    pet3->age = 4;
28
29    /* now print out the data described above */
30
31    printf("%s is a %s, and is %d years old.\n", pet1->name,
32          pet1->breed, pet1->age);
33
34    printf("%s is a %s, and is %d years old.\n", pet2->name,
35          pet2->breed, pet2->age);
36
37    printf("%s is a %s, and is %d years old.\n", pet3->name,
38          pet3->breed, pet3->age);
39
40    pet1 = pet3;    /* pet1 now points to the same structure that
41                   pet3 points to */
42    free(pet3);    /* this frees up one structure */
43    free(pet2);    /* this frees up one more structure */
44    /* free(pet1);  this cannot be done, see explanation in text */
45    return(0);
46 }
```

**ex063.c** – Outro exemplo de alocação dinâmica da memória.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      struct animal {
8          char name[25];
9          char breed[25];
10         int age;
11     } *pet[12], *point;    /* this defines 13 pointers, no variables */
12     int index;
13
14         /* first, fill the dynamic structures with nonsense */
15     for (index = 0; index < 12; index++) {
16         pet[index] = (struct animal *)malloc(sizeof(struct animal));
17         strcpy(pet[index]->name, "General");
18         strcpy(pet[index]->breed, "Mixed Breed");
19         pet[index]->age = 4;
20     }
21
22     pet[4]->age = 12;        /* these lines are simply to      */
23     pet[5]->age = 15;        /*      put some nonsense data into */
24     pet[6]->age = 10;        /*      a few of the fields.    */
25
26         /* now print out the data described above */
27
28     for (index = 0; index < 12; index++) {
29         point = pet[index];
30         printf("%s is a %s, and is %d years old.\n", point->name,
31             point->breed, point->age);
32     }
33
34         /* good programming practice dictates that we free up the */
35         /* dynamically allocated space before we quit.             */
36
37     for (index = 0; index < 12; index++)
38         free(pet[index]);
39     return(0);
40 }

```

**ex064.c** – Exemplo de como fazer uma lista ligada.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #define RECORDS 6
5
6 int main()
7 {
8     struct animal {
9         char name[25];          /* The animals name */
10        char breed[25];         /* The type of animal */
11        int age;                /* The animals age */
12        struct animal *next; /* a pointer to another record of this type */
13    } *point, *start, *prior; /* this defines 3 pointers, no variables */
14    int index;
15
16    /* the first record is always a special case */
17    start = (struct animal *)malloc(sizeof(struct animal));
18    strcpy(start->name,"General");
19    strcpy(start->breed,"Mixed Breed");
20    start->age = 4;
21    start->next = NULL;
22    prior = start;
23    /* a loop can be used to fill in the rest once it is started */
24    for (index = 0; index < RECORDS; index++) {
25        point = (struct animal *)malloc(sizeof(struct animal));
26        strcpy(point->name,"Frank");
27        strcpy(point->breed,"Labrador Retriever");
28        point->age = 3;
29        prior->next = point; /* point last "next" to this record */
30        point->next = NULL; /* point this "next" to NULL */
31        prior = point;      /* this is now the prior record */
32    }
33    /* now print out the data described above */
34    point = start;
35    do {
36        prior = point->next;
37        printf("%s is a %s, and is %d years old.\n", point->name,
38            point->breed, point->age);
39        point = point->next;
40    } while (prior != NULL);
41    /* good programming practice dictates that we free up the */
42    /* dynamically allocated space before we quit. */
43    point = start;          /* first block of group */
44    do {
45        prior = point->next; /* next block of data */
46        free(point);        /* free present block */
47        point = prior;       /* point to next */
48    } while (prior != NULL); /* quit when next is NULL */
49    return(0);
50 }
```



**Exercícios**

1. Re-escreva o exemplo ex055.c de modo a alocar dinamicamente as estruturas de dados.
2. Re-escreva o exemplo ex056.c de modo a alocar dinamicamente o vector de estruturas.

**A.12. Operações diversas**

**ex065.c** – Exemplo de processamento de maiúsculas e minúsculas.

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  void mix_up_the_chars(char line[]);
5
6  int main()
7  {
8      FILE *fp;
9      char line[80], filename[24];
10     char *c;
11     printf("Enter filename -> ");
12     scanf("%s",filename);
13     fp = fopen(filename,"r");
14
15     do {
16         c = fgets(line,80,fp);    /* get a line of text */
17         if (c != NULL) {
18             mix_up_the_chars(line);
19         }
20     } while (c != NULL);
21
22     fclose(fp);
23     return(0);
24 }
25
26 void mix_up_the_chars(char line[])
27     /* this function turns all upper case characters into lower case,
28     and all lower case to upper case. It ignores all other characters. */
29 {
30     int index;
31
32     for (index = 0; line[index] != 0; index++) {
33         if (isupper(line[index]))    /* 1 if upper case */
34             line[index] = tolower(line[index]);
35         else {
36             if (islower(line[index])) /* 1 if lower case */
37                 line[index] = toupper(line[index]);
38         }
39     }
40     printf("%s",line);
41 }
```

**ex066.c** – Exemplo de classificação de caracteres.

```
1
2 #include <stdio.h>
3 #include <ctype.h>
4
5 void count_the_data(char line[]);
6
7 int main()
8 {
9     FILE *fp;
10    char line[80], filename[24];
11    char *c;
12
13    printf("Enter filename -> ");
14    scanf("%s",filename);
15    fp = fopen(filename,"r");
16
17    do {
18        c = fgets(line,80,fp);    /* get a line of text */
19        if (c != NULL) {
20            count_the_data(line);
21        }
22    } while (c != NULL);
23
24    fclose(fp);
25    return(0);
26 }
27
28 void count_the_data(char line[])
29 {
30     int whites, chars, digits;
31     int index;
32
33     whites = chars = digits = 0;
34
35     for (index = 0; line[index] != 0; index++) {
36         if (isalpha(line[index]))    /* 1 if line[] is alphabetic */
37             chars++;
38         if (isdigit(line[index]))    /* 1 if line[] is a digit */
39             digits++;
40         if (isspace(line[index]))    /* 1 if line[] is blank, tab, */
41             whites++;                /* or newline */
42     }    /* end of counting loop */
43
44     printf("%3d%3d%3d %s",whites,chars,digits,line);
45 }
```

**ex067.c** – Exemplos de operações bit a bit.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char mask;
6     unsigned char number[6];
7     char and,or,xor,inv;
8     int index;
9
10    number[0] = 0X00;
11    number[1] = 0X11;
12    number[2] = 0X22;
13    number[3] = 0X44;
14    number[4] = 0X88;
15    number[5] = 0XFF;
16
17    printf(" nmbr mask  and   or   xor   inv\n");
18    mask = 0X0F;
19    for (index = 0;index <= 5;index++) {
20        and = mask & number[index];
21        or = mask | number[index];
22        xor = mask ^ number[index];
23        inv = ~number[index];
24        printf("%8x %8x %8x %8x %8x %8x\n",number[index],
25            mask,and,or,xor,inv);
26    }
27
28    printf("\n");
29    mask = 0X22;
30    for (index = 0;index <= 5;index++) {
31        and = mask & number[index];
32        or = mask | number[index];
33        xor = mask ^ number[index];
34        inv = ~number[index];
35        printf("%8x %8x %8x %8x %8x %8x\n",number[index],
36            mask,and,or,xor,inv);
37    }
38    return(0);
39 }
```

**ex068.c** – Exemplos de deslocamentos (*shifts*).

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int small, big, index, count;
6     printf("          shift left      shift right\n\n");
7     small = 1;
```

```
8   big = 0x4000;
9   for(index = 0;index < 17;index++) {
10      printf("%8d %8x %8d %8x\n",small,small,big,big);
11      small = small << 1;
12      big = big >> 1;
13   }
14
15   printf("\n");
16   count = 2;
17   small = 1;
18   big = 0x4000;
19   for(index = 0;index < 9;index++) {
20      printf("%8d %8x %8d %8x\n",small,small,big,big);
21      small = small << count;
22      big = big >> count;
23   }
24   return (0);
25 }
```

### A.13. Erros mais comuns

- Confundir maiúsculas e minúsculas – Em linguagem C e em Unix as maiúsculas e minúsculas são diferentes. Por exemplo, um ficheiro cujo nome termine por `.c` corresponde a um ficheiro em linguagem C enquanto que um ficheiro cujo nome termine por `.C` corresponde a um programa em C++.
- A função `main()` é do tipo `int` e deve devolver sempre um valor. Este valor deve ser 0 (zero) se não existirem erros no funcionamento do programa, e diferente de zero se por acaso tiver ocorrido algum erro.