PROGRAMAÇÃO

ESTRUTURAS DE DADOS

E ALGORITMOS

EM C

Professor Doutor António Manuel Adrego da Rocha
Professor Doutor António Rui Oliveira e Silva Borges

Departamento de Electrónica e Telecomunicações
Universidade de Aveiro

Prefácio

Este texto serve de suporte à disciplina de Programação II, cujo objectivo é o de fornecer uma familiarização com o ambiente de programação fornecido pelo Unix, na sua variante mais popularizada Linux e o domínio da linguagem C, na sua norma ANSI, para o desenvolvimento de programas de média e elevada complexidade.

Começamos por apresentar os aspectos essenciais da linguagem C em dois capítulos. Depois introduzirmos as construções mais complexas da linguagem de forma gradual, à medida que são necessárias à construção de estruturas de dados mais complexas, bem como para a optimização e generalização de algoritmos. Os aspectos fundamentais apresentados no texto são os seguintes:

- A familiarização progressiva com a linguagem de programação C e com as suas bibliotecas.
- A apresentação de algoritmos recursivos e sua comparação com os algoritmos iterativos equivalentes.
- A introdução da metodologia de decomposição modular das soluções, ou seja, o paradigma da programação modular.
- O estudo da organização da Memória de Acesso Aleatório (*RAM*), nas suas implementações estática e semiestática, e, de um conjunto significativo de algoritmos de pesquisa e de ordenação.
- O estudo da organização de memórias mais complexas que a Memória de Acesso Aleatório, como por exemplo, a Memória Fila de Espera (*FIFO*), a Memoria Pilha (*Stack*) e a Memória Associativa (*CAM*), nas suas implementações estática, semiestática e dinâmica, e, dos algoritmos associados para pesquisa, introdução e retirada de informação.

Assume-se que os alunos frequentaram a disciplina de Programação I, e portanto, já estão familiarizados com a metodologia de decomposição hierárquica das soluções, estabelecendo dependências de informação e no encapsulamento da informação com a criação de novas instruções no âmbito da linguagem Pascal, ou seja, com o paradigma da programação procedimental. Bem como, com a criação de estruturas de dados estáticas com alguma complexidade que modelam correctamente a resolução dos problemas. Pelo que, a apresentação da linguagem C é feita por comparação com a linguagem Pascal.

Pretende-se ainda, que os alunos se familiarizem com a terminologia informática apresentada nos textos de referência da área das Ciências da Computação, pelo que, se tenha optado pela apresentação sistemática, em itálico e entre parêntesis, dos nomes dos algoritmos e das estruturas de dados em inglês.

Capítulo 1

INTRODUÇÃO AO C

Sumário

Este capítulo é dedicado à introdução das primeiras noções sobre a gramática da linguagem C. Começamos por apresentar a estrutura de um programa e os seus elementos básicos. Explicamos os tipos de dados básicos existentes, a definição de constantes e de variáveis. Apresentamos os vários tipos de expressões e operadores existentes e a instrução de atribuição, que é a instrução básica de uma linguagem imperativa. Apresentamos de seguida as estruturas de controlo, que permitem alterar o fluxo da sequência das instruções. Apresentamos ainda as instruções de leitura de dados do teclado **scanf** e de escrita de dados no monitor **printf**. Finalmente, apresentamos as bibliotecas que contêm as funções mais usuais e que estendem a operacionalidade da linguagem.

1.1 Introdução

Em 1972, Dennis M. Ritchie desenvolveu a linguagem C, nos Laboratórios Bell da companhia AT & T, que é a principal empresa de telecomunicações dos Estados Unidos da América, como uma linguagem de programação concebida para a escrita de sistemas operativos, aquilo que se designa por Programação de Sistemas. Como a linguagem C era tão flexível e permitia que os compiladores produzissem código em linguagem máquina muito eficiente, em 1973, Dennis M. Ritchie e Ken Thompson reescreveram quase totalmente o sistema operativo Unix em C. Devido a esta ligação íntima, à medida que o Unix se tornou popular no meio académico, também a linguagem C se tornou a linguagem preferida para o desenvolvimento de aplicações científicas. Pelo que, apesar de ter sido concebida para a escrita de sistemas operativos, a linguagem C é hoje encarada como uma linguagem de uso geral.

A principal característica da linguagem C é que combina as vantagens de uma linguagem de alto nível descendente do AlGOL 68, com a eficiência da linguagem *assembly*, uma vez que permite a execução de operações aritméticas sobre ponteiros e operações sobre palavras binárias. A linguagem C também tem uma sintaxe muito compacta e permite que operadores de tipos diferentes possam ser combinados livremente.

Esta liberdade e poder da linguagem C, permite aos programadores experientes escreverem código compacto e eficiente que dificilmente poderiam ser escritos noutras linguagens de programação. Mas, como é fracamente estruturada em termos semânticos, também permite que construções sem sentido aparente, escritas por programadores inexperientes, sejam aceites pelo compilador como válidas. O facto da linguagem C ser muito poderosa, exige portanto, do programador muita disciplina e rigor na utilização das construções da linguagem, para que o código escrito seja legível e facilmente alterável.

Apesar da linguagem C ter sido desenvolvida no princípio da década de 1970, a norma ANSI (American National Standards Institute) foi apenas aprovada em 1989 (norma ISO/IEC 9899-1990).

1.2 A estrutura de um programa em C

Ao contrário do que se passa em Pascal, em que um programa apresenta uma organização hierárquica que reflecte directamente o algoritmo que lhe deu origem, na linguagem C, um programa é organizado horizontalmente como um agrupamento de variáveis e funções colocadas todas ao mesmo nível, uma estrutura conhecida pelo nome de **mar de funções**.

Neste contexto, a diferenciação entre o programa principal e os diversos subprogramas associados é feita pelo facto de existir uma função particular, de nome **main**, que é sempre invocada em primeiro lugar aquando da execução do programa. Assim, a função **main** desempenha na prática o papel do programa principal do Pascal. Segundo a norma ANSI, a função **main** é uma função de tipo inteiro, em que o valor devolvido serve para informar sobre o estado de execução do programa.

De facto, a noção de devolver um valor associado ao estado de execução de um programa corresponde à filosofia subjacente à arquitectura de comandos do Unix, em que a linguagem de comandos (*shell*) é, no fundo, uma verdadeira linguagem de programação, que permite construir comandos mais complexos por combinação de comandos mais simples (*shell scripts*), usando um conjunto de estruturas de controlo muito semelhantes aos encontrados na linguagem C ou Pascal. Neste contexto, os comandos mais simples são programas, de cujo sucesso de execução vai eventualmente depender a continuação das operações.

Ora, como esta é uma área que não será explorada no âmbito desta disciplina, em muitos programas, sobretudo naqueles que serão desenvolvidos nesta disciplina, não se coloca a questão de devolver um valor. Pelo que, para evitar a mensagem de aviso do compilador, recomenda-se terminar o **main** com a instrução **return** 0. Um programa em C tem a estrutura apresentada na Figura 1.1.

```
alusão a funções e definições externas

alusão a funções e definições locais

int main ( void )
{
  declaração de variáveis

  alusão a funções

  sequência de instruções

  return 0;
}

definição de funções locais
```

Figura 1.1 - Estrutura de um programa em C.

Vamos analisar as diversas partes de um programa em C através do exemplo do programa de conversão de distâncias de milhas para quilómetros apresentado na Figura 1.2. O ficheiro fonte que contém o programa começa por mencionar as funções e estruturas de dados externas necessárias à execução do programa, que estão implementadas noutros ficheiros providenciados pela linguagem C, as chamadas bibliotecas da linguagem, ou que em alternativa são desenvolvidos pelo utilizador. A linguagem C foi concebida tendo em mente facilitar a construção descentralizada de aplicações, através do fraccionamento do código de um programa por diferentes ficheiros fonte. Por isso, é necessário e inevitável, mesmo em programas muito simples, recorrer à alusão a funções e definições feitas externamente, ou seja, noutros ficheiros. Para tornar mais rigorosa esta referência, foi criado o conceito de **ficheiro de interface**, onde todas as alusões e definições associadas a um dado tipo de funcionalidade são colocadas. Estes ficheiros de interface distinguem-se dos ficheiros fonte, propriamente ditos, por terem a terminação **.h** em vez de **.c**.

A norma ANSI fornece um conjunto muito variado de ficheiros de interface que descrevem as diferentes funcionalidades fornecidas pelas bibliotecas de execução ANSI. Em Unix, por defeito, todos eles estão armazenados no directório */usr/include*. Daí, não ser necessário referenciar este caminho de um modo explícito. Quando o ficheiro em causa está neste directório, basta colocar o seu nome entre os símbolos < e >. Em todos os outros casos, a especificação do caminho deve ser incluída no nome do ficheiro e o conjunto ser colocado entre aspas duplas.

A inclusão de ficheiros de interface num dado ficheiro fonte é feita usando a directiva do pré-processador #***include*** numa das suas duas variantes:

- No caso do ficheiro de interface pertencer à linguagem C, então ele está armazenado no directório por defeito e usa-se a directiva **#include <**nome do ficheiro de interface**>**.
- No caso do ficheiro de interface ter sido criado pelo utilizador e não estar armazenado no directório por defeito, usa-se a directiva **#include "**nome do ficheiro de interface**"**.

No mínimo, todos os ficheiros que contenham código que faça acesso aos dispositivos convencionais de entrada e de saída têm que incluir o ficheiro de interface *stdio.h*, que descreve as funções e que contém as definições associadas com o acesso aos dispositivos de entrada e de saída e aos ficheiros. Normalmente, o dispositivo de entrada é o teclado e o dispositivo de saída é o monitor. Portanto, qualquer programa interactivo tem pelo menos a alusão a este ficheiro de interface, tal como se apresenta na Figura 1.2.

A seguir às definições de objectos externos segue-se a alusão às funções locais que vão ser usadas na função **main**, bem como a definição de estruturas de dados e constantes locais. Locais para a aplicação, mas que para o ficheiro, se comportam como definições globais. Repare que a estruturação do programa, é muito diferente do Pascal, sendo que as funções são primeiramente aludidas ou referidas, para se tornarem visíveis em todo o ficheiro e só depois da função **main** é que são definidas. Neste exemplo, define-se apenas, através da directiva **#define**, um identificador constante MIL_QUI, que representa o factor de conversão de milhas para quilómetros. Ele é visível para todo o código do ficheiro, ou seja, é uma constante global. A função **main** é implementada com instruções simples, e com recurso apenas às funções de entrada e de saída de dados da biblioteca ***stdio***.

```
/* Programa de conversão de distâncias de milhas para quilómetros */
                /* Instruções para o pré-processador */
#include <stdio.h>  /* interface com a biblioteca de entrada/saída */
#define  MIL_QUI  1.609                      /* factor de conversão */
          /* Instruções em linguagem C propriamente ditas */
int main ( void )
{
  double MILHAS,                    /* distância expressa em milhas */
         QUILOMETROS;          /* distância expressa em quilómetros */
  do  /* Leitura com validação de uma distância expressa em milhas */
  {
     printf ("Distância em milhas? ");
     scanf ("%lf", &MILHAS);
  } while (MILHAS < 0.0);
  QUILOMETROS = MIL_QUI * MILHAS;         /* Conversão da distância */
                 /* Impressão da distância expressa em quilómetros */
  printf ("Distância em quilómetros é %8.3f\n", QUILOMETROS);
  return 0;
}
```

Figura 1.2 - Programa da conversão de distâncias.

Vamos agora analisar com detalhe na Figura 1.3 a definição da função **main**, que tal como qualquer outra função na linguagem C, supõe a especificação do seu **cabeçalho** e do seu **corpo**. No cabeçalho, indica-se o tipo do valor devolvido, que como já foi referido anteriormente é sempre do tipo inteiro, o nome, e entre parênteses curvos, a lista de

parâmetros de comunicação. Neste caso a função não comunica directamente com o exterior, pelo que, não existe lista de parâmetros de comunicação. Quando tal acontece, utiliza-se o identificador **void**. O corpo da função é delimitado pelos separadores **{** e **}**, correspondentes, respectivamente, aos separadores **begin** e **end** do Pascal, e contém a declaração das variáveis locais, a alusão a funções usadas na sequência de instruções e a sequência de instruções propriamente dita. Constitui aquilo que em linguagem C se designa por um **bloco**.

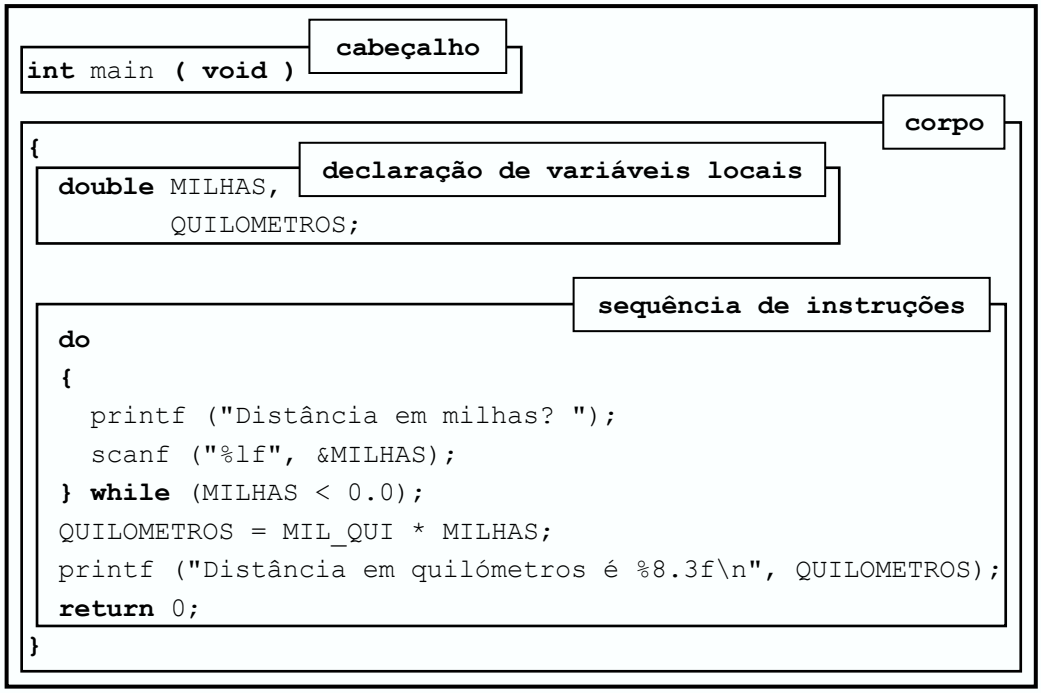

Figura 1.3 - A função main.

1.3 Elementos básicos da linguagem C

Os **identificadores** são nomes que são usados para designar os diferentes objectos existentes no programa, como por exemplo, o nome do programa, os nomes das funções, os nomes das constantes, tipos de dados e variáveis. Os identificadores obedecem à regra de produção apresentada na Figura 1.4.

identificador ::= letra do alfabeto | carácter *underscore* |
identificador letra do alfabeto |
identificador carácter *underscore* |
identificador algarismo decimal

Figura 1.4 - Definição formal de um identificador.

Ou seja, são formados por uma sequência de caracteres alfanuméricos e o carácter *underscore*, em que o primeiro carácter é obrigatoriamente uma letra do alfabeto ou o carácter *underscore*. Embora seja possível começar um identificador pelo carácter *underscore*, tal deve ser evitado. Este tipo de notação é, em princípio, reservado para o compilador. Não há limite para o comprimento de um identificador. Na prática, esse limite é imposto pelo compilador. A norma ANSI exige um comprimento mínimo de 31 e 6 caracteres, respectivamente, para os identificadores internos e externos.

Na linguagem C, os alfabetos maiúsculo e minúsculo são distintos, ou seja, a linguagem é sensível ao tipo de letra (*case sensitive*). Assim sendo, o identificador conv_dist é diferente do identificador CONV_DIST. Embora não seja obrigatório, é costume designar todos os identificadores da responsabilidade do programador com caracteres maiúsculos de maneira a distingui-los dos identificadores da linguagem C, que têm de ser escritos obrigatoriamente com caracteres minúsculos. Em alternativa, há programadores que gostam de usar um carácter maiúsculo no primeiro carácter de cada palavra que compõe o identificador e os restantes caracteres minúsculos. Nesse caso, devem designar pelo menos os nomes das constantes em caracteres maiúsculos. Mas, o importante é que cada programador defina o seu próprio estilo, e que exista uma certa coerência nas regras que adopte.

As palavras reservadas da linguagem C são: **auto**; **break**; **case**; **char**; **const**; **continue**; **default**; **do**; **double**; **else**; **enum**; **extern**; **float**; **for**; **goto**; **if**; **int**; **long**; **register**; **return**; **short**; **signed**; **sizeof**; **static**; **struct**; **switch**; **typedef**; **union**; **unsigned**; **void**; **volatile**; e **while**. As palavras reservadas aparecem a cheio ao longo do texto e no código apresentado.

Para melhorar a legibilidade do programa, devem ser introduzidos comentários relevantes, que expliquem o significado dos diferentes objectos, ou que operação é efectuada por grupos bem definidos de instruções. Um comentário é uma qualquer sequência de símbolos inserida entre /* e */ e que não contenha */. Isto significa que não se pode nunca encapsular comentários. A Figura 1.5 apresenta a definição formal do comentário.

comentário ::= /* qualquer sequência de símbolos que não contenha '*/' */

Figura 1.5 - Definição formal do comentário.

O uso adequado de comentários melhora extraordinariamente a legibilidade e a compreensão de um segmento de código. Assim, devem introduzir-se comentários, pelo menos, nas situações seguintes:

- Em título, para explicar o que faz o segmento de código e descrever, caso exista, o mecanismo de comunicação associado.
- Sempre que se declarem constantes ou variáveis, para explicar o seu significado, a menos que este seja trivial.
- A encabeçar as porções de código correspondentes à decomposição algorítmica que lhe deu origem.

1.4 Representação da informação

Em Pascal, os tipos de dados predefinidos estão directamente relacionados com o tipo de informação neles armazenado. Assim, para informação numérica, existem os tipos **inteiro** (***integer***) para a representação exacta e **real** (***real***) para a representação aproximada; para quantidades lógicas existe o tipo **booleano** (***boolean***); e para a representação de símbolos gráficos existe o tipo **carácter** (***char***).

Em linguagem C, pelo contrário, os tipos de dados predefinidos reflectem apenas o formato de armazenamento. São sempre tipos numéricos, embora em alguns casos possibilitem uma interpretação alternativa, função do contexto em que são usados.

A Figura 1.6 apresenta os tipos de dados simples existentes na linguagem C. Estes tipos de dados também se designam por escalares, uma vez que, todos os seus valores estão distribuídos ao longo de uma escala linear. Dentro dos tipos de dados simples, temos o tipo **ponteiro** (***pointer***), o tipo **enumerado** (***enum***) e os tipos aritméticos, que se dividem em tipos inteiros e tipos reais. Os tipos aritméticos e o tipo enumerado designam-se por tipos básicos. Os tipos aritméticos inteiros podem armazenar valores negativos e positivos, que é o estado por defeito ou usando o qualificativo ***signed***, ou em alternativa, podem armazenar apenas valores positivos, usando para o efeito o qualificativo ***unsigned*** que lhe duplica a gama dinâmica positiva. O tipo aritmético ***int*** pode ainda ser qualificado como ***short***, reduzindo-lhe a capacidade de armazenamento. O qualificativo ***long*** pode ser usado para aumentar a capacidade de armazenamento do tipo inteiro ***int*** e do tipo real ***double***.

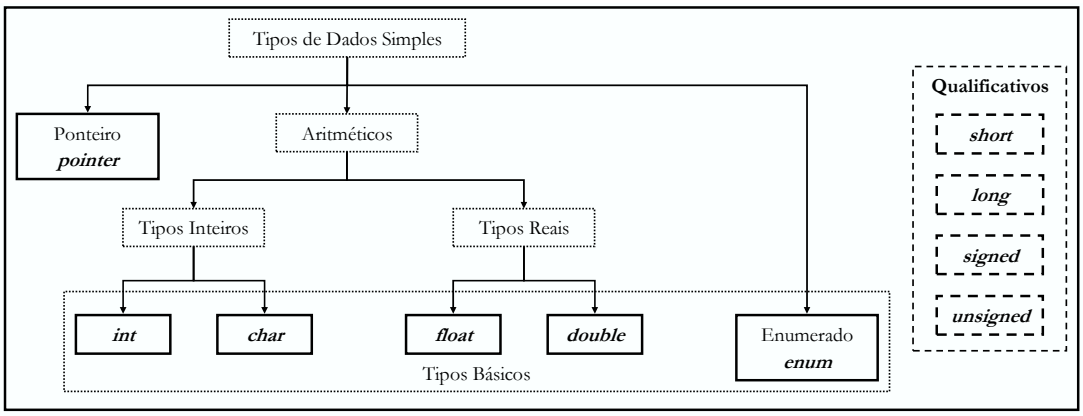

Figura 1.6 - Tipos de dados simples existentes na linguagem C.

1.4.1 Tipos de dados inteiros

Na linguagem C existem os seguintes tipos de dados inteiros:

- O tipo ***char*** permite a representação de quantidades com sinal num *byte* e portanto, permite armazenar valores entre -128 a 127.

- O tipo ***unsigned char*** permite a representação de quantidades sem sinal num *byte* e portanto, permite armazenar valores entre 0 e 255.

- O tipo ***short*** [***int***] permite a representação de números negativos e positivos em 2 *bytes* e portanto, permite armazenar valores entre -32768 e 32767.

- O tipo ***unsigned short*** [***int***] permite a representação de números positivos em 2 *bytes* e portanto, permite armazenar valores entre 0 e 65535.

- O tipo ***int*** permite a representação de números negativos e positivos em 2 *bytes* ou 4 *bytes*, consoante o computador.

- O tipo ***unsigned int*** permite a representação de números positivos em 2 *bytes* ou 4 *bytes*, consoante o computador.

- O tipo ***long*** [***int***] permite a representação de números negativos e positivos em 4 *bytes* ou 8 *bytes*, consoante o computador.

- O tipo ***unsigned long*** [***int***] permite a representação de números positivos em 4 *bytes* ou 8 *bytes*, consoante o computador.

O tamanho dos tipos inteiros ***int*** e ***long*** e a sua gama dinâmica dependem do compilador e do *hardware* utilizados. Esta informação é indicada no ficheiro *limits.h* localizado no directório *include* do ambiente de desenvolvimento. No caso do computador utilizado nesta disciplina, cujo processador é de 32 *bits*, os tipos ***int*** e ***unsigned int*** são representados em 4 *bytes*, pelo que, permitem armazenar respectivamente valores entre -2147483648 e 2147483647 e valores entre 0 e 4294967295. Os tipos ***long*** e ***unsigned long*** são também representados em 4 *bytes*.

1.4.2 Tipos de dados reais

Na linguagem C existem os seguintes tipos de dados reais: ***float***, ***double***, e ***long double***. O tamanho, a precisão e a gama dinâmica dos tipos reais dependem do compilador e do *hardware* utilizados. Esta informação é indicada no ficheiro *float.h* localizado no directório *include* do ambiente de desenvolvimento. Para o caso do computador utilizado nesta disciplina, o tamanho, a precisão e a gama dinâmica dos tipos reais são os seguintes:

- O tipo ***float*** utiliza 4 *bytes*, o que permite armazenar valores entre 1.2×10^{-38} e 3.4×10^{38}, com uma mantissa de 6-7 algarismos significativos.

- O tipo ***double*** utiliza 8 *bytes*, o que permite armazenar valores entre 2.2×10^{-308} e 1.8×10^{308}, com uma mantissa de 15-16 algarismos significativos.

- O tipo ***long double*** utiliza 12 *bytes*, o que permite armazenar valores entre 3.4×10^{-4932} e 1.2×10^{4932}, com uma mantissa de 18-19 algarismos significativos.

1.4.3 Representação de caracteres e inteiros

A maioria das linguagens de programação, entre as quais se inclui o Pascal, faz a distinção entre o tipo inteiro e o tipo carácter. Mesmo, apesar de nessas linguagens, os caracteres serem armazenados na memória em numérico, usando para o efeito o código ASCII.

Na linguagem C não existe tal distinção. O tipo ***char*** que utiliza um *byte* permite armazenar quer um carácter quer um valor inteiro. A Figura 1.7 apresenta dois exemplos que exemplificam esta polivalência. No primeiro caso a atribuição do valor 65, que é o código ASCII do carácter A, é equivalente à atribuição do próprio carácter 'A' à variável CAR. No segundo caso a atribuição do valor 3 à variável NUM é diferente da atribuição à variável CAR do carácter '3', cujo código ASCII é 51. Devido a esta polivalência de interpretação do valor inteiro armazenado na variável, o valor escrito no monitor depende do especificador de formato que seja empregue na instrução de saída de dados. O formato **%d** representa o valor decimal, enquanto que o formato **%c** representa o carácter.

```
char CAR;                /* declaração da variável CAR de tipo char */
...
CAR = 'A';                        /* ambas as atribuições armazenam */
CAR = 65;                      /* na posição de memória CAR o valor 65 */
```

```
char NUM, CAR; /* declaração das variáveis NUM e CAR de tipo char */
...
NUM = 3;           /* armazena na posição de memória NUM o valor 3 */
CAR = '3';        /* armazena na posição de memória CAR o valor 51 */
```

Figura 1.7 - Utilização do tipo ***char***.

1.5 Constantes e variáveis

Uma **constante** é um objecto, cujo valor se mantém invariante durante a execução do programa. A utilização de um valor constante num programa, não fornece qualquer indicação sobre o seu significado ou finalidade. Pelo que, a utilização de uma mnemónica, ou seja, um nome associado a um valor constante, permite aumentar a legibilidade de um programa. Por outro lado, se um valor constante aparecer mais do que uma vez ao longo do programa, pode acontecer que o programador cometa algum lapso na repetição do valor e ter um erro algorítmico que não é detectável pelo compilador e que pode ser muito difícil de detectar pelo próprio programador. A utilização de uma constante permite assim parametrizar um programa, melhorar a legibilidade, já que os valores são substituídos por nomes com significado explícito e, a robustez, porque a alteração do valor é realizada de um modo centralizado.

Ao contrário do que se passa em Pascal, a linguagem C não contempla a definição explícita de identificadores associados com constantes. Esta restrição pode ser ultrapassada, tal como se apresenta na Figura 1.2, usando a seguinte directiva do pré-processador.

#define *identificador de constante* *expressão*

O que o pré-processador faz, ao encontrar esta directiva no ficheiro fonte, é efectuar a partir desse ponto a substituição de todas as ocorrências do identificador de constante pela expressão, o que se designa por uma macro de substituição. Como se trata de um processo de substituição puro, e não de cálculo do valor associado, torna-se necessário, sempre que a expressão não for um literal, colocá-la entre parênteses curvos para garantir o cálculo correcto do seu valor, independentemente do contexto em que está localizada.

#define *identificador de constante* **(** *expressão* **)**

Um dos erros mais frequentemente cometido, por programadores que se estão a iniciar na utilização da linguagem C, quando utilizam esta directiva na definição de um identificador constante é a sua terminação com o **;**. Nesse caso o **;** torna-se parte da substituição podendo gerar situações de erro.

As constantes numéricas inteiras podem ser representadas no sistema decimal, no sistema octal, em que a constante é precedida pelo dígito 0 ou no sistema hexadecimal, em que a constante é precedida pelo dígito 0 e pelo carácter x. Quando as constantes estão representadas nos sistemas octal ou hexadecimal, o sinal é normalmente expresso de uma maneira implícita, usando a representação em complemento verdadeiro. A Figura 1.8 apresenta alguns exemplos, considerando uma representação do tipo ***int*** em 32 *bits*.

Sistema decimal	Sistema octal	Sistema hexadecimal
54	066	0x36
-135	-0207	-0x87
em complemento verdadeiro	037777777571	0xFFFFFF79
0	00	0x0

Figura 1.8 - Exemplos de constantes inteiras.

O compilador atribui por defeito o tipo ***int*** a uma constante numérica inteira. Quando este tipo não tem capacidade de armazenamento suficiente, os tipos seguintes são sucessivamente atribuídos. No caso de uma constante decimal, usa-se o tipo ***long***, ou se ainda for insuficiente o tipo ***unsigned long***. No caso de uma constante octal ou hexadecimal, usa-se pela seguinte ordem o tipo ***unsigned int***, ou o tipo ***long***, ou o tipo ***unsigned long***.

A atribuição do tipo pode ser forçada pelos sufixos U para ***unsigned*** e L para ***long***. Uma constante seguida do sufixo U é do tipo ***unsigned int*** ou do tipo ***unsigned long***. Uma constante seguida do sufixo L é do tipo ***long*** ou do tipo ***unsigned long***. Uma constante seguida do sufixo UL é do tipo ***unsigned long***.

As constantes numéricas reais são sempre expressas no sistema decimal, usando quer a representação em parte inteira e parte fraccionária, quer a chamada notação científica. O compilador atribui por defeito o tipo ***double*** a uma constante numérica real. A atribuição do tipo pode ser forçada pelos sufixos F e L. Uma constante seguida do sufixo F é do tipo ***float***. Uma constante seguida do sufixo L é do tipo ***long double***. São exemplos de constantes reais os valores 0.0148, 1.48e-2 e 0.0.

As constantes de tipo carácter podem ser expressas indiferentemente pelo respectivo símbolo gráfico, colocado entre aspas simples, ou através do valor do seu código de representação nos sistemas octal e hexadecimal, precedidas do carácter '\', tal como se mostra na Figura 1.9.

Sistema decimal	Sistema octal	Sistema hexadecimal
'B'	'\102'	'\x42'

Figura 1.9 - Exemplo de constante de tipo carácter.

Para alguns caracteres de controlo, pode ainda ser usada uma representação alternativa que consiste numa letra do alfabeto minúsculo precedida do carácter '\'. Por exemplo o carácter de *fim de linha* é o '\n', o carácter de *backspace* é o '\b', o carácter de *tabulação* é o '\t', o carácter aspas duplas é o '\"' e o carácter ponto de interrogação é o '\?'.

As constantes de tipo cadeia de caracteres são expressas como uma sequência de caracteres, representados por qualquer dos métodos anteriores, colocados entre aspas duplas. Por exemplo "Ola malta!\n".

Uma **variável** é um objecto, cujo valor se altera em princípio durante a execução do programa, excepção eventualmente feita às variáveis de entrada, cujo valor depois de lido do teclado é, em princípio, mantido inalterado até ao fim da execução do programa.

Todas as variáveis usadas num programa têm que ser previamente definidas ou declaradas. O objectivo da declaração é simultaneamente a reserva de espaço em memória para o armazenamento dos valores que as variáveis vão sucessivamente tomar, e a associação de cada identificador com a área de memória correspondente.

A Figura 1.10 apresenta a definição formal da declaração de variáveis na linguagem C. Para declarar variáveis começa-se por identificar o tipo de dados seguido da variável, ou lista de variáveis que se pretendem declarar desse tipo, separadas por vírgulas, terminando a declaração com o separador **;**. É conveniente agrupar a declaração de variáveis do mesmo tipo na mesma linha para aumentar a legibilidade do programa.

A reserva de espaço em memória não pressupõe, em princípio, a atribuição de um valor inicial à variável. Em consequência, nada deve ser presumido sobre o seu valor antes que uma primeira atribuição tenha sido efectivamente realizada. No entanto, a linguagem C permite combinar a definição de uma variável com a atribuição de um valor inicial, usando para o efeito o operador de atribuição seguido da expressão de inicialização.

declaração de variáveis ::= *declaração de variáveis de um tipo* |
declaração de variáveis | *declaração de variáveis de um tipo*

declaração de variáveis de um tipo ::= *tipo de dados* *lista de variáveis* **;**

tipo de dados ::= qualquer tipo de dados válido na linguagem C

lista de variáveis ::= *identificador de variável genérico* |
lista de variáveis **,** *identificador de variável genérico*

identificador de variável genérico ::= *identificador de variável* |
identificador de variável **=** *expressão de inicialização*

identificador de variável ::= identificador válido na linguagem C

Figura 1.10 - Definição formal da declaração de variáveis.

Ao contrário de outras linguagens de programação a linguagem C usa apenas um único símbolo, o símbolo **=**, como operador de atribuição. A diferença do operador de atribuição em relação ao Pascal é um dos erros mais frequentemente cometido por programadores que se estão a iniciar na utilização da linguagem C.

A região de reserva de espaço em memória principal não é necessariamente contígua, pelo que, é frequente surgirem buracos resultantes do alinhamento das variáveis, em áreas de endereços múltiplos de 4, para maximizar a taxa de transferência de informação entre o processador e a memória principal. A Figura 1.11 apresenta alguns exemplos de declaração de variáveis e sua colocação na memória.

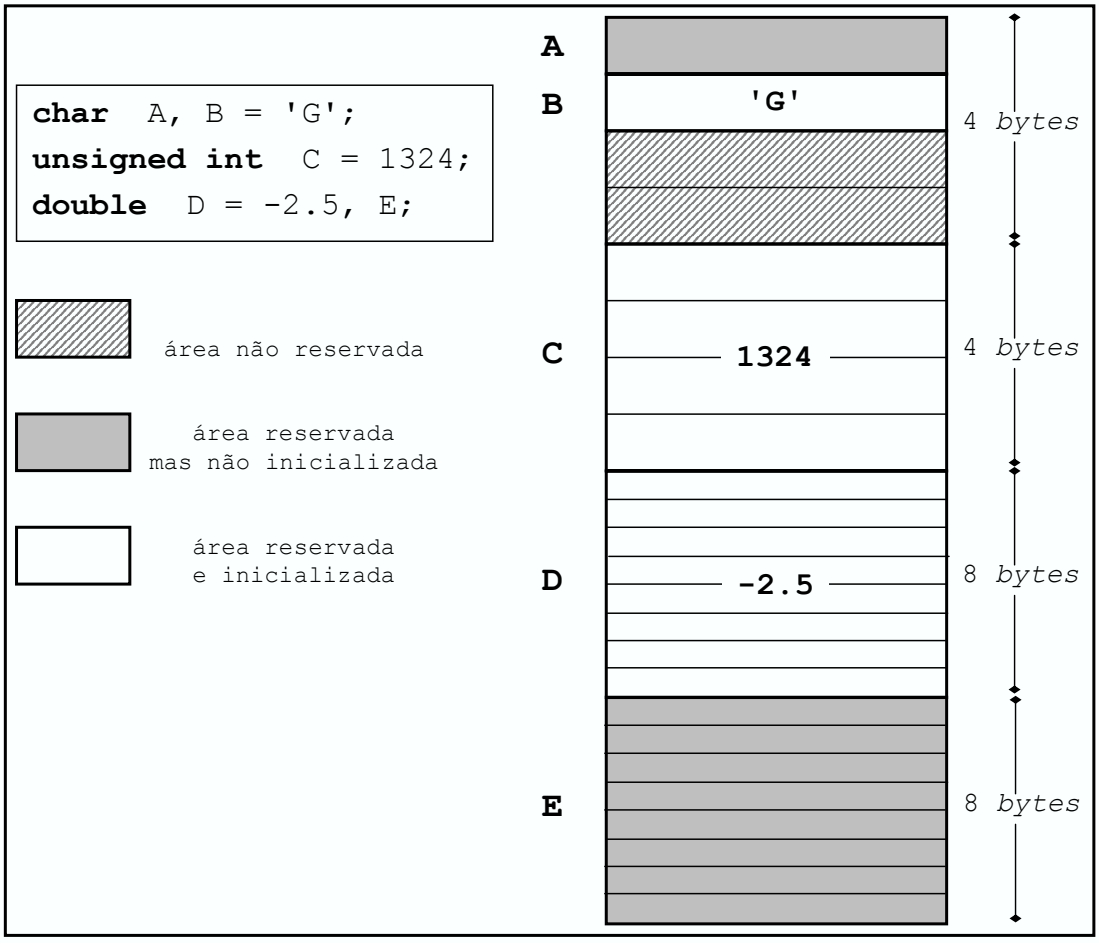

Figura 1.11 - Alguns exemplos de declaração de variáveis e sua colocação na memória.

1.6 Sequenciação

Tal como foi referido anteriormente, o corpo de uma função é delimitado pelos separadores **{** e **}**, correspondentes, respectivamente, aos separadores **begin** e **end** do Pascal, e contém a declaração das variáveis locais, a alusão a funções usadas na sequência de instruções e a sequência de instruções propriamente dita. A Figura 1.12 apresenta a definição formal de uma sequência de instruções. Ao contrário do que se passa em Pascal, na linguagem C cada instrução simples é obrigatoriamente terminada com o separador **;**, a menos que o último símbolo da instrução seja o separador **}**.

sequência de instruções simples ::= *instrução simples* |
sequência de instruções simples *instrução simples*

instrução simples ::= instrução de atribuição |
instrução decisória |
instrução repetitiva |
instrução de entrada-saída |
invocação de uma função

Figura 1.12 - Definição formal da sequência de instruções.

Em Pascal, a invocação de uma função, sendo uma expressão, não pode ser considerada uma instrução. Na linguagem C, contudo, o conceito de procedimento não tem uma existência separada. Define-se como sendo uma função de um tipo especial, o tipo **void**. Pelo que, a invocação de um procedimento é, por isso, em tudo semelhante à invocação de uma função de qualquer outro tipo, quando o valor devolvido não é tido em consideração. Logo, nestas circunstâncias, na linguagem C, a pura e simples invocação de uma função de qualquer tipo é considerada uma instrução.

A sequenciação de instruções é a forma mais simples de controlo de fluxo num programa, em que as instruções são executadas pela ordem em que aparecem no programa. Dentro das instruções simples, apenas as instruções de atribuição, de entrada-saída e de invocação de uma função, são verdadeiramente instruções de sequenciação, já que, as instruções decisórias e repetitivas permitem alterar a ordem do fluxo do programa.

Tal como no Pascal, na linguagem C também existe o conceito de instrução composta, cuja definição formal se apresenta na Figura 1.13, e que é composta por uma sequência de instruções simples encapsuladas entre os separadores **{** e **}**. Uma instrução composta é um bloco de instruções simples que se comporta como uma instrução única e é usada em instruções decisórias e repetitivas.

instrução composta ::= **{** *sequência de instruções simples* **}**

Figura 1.13 - Definição formal da instrução composta.

1.6.1 Expressões

A Figura 1.14 apresenta a definição formal de uma expressão. Uma expressão é uma fórmula que produz um valor. Pode assumir as seguintes formas: ser uma constante; ser uma variável; ser o resultado da invocação de uma função; ser uma expressão composta por operandos e operadores, sendo que existem operadores unários e binários; e ser uma expressão entre parênteses curvos.

expressão ::= constante | variável | invocação de uma função |
operador unário *expressão* | *expressão* **operador unário**
expressão **operador binário** *expressão* | **(** *expressão* **)**

Figura 1.14 - Definição formal de uma expressão.

Os primeiros três tipos de expressões são a fórmula mais simples de produzir um valor e resumem-se à atribuição a uma variável, do valor de uma constante, ou do valor de uma variável ou do valor devolvido por uma função. As expressões mais complexas envolvem operadores unários ou binários. Uma expressão pode ser composta por uma expressão colocada entre parênteses curvos. Este tipo de expressão usa-se para compor expressões mais complexas, ou para modificar a prioridade do cálculo de expressões parcelares.

Vamos agora analisar as **expressões aritméticas**. O cálculo de uma expressão complexa supõe um processo de decomposição prévio em expressões mais simples, que é determinado pela **precedência** e pela **associatividade** dos operadores presentes. Precedência significa importância relativa entre os operadores. Operadores de maior precedência forçam a ligação a si dos operandos, antes dos operadores de menor precedência Por exemplo, como a multiplicação tem precedência sobre a adição, então a expressão **a + b * c** tem subjacente o agrupamento **a + (b * c)**.

Quando numa expressão todos os operadores têm a mesma precedência, a associatividade permite determinar a ordem de ligação dos operandos aos operadores, da direita para a esquerda, ou da esquerda para a direita. Por exemplo, como a associatividade da adição é da direita para a esquerda, então a expressão **a + b + c** tem subjacente o agrupamento **(a + b) + c**.

No entanto, qualquer que seja o agrupamento imposto pela precedência e pela associatividade dos operadores presentes, ele pode ser sempre alterado pela introdução de parênteses curvos.

No caso de aritmética inteira, sempre que numa expressão surgem constantes, variáveis, ou se invocam funções de tipo ***char*** ou ***short***, os seus valores são automaticamente convertidos pelo compilador em quantidades de tipo ***int***. Do mesmo modo, constantes, variáveis, ou funções de tipo ***unsigned char*** ou ***unsigned short***, são automaticamente convertidas pelo compilador em quantidades de tipo ***int***, ou de tipo ***unsigned int***, se o primeiro tipo não tiver capacidade de armazenamento suficiente. No caso de aritmética real, sempre que numa expressão surgem constantes, variáveis, ou se invocam funções de tipo ***float***, os seus valores são automaticamente convertidos pelo compilador em quantidades de tipo ***double***.

Os operadores binários supõem normalmente operandos do mesmo tipo. Quando são de tipo diferente, a expressão do tipo com menor capacidade de armazenamento, é automaticamente convertida no tipo da outra expressão. A hierarquia dos diferentes tipos de dados, expressa por ordem decrescente da sua capacidade de armazenamento, é a que se apresenta a seguir.

long double → double → float → unsigned long → long → unsigned int → int

Além das conversões automáticas que foram referidas, a conversão de uma expressão num tipo específico qualquer pode ser sempre forçada através do operador **cast**.

(*qualquer tipo de dados escalar válido em C* **)** **(** *expressão numérica* **)**

A Figura 1.15 apresenta um exemplo da utilização do operador ***cast***. A divisão de duas variáveis inteiras dá um resultado inteiro. Para forçar a divisão real é preciso forçar um dos operandos a ***double***, fazendo um **cast** de um dos operandos, neste caso da variável A.

```
int A = 5, B = 2;  double DIVISAO;
...
DIVISAO = A / B;                                    /* DIVISAO = 2.0 */
-------------------------------------------------------------------------
DIVISAO = (double) A / B;                           /* DIVISAO = 2.5 */
```

Figura 1.15 - Exemplo da utilização do operador cast.

Se a conversão se efectua no sentido crescente da hierarquia, não há risco de *overflow* ou de perda de precisão. Caso contrário, estes problemas podem ocorrer. Concretamente, quando a conversão se efectua no sentido decrescente da hierarquia, podemos ter as situações apresentadas na Figura 1.16.

Tipos de dados de partida	**Tipos de dados de chegada**
long double	***double*** ou ***float***
existe arredondamento e portanto pode ocorrer *overflow*	
double	***float***
existe arredondamento e portanto pode ocorrer *overflow*	
qualquer tipo real	**qualquer tipo inteiro**
existe truncatura e portanto pode ocorrer *overflow*	
tipo inteiro *signed* (***unsigned***)	**mesmo tipo inteiro *unsigned*** (***signed***)
mudança na interpretação do valor	
tipo inteiro *unsigned*	**tipo hierarquicamente inferior *unsigned***
resto do módulo do registo de chegada	
tipo inteiro *signed*	**tipo hierarquicamente inferior *signed***
pode ocorrer *overflow*	

Figura 1.16 - Situações possíveis se a conversão se efectuar no sentido decrescente da hierarquia.

Vamos agora apresentar alguns exemplos representativos destes problemas. A Figura 1.17 apresenta um exemplo da situação em que se atribui um valor negativo de uma variável ***int*** a uma variável ***unsigned int***. O valor armazenado na memória em binário vai ser interpretado como sendo positivo, pelo que, há uma mudança na interpretação do valor.

```
int A = -1024;  unsigned int B;
...
B = A;
    /* A = -1024₁₀ = 1111 1111 1111 1111 1111 1100 0000 0000₂ */
  /* B = 1111 1111 1111 1111 1111 1100 0000 0000₂ = 4294966272₁₀*/
```

Figura 1.17 - Mudança na interpretação do valor.

A Figura 1.18 apresenta um exemplo da situação em que se atribui uma variável ***unsigned int*** a uma variável ***unsigned char***, que tem uma menor capacidade de armazenamento. O valor que vai ser armazenado em B é constituído pelos últimos 8 *bits* de A, ou seja, é o resto da divisão de A por 256, que é o máximo valor que se pode armazenar num *byte*.

```
unsigned int A = 1025;  unsigned char B;
...
B = A;
     /* A = 1025₁₀ = 0000 0000 0000 0000 0000 0100 0000 0001₂ */
                        /* B = 0000 0001₂ = 1₁₀*/
```

Figura 1.18 - Resto do módulo do registo de chegada.

A Figura 1.19 apresenta um exemplo da situação em que se atribui uma variável ***int*** a uma variável ***char***, que tem uma menor capacidade de armazenamento. Como o valor de A excede a capacidade de armazenamento de B, então temos uma situação de *overflow*. Esta situação é facilmente detectada, uma vez que, o valor de chegada é negativo, quando o valor de partida era positivo.

```
int A = 1152;  char B;
...
B = A;
     /* A = 1025₁₀ = 0000 0000 0000 0000 0000 0100 1000 0000₂ */
                        /* B = 1000 0000₂ = -128₁₀*/
```

Figura 1.19 - Ocorrência de *overflow*.

No cálculo de uma expressão complexa, uma vez fixado o agrupamento, segundo as regras da prioridade e da associatividade dos operadores envolvidos, modificadas ou não pela introdução de parênteses curvos, a ordem pela qual o compilador calcula as diferentes subexpressões é em larga medida arbitrária. O compilador pode mesmo reorganizar a expressão, se isso não afectar o resultado final. Em geral, esta questão não acarreta consequências graves.

Contudo, sempre que o cálculo de uma expressão envolva operadores com **efeitos colaterais**, ou seja, uma expressão em que o valor de uma ou mais variáveis é afectado pelo processo de cálculo, normalmente devido à utilização dos operadores unários incremento e decremento, o código resultante pode deixar de ser portável. Assim, é de extrema importância organizar cuidadosamente a formação das expressões para se evitar que tais situações ocorram.

Quando no cálculo de uma expressão existe o risco de ocorrência de *overflow* em resultado de uma possível transformação da expressão numa equivalente, tal como se mostra na Figura 1.20, isso deve ser impedido por decomposição do cálculo da expressão em duas ou mais expressões parcelares, senão o código resultante pode deixar de ser portável.

```
int X, K1 = 1024, K2 = 4096, K3 = 4094;
...
X = K1 * K1 * (K2 - K3);
     /* se o compilador transformar a expressão na expressão */
  /* aparentemente equivalente X = K1 * K1 * K2 - K1 * K1 * K3 */
  /* vai ocorrer overflow para uma representação int em 32 bits */
```

Figura 1.20 - Exemplo de uma expressão onde existe o risco de ocorrência de *overflow*.

A linguagem C permite ainda a construção de um tipo especial de expressão, cuja finalidade é fornecer o tamanho em *bytes* do formato de um tipo de dados particular. Esse tipo pode ser representado, explicitamente, pelo seu identificador, ou, implicitamente, por qualquer expressão desse tipo.

sizeof (*qualquer tipo de dados válido em C* **)** ou **sizeof (** *expressão* **)**

No segundo caso, o valor da expressão nunca é calculado, sendo unicamente determinado o seu tipo. A norma ANSI exige que o tipo do resultado do operador **sizeof** seja do tipo inteiro e ***unsigned***, ou seja, do tipo ***unsigned int*** ou do tipo ***unsigned long***. Este tipo é o tipo *size_t* que está definido na biblioteca ***stdlib***.

1.6.2 Operadores

A Figura 1.21 apresenta os operadores aritméticos disponíveis na linguagem C.

Operadores Unários

Operador	Símbolo	Sintaxe	Observações
	+	+x	
Simétrico	-	-x	
Incremento de 1	++	++x	x tem que ser
		x++	uma variável
Decremento de 1	--	--x	x tem que ser
		x--	uma variável

Operadores Binários

Operador	Símbolo	Sintaxe	Observações
Adição	+	x + y	
Subtracção	-	x - y	
Multiplicação	*	x * y	
Divisão	/	x / y	y ≠ 0
Resto da divisão inteira	%	x % y	y ≠ 0
	e x e y têm de ser expressões inteiras		

Figura 1.21 - Operadores aritméticos.

Os operadores unários incremento e decremento, só podem ser usados com variáveis e, incrementam e decrementam o valor da variável de uma unidade. Podem ser colocados antes da variável, o que se designa por pré-incremento e pré-decremento. Nesse caso o valor da variável é alterado antes da sua utilização na expressão em que a variável está inserida. Ou, podem ser colocados depois da variável, o que se designa por pós-incremento e pós-decremento. Nesse caso o valor da variável só é alterado depois da sua utilização na expressão em que a variável está inserida.

A Figura 1.22 apresenta um exemplo da utilização do operador unário incremento. Na primeira expressão, uma vez que estamos perante o pós-incremento de X, então no cálculo de Y é utilizado o valor de X antes deste ser incrementado, pelo que, Y fica com o valor 10 e só depois é que o valor de X é incrementado. Na segunda expressão, uma vez que estamos perante o pré-incremento de X, então em primeiro lugar X é incrementado e só depois é calculado o valor de Y, pelo que, Y fica com o valor 15. Em ambas as expressões o valor final de X é igual a 3.

```
int X = 2, Y;
...
Y = 5 * X++;        /* após o cálculo da expressão X = 3 e Y = 10 */
-------------------------------------------------------------------
Y = 5 * ++X;        /* após o cálculo da expressão X = 3 e Y = 15 */
```

Figura 1.22 - Exemplo da utilização do operador unário incremento.

A Figura 1.23 apresenta uma expressão que utiliza o operador pós-incremento incorrectamente, uma vez que, o código resultante pode deixar de ser portável. Neste tipo de expressão é impossível prever o valor final da expressão, uma vez que tal depende da ordem de cálculo dos operandos. Pelo que, quando numa expressão existe um operador com efeito colateral, a variável afectada só pode ser usada uma e uma única vez.

```
int X, K = 3;
...
X = K * K++;
 /* se a ordem de cálculo for da esquerda para a direita, X = 9 */
/* se a ordem de cálculo for da direita para a esquerda, X = 12 */
```

Figura 1.23 - Exemplo de uma expressão com utilização incorrecta do operador pós-incremento.

O operador divisão é usado simultaneamente para representar o quociente da divisão inteira e da divisão real. Tal como se mostra na Figura 1.15, tratar-se-á de uma divisão inteira, sempre que os operandos forem inteiros e tratar-se-á de uma divisão real, quando pelo menos um dos operandos for real.

Quando as expressões do quociente e do resto da divisão inteira são quantidades negativas, o resultado não é univocamente determinado e depende do compilador utilizado. No caso do quociente da divisão inteira, a norma ANSI possibilita dois tipos de aproximação, a truncatura ou a aproximação ao maior inteiro, menor ou igual ao correspondente quociente real. Se os operandos são ambos positivos ou negativos, o resultado obtido pelos dois métodos é idêntico. Se um dos operandos é negativo, tal não se passa. Por exemplo, 7/(-3) ou -7/3, tanto pode produzir um quociente de -2, como de -3. Para evitar esta ambiguidade e garantir-se a portabilidade, sempre que haja a possibilidade do quociente ser negativo, a operação x/y deve ser substituída, por exemplo, pela expressão seguinte, de maneira a forçar a aproximação por truncatura.

(***tipo inteiro***) ((***double***) x/y)

Para o resto da divisão inteira a norma ANSI impõe que se verifique a seguinte condição.

x = x % y + (x / y) * y

Isto significa que, conforme o tipo de aproximação usado em x/y e a localização do operando negativo, diferentes restos são possíveis quando um dos operandos é negativo.

7 % (-3) = 1 -7 % 3 = -1 (truncatura)
7 % (-3) = -2 -7 % 3 = 2 (aproximação ao maior inteiro menor ou igual)

Na prática, o problema não é tão grave, porque a expressão x%y não faz matematicamente sentido para y negativo e, por isso, não deve nunca ser usada neste contexto. Além disso, quando x é negativo, é fácil verificar que os dois resultados possíveis são congruentes. Contudo, para se obter uma portabilidade completa, é conveniente substituir a expressão x%y, pela construção condicional seguinte.

```
if  (x >= 0)   r = x % y ;
else  r = y - (-x) % y ;
```

A Figura 1.24 apresenta os operadores relacionais disponíveis na linguagem C. Em relação ao Pascal existe apenas diferença nos operadores de igualdade e de desigualdade.

Operador	Símbolo	Sintaxe
Igual	==	x == y
Diferente	!=	x != y
Maior	>	x > y
Menor	<	x < y
Maior ou igual	>=	x >= y
Menor ou igual	<=	x <= y

Figura 1.24 - Operadores relacionais.

A expressão resultante é de tipo ***int*** e assume o valor zero, se o resultado da comparação for falso, e o valor um, se o resultado for verdadeiro. Ao contrário do que se passa geralmente, a precedência dos diversos operadores não é a mesma. Os operadores maior ou igual, menor ou igual, maior e menor têm maior precedência do que os operadores igual e diferente. Dado que os tipos reais fornecem apenas uma representação aproximada das quantidades numéricas, nunca deve ser usado o operador igualdade com operandos desse tipo. Mesmo os operadores maior ou igual e menor ou igual devem ser usados com extremo cuidado. A expressão booleana de terminação de um processo de contagem, nomeadamente, nunca deve ser formada com expressões de tipo real.

A Figura 1.25 apresenta os operadores lógicos que se aplicam tanto a expressões numéricas inteiras como a reais. Os operandos são interpretados como representando o valor falso, se forem iguais a zero, e como representando o valor verdadeiro, em todos os restantes casos. A expressão resultante é de tipo ***int*** e assume o valor zero, se o resultado da comparação for falso, e o valor um, se o resultado for verdadeiro. Ao contrário do Pascal, uma expressão formada por operadores lógicos nem sempre é calculada até ao fim. O cálculo procede da esquerda para a direita e o processo é interrompido logo que o resultado esteja definido.

Operador Unário		
Operador	**Símbolo**	**Sintaxe**
Negação (not)	!	!x
Operadores Binários		
Operador	**Símbolo**	**Sintaxe**
Conjunção (and)	&&	x && y
Disjunção inclusiva (or)	||	x || y

Figura 1.25 - Operadores lógicos.

A Figura 1.26 apresenta os operadores para manipulação de *bits* que se aplicam apenas a expressões numéricas inteiras.

Operador Unário		
Operador	**Símbolo**	**Sintaxe**
Complemento (not)	~	~x
Operadores Binários		
Operador	**Símbolo**	**Sintaxe**
Conjunção (and)	&	x & y
Disjunção inclusiva (or)	|	x | y
Disjunção exclusiva (xor)	^	x ^ y
Deslocamento à direita	>>	x >> y
Deslocamento à esquerda	<<	x << y

Figura 1.26 - Operadores de manipulação de *bits*.

Os operadores lógicos actuam isoladamente sobre cada um dos *bits* dos operandos. Para os operadores lógicos binários, as operações são efectuadas em paralelo sobre os *bits* localizados em posições correspondentes de cada um dos operandos.

O resultado da operação é uma quantidade inteira do tipo do operando com maior capacidade de armazenamento, no caso dos operadores lógicos binários, ou do tipo do operando x, no caso do operador complemento booleano ou dos operadores de deslocamento. Para os operadores de deslocamento, o operando y representa o número de posições a deslocar no sentido pretendido. O resultado da operação não está definido, quando y é maior ou igual do que o comprimento em *bits* do operando x, ou é negativo. A norma ANSI impõe a realização de um deslocamento lógico, quando x for de um tipo qualquer ***unsigned***. Contudo, nada é garantido quando x for de um tipo ***signed***, embora normalmente o deslocamento seja então aritmético. Assim, para se obter uma portabilidade completa, deve fazer-se sempre um ***cast*** para tipos ***unsigned***.

(***unsigned int*** ou ***unsigned long***) x >> y
(***unsigned int*** ou ***unsigned long***) x << y

A Figura 1.27 apresenta a tabela de associatividade e de precedência, por ordem decrescente, entre os operadores das expressões numéricas.

Operadores na classe	Associatividade	Precedência
operadores unários		
operador **cast**	direita → esquerda	**maior**
operador **sizeof**	direita → esquerda	
- +	direita → esquerda	
-- ++ ! ~	direita → esquerda	
operadores binários		
*** / %**	esquerda → direita	
- +	esquerda → direita	↓
>> <<	esquerda → direita	
>= <= > <	esquerda → direita	
=! ==	esquerda → direita	
&	esquerda → direita	
^	esquerda → direita	
|	esquerda → direita	
&&	esquerda → direita	
||	esquerda → direita	**menor**

Figura 1.27 - Precedência e associatividade entre os operadores das expressões numéricas.

1.6.3 Instruções de atribuição

Ao contrário do Pascal que só tem uma forma da instrução de atribuição, na linguagem C existem as quatro variantes da instrução de atribuição que se apresentam na Figura 1.28.

Regra geral, os tipos da variável e da expressão podem ser quaisquer, desde que sejam tipos escalares. Após o cálculo da expressão e antes que a atribuição tenha lugar, o valor da expressão é automaticamente convertido para o tipo da variável.

instrução de atribuição ::= *identificador de variável* **=** *expressão* **;** |
identificador de variável **operador binário=** *expressão* **;** |
operador unário inc_dec identificador de variável **;** |
identificador de variável operador unário inc_dec **;** |
identificador de variável **=** *expressão-1* **?** *expressão_2* **:** *expressão_3* **;**

operador unário inc_dec ::= operador unário **++** | operador unário **−−**

Figura 1.28 - Definição formal da instrução de atribuição.

A primeira variante da instrução de atribuição é a instrução de atribuição típica das linguagens imperativas, em que se atribui o valor de uma expressão a uma variável. É semelhante à encontrada em Pascal. A única diferença é que o operador de atribuição é, neste caso o **=**, em vez de **:=**. A Figura 1.29 apresenta três exemplos.

```
int X, Y, Z;
...
Y = X + 2 * Z;
X = X + 1;
Z = pow (Y, 2);          /* pow é a função potência, donde Z = Y² */
```

Figura 1.29 - Exemplos da primeira variante da instrução de atribuição.

A segunda variante é uma construção típica da sintaxe compacta da linguagem C, onde o operador de atribuição é precedido por um qualquer operador binário. Esta variante é definida pela expressão geral **operador binário=**, em que o **operador binário** representa qualquer operador binário aritmético (*, /, %, +, −), ou de manipulação de *bits* (&, |, ^, >>, <<). A instrução é equivalente à operação binária da expressão com a própria variável como segundo operando, ou seja, é equivalente à seguinte instrução de atribuição.

identificador de variável **=** *identificador de variável* **operador binário** *expressão*

A importância desta notação tem a ver não só com uma maior clareza na indicação da operação a realizar, particularmente quando o identificador representa um campo de uma variável de tipo complexo, como também com a possibilidade fornecida ao compilador de proceder mais facilmente a uma optimização do código gerado. A Figura 1.30 apresenta dois exemplos.

```
int X, Y, Z;
...
Y += 5;                              /* equivalente a Y = Y + 5 */
Z *= 5 + X;                      /* equivalente a Z = Z * (5 + X) */
```

Figura 1.30 - Exemplos da segunda variante da instrução de atribuição.

A terceira variante é a utilização dos operadores unários incremento e decremento sobre uma variável. Independentemente se serem utilizados na situação de pré ou pós actuação sobre a variável, os efeitos colaterais apresentados por este tipo de operadores resultam no incremento ou decremento de uma unidade ao valor da variável. As instruções **++***identificador de variável***;** e *identificador de variável* **++;** são equivalentes à seguinte instrução.

identificador de variável **=** *identificador de variável* **+** 1 **;**

E, as instruções **−−***identificador de variável***;** e *identificador de variável***−−;** são equivalentes à seguinte instrução.

identificador de variável **=** *identificador de variável* **−** 1 **;**

A importância desta notação tem a ver de novo com a compactação resultante e com a possibilidade fornecida ao compilador de proceder mais facilmente a uma optimização do código gerado. A Figura 1.31 apresenta dois exemplos.

```
int X, Y;
...
Y++;                                    /* equivalente a Y = Y + 1 */
--X;                                    /* equivalente a X = X - 1 */
```

Figura 1.31 - Exemplos da terceira variante da instrução de atribuição.

A quarta variante é uma instrução de atribuição condicional, onde o valor da primeira expressão é avaliada e caso seja verdadeira, então é atribuído o valor resultante do cálculo da segunda expressão à variável, senão é atribuído o valor resultante do cálculo da terceira expressão à variável. Comporta-se assim como a instrução condicional binária ***if then else*** e é equivalente ao código em linguagem C que se apresenta a seguir.

if (*expressão_1*) *identificador de variável* **=** *expressão_2* **;**
else *identificador de variável* **=** *expressão_3* **;**

Em termos formais, trata-se de um caso particular da primeira variante em que a expressão aí indicada é do tipo *expressão_1* **?** *expressão_2* **:** *expressão_3*, ou seja, o agrupamento das três expressões pelo operador **? :**. Este operador constitui o único exemplo de operador ternário existente na linguagem C. Enquanto que a segunda e a terceira expressões podem ser de qualquer tipo válido na linguagem C, desde que compatíveis, segundo as regras de conversão, com o tipo da variável, a primeira expressão é de um tipo escalar básico. A precedência deste operador surge imediatamente abaixo dos operadores descritos até ao momento e a sua associatividade é da direita para a esquerda. A Figura 1.32 apresenta um exemplo.

```
int X, ABSX;
...
ABSX = X < 0 ? -X : X;
  /* equivalente a if (X < 0)  then ABSX = -X;  else ABSX = X; */
```

Figura 1.32 - Exemplo da quarta variante da instrução de atribuição.

Uma característica notável da Linguagem C é que a instrução de atribuição é também uma expressão. O seu valor é o valor atribuído ao operando da esquerda, ou seja, ao identificador de variável. O que faz com que os operadores **=** e **operador binário=** apresentem uma precedência imediatamente abaixo do operador condicional e uma associatividade igualmente da direita para a esquerda. Tornam-se por isso possíveis instruções de atribuição múltipla, cuja sintaxe é a que se apresenta a seguir.

identificador de variável_1 **=** ... **=** *identificador de variável_N* **=** *expressão* **;**

Note-se que, devido à associatividade e à regra geral de conversão automática, é relevante a ordem pela qual surgem na instrução as variáveis, quando pertencentes a tipos escalares diferentes. Qualquer alteração a esta ordem pode conduzir a valores distintos atribuídos a algumas delas. É também perfeitamente possível substituir, em qualquer ponto da cadeia de atribuições, o operador **=** pelo operador **operador binário=**, numa das suas múltiplas formas. Isto, contudo, não deve nunca ser feito, porque a interpretação da instrução resultante torna-se muito difícil e, portanto, muito sujeita a erros. A Figura 1.33 apresenta dois exemplos de instruções de atribuição múltipla.

```
int X, Y, Z;
...
X = Y = Z;                              /* equivalente a X = (Y = Z); */
X += Y -= Z;                        /* equivalente a X = (X + (Y = (Y-Z)); */
```

Figura 1.33 - Exemplos de instruções de atribuição múltipla.

A Figura 1.34 apresenta a precedência e a associatividade entre os operadores de atribuição.

Operadores na classe	Associatividade	Precedência
operador condicional **? :**	direita → esquerda	**maior**
= += -= *= /= %= >>= <<= &= ^= |=	direita → esquerda	**menor**

Figura 1.34 - Precedência e associatividade entre os operadores de atribuição.

1.7 Estruturas de controlo

1.7.1 Instruções decisórias

Na linguagem C existem dois tipos de instruções de tomada de decisão. A instrução decisória binária **if** e a instrução decisória múltipla **switch**.

1.7.1.1 A instrução decisória binária *if*

A instrução decisória binária **if** (se), cuja definição formal se apresenta na Figura 1.35, tem duas variantes que são fundamentalmente semelhantes às encontradas em Pascal. A única diferença reside no facto do separador **then** não surgir aqui. Em consequência, a expressão decisória é obrigatoriamente colocada entre parênteses curvos para estabelecer a separação da instrução a executar. A expressão decisória deve ser de um tipo escalar básico. A expressão é falsa se for igual a zero e é verdadeira se assumir qualquer outro valor. Nestas condições, o compilador aceita qualquer expressão numérica como expressão decisória válida. Porém, por questões de clareza, isto deve ser evitado. É de bom estilo que a expressão decisória represente sempre uma expressão booleana.

instrução decisória binária ::= **if (** *expressão* **)** instrução simples ou composta |
if (*expressão* **)** instrução simples ou composta
else instrução simples ou composta

expressão ::= expressão decisória de um tipo escalar básico

Figura 1.35 - Definição formal da instrução **if**.

Uma questão muito importante para editar programas legíveis é o alinhamento das instruções. A Figura 1.36 apresenta como se deve alinhar a instrução **if**. No caso da variante mais simples e se existir apenas uma instrução simples curta, então a instrução **if** pode ser toda escrita na mesma linha, mas se a instrução simples for longa deve ser escrita na linha seguinte mais alinhada para a direita. Caso a instrução seja composta, então os separadores **{** e **}** devem ser alinhados com o **if**. No caso da variante completa, devemos alinhar o separador **else** com o **if**.

```
if  ( expressão )  instrução simples;
```

```
if  ( expressão )
{
   instrução simples;
   ...
   instrução simples;
}
```

```
if  ( expressão )
{
   instrução simples;
   ...
   instrução simples;
}
else
{
   instrução simples;
   ...
   instrução simples;
}
```

Figura 1.36 - Alinhamento da instrução **if**.

A Figura 1.37 apresenta um encadeamento de instruções **if**, o que se designa por **instruções if encadeadas** (*nested if structures*). Neste tipo de agrupamento, a regra de agrupamento é semelhante à de Pascal. O compilador associa sempre o separador **else** à instrução **if** que ocorreu imediatamente antes.

```
if  ((CAR >= 'A') && (CAR <= 'Z'))
   printf ("Carácter Maiusculo\n");
else  if  ((CAR >= 'a') && (CAR <= 'z'))
         printf ("Carácter Minusculo\n");
      else  if  ((CAR >= '0') && (CAR <= '9'))
               printf ("Carácter Numerico\n");
            else  printf ("Outro Carácter\n");
```

Figura 1.37 - Construção de instruções **if** encadeadas.

Vamos agora considerar a situação apresentada na Figura 1.38, em que queremos ter o separador **else** para o primeiro **if**, mas em que o segundo **if** não o tem. Neste tipo de situação, que se designa por **else desligado** (*dangling else*), o separador **else** vai ser atribuído pelo compilador ao segundo **if**, independentemente de ter sido alinhado com o primeiro **if**.

```
if  (V1 > 0)
  if  (V2 > 0)  V2++;
else  V1++;                          /* situação do else desligado */
```

Figura 1.38 - Situação do **else** desligado.

Para resolver este problema existem as duas soluções apresentadas na Figura 1.39. A primeira, consiste em usar uma instrução composta a abraçar o segundo **if**, de maneira a informar o compilador onde acaba o segundo **if**. A segunda, consiste em usar um separador **else** com uma instrução nula, para emparelhar com o segundo **if**, e assim forçar o emparelhamento do segundo separador **else** com o primeiro **if**. A instrução nula é o **;**.

```
if  (V1 > 0)
{
   if  (V2 > 0)  V2++;
}
else  V1++;
```

```
if  (V1 > 0)
   if  (V2 > 0)  V2++;
   else ;
else  V1++;
```

Figura 1.39 - Soluções para a situação do **else** desligado.

Devido ao facto das instruções de atribuição constituírem também expressões, um erro muito frequentemente cometido por programadores que se estão a iniciar na utilização da linguagem C e que não é sinalizado pelo compilador, consiste em trocar o operador identidade pelo operador de atribuição na escrita da condição decisória, tal como se apresenta na Figura 1.40. O valor da expressão A = 5 é 5 e como é um valor diferente de zero, então a expressão é verdadeira e vai ser executada a instrução B = 2. Pelo que, após a execução da instrução, as variáveis A e B vão assumir respectivamente, os valores 5 e 2.

```
if (A = 5)  B = 2;   /* o que se pretendia era if (A == 5)  B = 2; */
else B = 4;
```

Figura 1.40 - Erro devido à troca do operador identidade pelo operador de atribuição.

1.7.1.2 A instrução decisória múltipla *switch*

Muitas das situações de estruturas **if** encadeadas podem ser resolvidas através da instrução decisória múltipla **switch** (comutador), cuja definição formal se apresenta na Figura 1.41.

instrução decisória múltipla ::= **switch (*expressão*)**
{
bloco de execução
}

bloco de execução ::= *bloco de execução selectivo* |
bloco de execução bloco de execução terminal

bloco de execução selectivo ::= *lista de constantes sequência de instruções simples* |
bloco de execução selectivo
lista de constantes sequência de instruções simples

lista de constantes ::= **case** constante inteira **:** |
lista de constantes
case constante inteira **:**

bloco de execução terminal ::= **default :** *sequência de instruções simples*

Figura 1.41 - Definição formal da instrução **switch**.

Embora as palavras reservadas na linguagem C sejam distintas das do Pascal, a estrutura sintáctica da instrução de decisão múltipla é essencialmente a mesma nas duas linguagens. Na norma ANSI, a expressão de decisão, bem como as constantes que formam a lista de constantes, são de qualquer tipo escalar básico inteiro. A principal diferença decorre do modo como os vários ramos de selecção estão organizados. Em Pascal, eles são mutuamente exclusivos, enquanto que, na linguagem C, a execução é sequencial a partir do

ponto de entrada. Para se conseguir o mesmo tipo de execução encontrado em Pascal, a última instrução de cada sequência de instruções terá que ser obrigatoriamente a instrução **break**, tal como se mostra na Figura 1.43.

A Figura 1.42 apresenta como se deve alinhar a instrução. Para aumentar a legibilidade, o bloco de execução selectivo e o bloco de execução terminal devem ser alinhados mais à direita. As sequências de instruções simples devem ser todas alinhadas, permitindo uma melhor análise das instruções que vão ser executadas.

```
switch  ( expressão )
{
   case V1  :  instrução simples;
   case V2  :  instrução simples;
               break;
   case V3  :  instrução simples;
               instrução simples;
               break;
   case V4  :  instrução simples;
               break;
   default  :  instrução simples;
}
```

Figura 1.42 - Exemplo da utilização e alinhamento da instrução **switch**.

A Figura 1.43 faz a comparação da instrução **switch** com a instrução **case**.

```
case CAR of                                    /* na linguagem Pascal */
   'a', 'e', 'o' : writeln ('Vogais ásperas');
   'i', 'u'      : writeln ('Vogais doces');
   else            writeln ('Outros símbolos gráficos')
end;
```

```
switch (CAR)                                   /* na linguagem C */
{
   case 'a':
   case 'e':
   case 'o': printf ("Vogais ásperas\n");
             break;
   case 'i':
   case 'u': printf ("Vogais doces\n");
             break;
   default : printf ("Outros símbolos gráficos\n");
}
```

Figura 1.43 - Comparação da instrução **switch** com a instrução **case**.

É preciso ter em consideração que a instrução **switch** não é tão poderosa como a instrução **case** do Turbo Pascal, uma vez que a última permite que a lista de valores enumerada possa ser constituída por um literal, ou por um conjunto de literais separados pela vírgula, ou por um intervalo de valores, ou por combinações de todas estas situações.

1.7.2 Instruções repetitivas

Tal como na linguagem Pascal, na linguagem C existem dois tipos de instruções de repetição. As instruções **while** e **do while**, cujo número de iterações é previamente desconhecido, têm uma estrutura de controlo condicional. A instrução **for**, cujo número de iterações é previamente conhecido, tem normalmente uma estrutura de controlo contadora.

1.7.2.1 As instruções repetitivas *while* e *do while*

A Figura 1.44 apresenta a definição formal das instruções **while** (enquanto fazer), e **do while** (fazer enquanto).

instrução while ::= **while (*expressão*)**
instrução simples ou composta

instrução do while ::= **do**
{
sequência de instruções simples
} while (*expressão*) ;

Figura 1.44 - Definição formal das instruções **while** e **do while**.

As instruções **while** e **do while** correspondem, respectivamente, às sintaxes **while do** e **repeat until** encontradas no Pascal. A expressão decisória deve ser de um tipo escalar básico. A expressão é falsa se for igual a zero e é verdadeira se assumir qualquer outro valor. Nestas condições, o compilador aceita qualquer expressão numérica como expressão decisória válida. Porém, por questões de clareza, isto deve ser evitado. É de bom estilo que a expressão decisória represente sempre uma expressão booleana. Ao contrário da instrução **repeat until** do Pascal, a sequência de instruções simples da instrução **do while** é colocada entre os separadores **{** e **}**.

No entanto, existe uma diferença semântica importante entre a instrução **repeat until** do Pascal e a instrução **do while** da linguagem C. Em Pascal, o ciclo repetitivo é executado até a expressão decisória de terminação ser verdadeira. Mas, na linguagem C, o ciclo repetitivo é executado enquanto a expressão decisória de terminação for verdadeira. Ou seja, em situações equivalentes, uma é necessariamente a negação da outra. Tal como em Pascal, a utilização correcta de qualquer das instruções supõe que a expressão decisória de terminação possa ser modificada, durante a execução do ciclo repetitivo, para que o seu valor passe eventualmente de verdadeiro a falso. Caso contrário, o ciclo repetitivo seria infinito. É preciso igualmente garantir que todas as variáveis que constituem a expressão decisória são previamente inicializadas.

A Figura 1.45 apresenta como se deve alinhar a instrução **while**. No caso da variante mais simples e se existir apenas uma instrução simples curta, então a instrução **while** pode ser toda escrita na mesma linha, mas se a instrução simples for longa deve ser escrita na linha seguinte mais alinhada para a direita. No caso da variante, em que, o corpo do ciclo repetitivo é constituído por uma instrução composta, os separadores **{** e **}** que definem a instrução composta devem ser alinhadas pela palavra reservada **while**, e a sequência de instruções simples são escritas, uma por linha, todas alinhadas mais à direita de maneira a que seja legível onde começa e acaba o ciclo repetitivo.

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

```
while  ( expressão )
{
   instrução simples;
   ...
   instrução simples;
}
```

Figura 1.45 - Alinhamento da instrução **while**.

A Figura 1.46 apresenta como se deve alinhar a instrução **do while**. As palavras reservadas **do** e **while** devem ser alinhadas e a condição booleana de terminação deve ser escrita à frente do terminador **while**. As instruções que constituem o corpo do ciclo repetitivo são escritas uma por linha e todas alinhadas mais à direita de maneira a que seja legível onde começa e acaba o ciclo repetitivo.

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

Figura 1.46 - Alinhamento da instrução **do while**.

A Figura 1.47 faz a comparação dos ciclos repetitivos **do while** e **while**, para calcular a média de um número indeterminado de números lidos do teclado, sendo que, a leitura termina quando é lido o valor zero. No caso do ciclo repetitivo **do while**, as instruções constituintes do corpo do ciclo repetitivo, contagem do número útil de números lidos e sua soma, necessitam de ser protegidas quando é lido o valor de terminação, para não provocar um cálculo erróneo.

```
SOMA = 0.0;                /* cálculo da média com o ciclo do while */
do
{
   printf ("Introduza um numero? ");
   scanf ("%lf", &NUMERO);
   if  (NUMERO != 0.0)
   {
          SOMA += NUMERO;
          N++;
   }
} while  (NUMERO != 0.0);
```

```
SOMA = 0.0;                    /* cálculo da média com o ciclo while */
printf ("Introduza um numero? ");
scanf ("%lf", &NUMERO);
while  (NUMERO != 0.0)
{
   SOMA += NUMERO;
   N++;
   printf ("Introduza um numero? ");
   scanf ("%lf", &NUMERO);
}
```

Figura 1.47 - Exemplo comparativo da utilização dos ciclos repetitivos **do while** e **while**.

1.7.2.2 A instrução repetitiva *for*

A Figura 1.48 apresenta a definição formal da instrução repetitiva **for** (para fazer enquanto), cujo número de iterações é previamente conhecido. A instrução **for** da linguagem C constitui uma superinstrução que não tem correspondência em mais nenhuma linguagem.

A parte da inicialização é executada em primeiro lugar e uma só vez. Em geral, a sua função é atribuir valores iniciais a uma ou mais variáveis usadas no ciclo repetitivo. A parte da terminação é uma expressão que é calculada antes do início de cada nova iteração e que determina a continuação, se for verdadeira, ou não, se for falsa, do processo repetitivo.

A expressão deve ser de um tipo escalar básico. A expressão é falsa se for igual a zero e é verdadeira se assumir qualquer outro valor. Nestas condições, o compilador aceita qualquer expressão numérica como expressão decisória válida. Porém, por questões de clareza, isto deve ser evitado. É de bom estilo que a expressão decisória represente sempre uma expressão booleana. Finalmente, a parte de actualização é executada no fim de cada iteração. Em geral, a sua função é actualizar os valores de uma ou mais variáveis usadas no ciclo repetitivo, entre as quais, sempre a variável contadora. Normalmente, as expressões de actualização das variáveis recorrem aos operadores unários incremento e decremento ou ao operador de atribuição precedido por um qualquer operador binário.

instrução for ::= **for (** *inicialização* **;** *terminação* **;** *actualização* **)**
instrução simples ou composta

inicialização ::= identificador de variável **=** *expressão* |
inicialização **,** identificador de variável **=** *expressão*

terminação ::= *expressão*

actualização ::= identificador de variável **operador binário=** *expressão* |
operador unário inc_dec identificador de variável |
identificador de variável *operador unário inc_dec* |
actualização **,** identificador de variável **operador binário=** *expressão* |
actualização **,** *operador unário inc_dec* identificador de variável |
actualização **,** identificador de variável *operador unário inc_dec*

operador unário inc_dec ::= operador unário **++** | operador unário **−−**

Figura 1.48 - Definição formal da instrução **for**.

A instrução **for** deve ser alinhada da mesma maneira que a instrução **while**. A Figura 1.49 faz a comparação da instrução **for** do Pascal e do C usando como exemplo, o cálculo das primeiras N potências de 2.

```
POT := 1;                                          /* na linguagem Pascal */
for  I := 1  to  N  do
begin
   writeln ('Potencia de 2 = ', POT:10);
   POT := POT * 2
end;
```

```
for  (POT = 1, I = 1 ; I <= N ; I++, POT *= 2)   /* na linguagem C */
   printf ("Potência de 2 = %10d\n", POT);
```

Figura 1.49 - Exemplo do cálculo das potências de 2 com um ciclo repetitivo **for**.

Qualquer uma das três partes componentes da instrução **for**, a inicialização, a terminação ou a actualização pode ser omitida. Situações especiais correspondem aos seguintes casos:

- Quando todos os elementos são omitidos, ou seja, **for (; ;)**, temos um ciclo repetitivo infinito.

- Quando a inicialização e a actualização são omitidas, ou seja, **for (;** terminação **;)**, temos um ciclo repetitivo funcionalmente equivalente ao ciclo repetitivo **while**.

Ao contrário do que se passa em Pascal, a variável contadora tem um valor bem definido após o esgotamento do ciclo repetitivo, que será aquele que conduziu a um valor falso da expressão de terminação.

Devido à sua grande versatilidade, a instrução for deve ser utilizada com extremo cuidado. É de bom estilo usá-la apenas como uma generalização da instrução **for** de Pascal. O que significa, nomeadamente, que o valor da variável contadora nunca deve ser modificado dentro do ciclo repetitivo.

Um dos erros mais frequentemente cometido por programadores que se estão a iniciar na utilização da linguagem C, consiste em esquecer que o ciclo repetitivo **for** actua enquanto a expressão de terminação for verdadeira. Se no exemplo anterior fosse utilizada a condição I == N, o ciclo repetitivo não seria executado uma única vez, a não ser que N fosse 1, quando se pretende que o ciclo repetitivo seja executado N vezes.

Os ciclos repetitivos podem ser encadeados, tal como a instrução condicional binária, dando origem a estruturas repetitivas em que, uma instrução do corpo do ciclo repetitivo é ela mesmo um ciclo repetitivo. A este tipo de construção dá-se o nome de **ciclos imbricados** (*nested loops*).

1.7.2.3 Instruções *nula*, *break* e *continue*

Há ainda três instruções que são muitas vezes usadas em conjunto com as instruções repetitivas. São elas a **instrução nula** ou **muda**, a instrução **break** e a instrução **continue**.

A instrução nula, expressa pela colocação do separador **;**, surge muitas vezes associada com as instruções **while** e **for** quando se pretende que o corpo do ciclo repetitivo não tenha qualquer instrução. A Figura 1.50 apresenta um exemplo que determina o comprimento de uma cadeia de caracteres. Uma vez que uma cadeia de caracteres é um agregado de caracteres terminado obrigatoriamente com o carácter '\0', então é preciso detectar o índice do agregado onde ele está armazenado. O que é possível fazer através de uma instrução repetitiva em que a expressão de teste, recorrendo ao operador unário incremento, provoca o deslocamento dentro do agregado e, portanto, o ciclo repetitivo não carece de qualquer instrução.

```
COMP = -1;
while (s[++comp] != '\0') ;
                /* ou em alternativa com o ciclo for */
for (comp = 0; s[comp] != '\0'; comp++) ;
```

Figura 1.50 - Cálculo do comprimento de uma cadeia de caracteres.

A instrução **break** é uma instrução que, genericamente, permite a saída intempestiva do bloco que está a ser executado. A sua utilização no caso da instrução de decisão múltipla **switch** possibilitou, como se viu, transformar operacionalmente esta instrução e adequá-la ao tipo de funcionalidade encontrado na instrução **case** do Pascal. A sua aplicação em conjunção com as instruções **while** e **do while** é também muito interessante, já que possibilita em muitas situações práticas reduzir a complexidade da expressão de terminação, aumentando por isso a clareza e a legibilidade do código correspondente.

A Figura 1.51 apresenta um exemplo em que se pretende obter a soma de um máximo de 10 valores não negativos lidos do dispositivo de entrada, sendo que a leitura parará mais cedo se for lido um valor negativo. Compare as soluções e repare na simplificação da condição de terminação da versão em linguagem C, devido à utilização da instrução **break**.

```
N := 0;                                              /* na linguagem Pascal */
SOMA := 0.0;
readln (NUMERO);
while (N < 10) and (NUMERO >= 0.0) do
begin
  N := N + 1;
  SOMA := SOMA + NUMERO;
  readln (NUMERO)
end;
```

```
N = 0;                                               /* na linguagem C */
SOMA = 0.0;
scanf ("%lf", &NUMERO);
while (N < 10)
{
   if (NUMERO < 0.0) break;
   N++;
   SOMA += NUMERO;
   scanf ("%lf", &NUMERO);
}
```

Figura 1.51 - Exemplo da utilização da instrução **break**.

A instrução **continue** é uma instrução que permite interromper a execução do ciclo repetitivo, forçando o fim da iteração presente. A sua aplicação em conjunção com a instrução **for** possibilita em muitas situações práticas reduzir a complexidade do ciclo repetitivo, aumentando por isso também a clareza e a legibilidade do código correspondente.

A Figura 1.52 apresenta um exemplo em que se pretende contar o número de caracteres alfabéticos de uma linha de texto e converter as vogais aí existentes de minúsculas para maiúsculas e vice-versa. O exemplo apresentado a seguir socorre-se das rotinas de manipulação de caracteres e de cadeias de caracteres das bibliotecas de execução ANSI e supõe, por isso, a inclusão dos ficheiros de interface *ctype.h* e *string.h*.

```
N := 0;                                              /* na linguagem Pascal */
for I := 1 to length (LINHA) do
  if LINHA[I] in ['A'..'Z','a'..'z']
  then begin
         N := N + 1;
         if LINHA[I] in ['A','E','I','O','U','a','e','i','o','u']
         then if LINHA[I] in ['A','E','I','O','U']
              then LINHA[I] := chr(ord(LINHA[I])-ord('A')+ord('a'))
              else LINHA[I] := chr(ord(LINHA[I])-ord('a')+ord('A'))
  end;
```

```
for (I = 0, N = 0; I < strlen (LINHA); I++)          /* na linguagem C */
{
   if (!isalpha (LINHA[I])) continue;
   N++;
   if ((toupper(LINHA[I]) != 'A') && (toupper(LINHA[I]) != 'E') &&
       (toupper(LINHA[I]) != 'I') && (toupper(LINHA[I]) != 'O') &&
       (toupper(LINHA[I]) != 'U')) continue;
   if (isupper (LINHA[I])) LINHA[I] = LINHA[I]-'A'+'a';
   else LINHA[I] = LINHA[I]-'a'+'A';
}
```

Figura 1.52 - Exemplo da utilização da instrução **continue**.

Compare as soluções e repare que as instruções condicionais na versão em linguagem C estão simplificadas, uma vez que não necessitam de ser encadeadas, para evitar as situações em que os caracteres não precisam de ser convertidos. Se o carácter não é um carácter alfabético, ou se é um carácter alfabético, mas não é uma vogal, então força-se a próxima iteração do ciclo repetitivo, usando para esse efeito a instrução **continue**.

1.7.2.4 Ciclos repetitivos infinitos

Um ciclo repetitivo infinito é um ciclo repetitivo que não tem condição de terminação, ou cuja condição de terminação está mal implementada. A maior parte desses ciclos repetitivos são criados por engano. Mas, existem situações em que se pretende de facto implementar um ciclo repetitivo infinito. Por exemplo, quando se pretende uma aplicação que repete sistematicamente uma determinada operação até que uma condição particular se verifica. Nessa situação é mais fácil construir um ciclo repetitivo, que é terminado através do recurso à instrução **break**. A Figura 1.53 apresenta as duas formas mais simples de implementar ciclos repetitivos infinitos.

```
while ( 1 )                                          /* com o ciclo while */
  instrução simples ou composta
-------------------------------------------------------------------------------
for  ( ; ; )                                           /* com o ciclo for */
  instrução simples ou composta
```

Figura 1.53 - Ciclos repetitivos infinitos.

1.8 Entrada e saída de dados

Tal como em qualquer outra linguagem, as instruções de entrada e de saída da linguagem C são instruções especiais que estabelecem uma interface com o sistema operativo na comunicação com os diferentes dispositivos do sistema computacional, em particular com os dispositivos convencionais de entrada e de saída. Normalmente, o dispositivo de entrada é o teclado e é mencionado pelo identificador *stdin*, e o dispositivo de saída é o monitor e é mencionado pelo identificador *stdout*.

Os dispositivos de entrada e de saída funcionam da seguinte forma. Sempre que se prime uma tecla do teclado, é gerada uma palavra binária, representando o valor atribuído ao carácter correspondente, em código ASCII por exemplo, que é armazenada num registo do controlador do teclado. O sistema operativo é alertado para esse facto e, eventualmente, vai ler o conteúdo do registo e armazenar sequencialmente o valor lido numa região da memória principal que se designa por **armazenamento tampão de entrada** (*input buffer*). O valor é também quase sempre colocado no armazenamento tampão de saída do monitor, para posterior envio para o monitor, produzindo-se assim a interacção visual com o utilizador. Em termos da instrução de entrada, o armazenamento tampão de entrada é visto como uma sequência ordenada de caracteres, organizada em linhas que consistem em zero ou mais caracteres, com representação gráfica, ou com funções de controlo, seguidos do carácter de *fim de linha* que é o '\n'. A esta sequência de caracteres dá-se o nome de fluxo de caracteres (*text stream*). De um modo geral, só linhas completas presentes no fluxo de caracteres de entrada estão disponíveis para leitura. À medida que os caracteres vão sendo processados pela instrução de entrada, vão sendo retirados do fluxo de caracteres de entrada.

Por outro lado, o fluxo de texto de saída é implementado numa região da memória principal, directamente acessível pelo sistema operativo, que se designa por **armazenamento tampão de saída** (*output buffer*). Após a escrita de uma sequência de caracteres pela instrução de saída, o sistema operativo é alertado para esse facto e, eventualmente, vai transferir essa sequência, carácter a carácter, para um registo do controlador do monitor. Em resultado disso, a mensagem associada é reproduzida no monitor a partir da posição actual do cursor e da esquerda para a direita. Além de caracteres com representação gráfica, as sequências escritas podem conter caracteres de controlo diversos que possibilitam, entre outras acções mais ou menos específicas, o deslocamento do cursor para uma outra posição do monitor.

Na linguagem C, as instruções de entrada e de saída são funções de tipo ***int*** que pertencem à biblioteca de execução ANSI e cuja descrição está contida no ficheiro de interface *stdio.h*.

1.8.1 A função *scanf*

Na linguagem C, a entrada de dados é implementada pela função **scanf** cuja sintaxe se apresenta na Figura 1.54. A função **scanf** lê sequências de caracteres do fluxo de caracteres de entrada (*stdin*) e processa-as segundo as regras impostas pelo formato de leitura, armazenando sucessivamente os valores convertidos nas variáveis, cuja localização é indicada na lista de ponteiros de variáveis.

Salvo em duas situações especiais adiante referidas, deve existir uma relação de um para um entre cada especificador de conversão e cada variável da lista de ponteiros de variáveis. Se o número de variáveis da lista de ponteiros de variáveis for insuficiente, o resultado da operação não está definido. Se, ao contrário, o número de variáveis for demasiado grande, as variáveis em excesso não são afectadas. O tipo da variável e o especificador de conversão devem ser compatíveis, já que a finalidade deste último é indicar, em cada caso, que tipos de sequências de caracteres são admissíveis e como devem ser tratadas. Quando o especificador de conversão não é válido, o resultado da operação não está definido.

O processo de leitura só termina quando o formato de leitura se esgota, é lido o carácter de *fim de ficheiro*, ou existe um conflito de tipo entre o que está indicado no formato de leitura e a correspondente quantidade a ser lida. Neste último caso, o carácter que causou o conflito é mantido no fluxo de caracteres de entrada. A função devolve o número de valores lidos e armazenados, ou o valor *fim de ficheiro* (EOF), se o carácter de *fim de ficheiro* é lido antes que qualquer conversão tenha lugar. Se, entretanto, ocorreu um conflito, o valor devolvido corresponde ao número de valores lidos e armazenados até à ocorrência do conflito.

Pondo de parte o facto de se tratar de uma instrução de leitura formatada, **scanf** é semanticamente equivalente à instrução **read** de Pascal. Não existe, contudo, na linguagem C um equivalente directo para a instrução **readln**. O papel de um carácter separador no formato de leitura é forçar a eliminação do fluxo de caracteres de entrada de todos os caracteres deste tipo até à primeira ocorrência de um carácter distinto. O papel de um literal no formato de leitura é estabelecer a correspondência com sequências idênticas de caracteres existentes no fluxo de caracteres de entrada, e, por consequência, eliminá-las do processo de leitura. A correspondência com o carácter **%**, que não é um literal, é obtida através do especificador de conversão **%%** e não há lugar a qualquer armazenamento. O supressor de atribuição, que é o carácter *****, possibilita que uma sequência de caracteres seja lida e convertida, mas o seu valor não seja armazenado em qualquer variável.

int scanf **(** *formato de leitura* **,** *lista de ponteiros de variáveis* **)**

formato de leitura ::= **"***cadeia de caracteres de definição***"**

cadeia de caracteres de definição ::= *especificador de conversão* | *carácter separador* | *literal* |
cadeia de caracteres de definição *especificador de conversão* |
cadeia de caracteres de definição *carácter separador* |
cadeia de caracteres de definição *literal*

especificador de conversão ::= **%***carácter de conversão* |
%*modificador de conversão* *carácter de conversão*

carácter de conversão ::=
i ‖ lê uma quantidade inteira genérica
d ‖ lê uma quantidade inteira em decimal (***signed***)
u ‖ lê uma quantidade inteira em decimal (***unsigned***)
o ‖ lê uma quantidade inteira em octal (***unsigned***)
x ‖ lê uma quantidade inteira em hexadecimal (**unsigned**)
f ‖ e ‖ g ‖ lê uma quantidade real em decimal
c ‖ lê caracteres
s ‖ lê uma cadeia de caracteres
[lista de caracteres**]** ‖ lê uma cadeia de caracteres imposta pela lista de caracteres. Se o primeiro carácter for o **^** então lê uma cadeia de caracteres até encontrar um carácter da lista de caracteres
p ‖ lê o valor de um ponteiro para **void**
n ‖ permite contar o número de caracteres lidos até ao seu aparecimento na cadeia de caracteres de definição

modificador de conversão ::= *largura máxima de campo* | *especificador de dimensão* |
largura máxima de campo *especificador de dimensão* |
supressor de atribuição |
supressor de atribuição *largura máxima de campo* |
supressor de atribuição *especificador de dimensão* |
supressor de atribuição *largura máxima de campo* *especificador de dimensão*

largura máxima de campo ::= valor decimal positivo que indica o número máximo de caracteres a serem lidos

especificador de dimensão ::=
h ‖ com i, d, u, o, x, indica tipo **short**
l ‖ com i, d, u, o, x, indica tipo **long**
‖ com f, e, g, indica tipo **double**
L ‖ com f, e, g, indica tipo **long double**

supressor de atribuição ::= *****

carácter separador ::= carácter espaço, carácter *tabulador* '\t' e carácter *fim de linha* '\n'

literal ::= um ou mais caracteres diferentes do *carácter separador* ou do carácter **%**

lista de ponteiros de variáveis ::= *ponteiro de variável de tipo aritmético* |
ponteiro de variável de tipo ***void*** |
lista de ponteiros de variáveis **,** *ponteiro de variável de tipo aritmético* |
lista de ponteiros de variáveis **,** *ponteiro de variável de tipo* ***void***

ponteiro de variável de tipo aritmético ::= **&** variável de tipo aritmético

ponteiro de variável de tipo void ::= **&** variável de tipo ***void***

Figura 1.54 - Definição formal da função **scanf**.

Na função **scanf** as variáveis são parâmetros de entrada-saída, pelo que, têm de ser passadas à função por referência, ou seja, por endereço. Para passar o endereço de uma variável de tipo simples, temos que preceder a variável pelo operador endereço que é o carácter **&**. No caso das variáveis de tipo agregado, o próprio nome da variável representa o endereço do elemento inicial da variável, e portanto, são usados no **scanf** normalmente.

1.8.1.1 Leitura de caracteres

Para a leitura de um carácter, o especificador de conversão deve ser **%c** ou **%1c**. Para a leitura de mais do que um carácter, o número de caracteres a ser lido será igual à largura máxima de campo e será necessário garantir que o ponteiro da variável referencia uma região de memória com capacidade para o armazenamento dos caracteres lidos, tipicamente, um agregado de caracteres.

A Figura 1.55 apresenta um exemplo de leitura de um carácter para a variável CAR e de oito caracteres para um agregado de caracteres. A declaração **char** ARRAYCAR[8] define um agregado de 8 caracteres, sendo que o índice do primeiro elemento do agregado é o índice zero. A invocação da função, para uma linha de dados introduzida pelo teclado e constituída pelos caracteres a1234567890'\n', resulta no armazenamento do carácter 'a' na variável CAR e dos caracteres '1' a '8' nos elementos sucessivos do agregado ARRAYCAR. Para passar o endereço da variável CAR usa-se o operador &. Para indicar o início do agregado de caracteres ARRAYCAR, ou se menciona apenas o nome do agregado, ou se indica o endereço do primeiro elemento do agregado &ARRAYCAR[0].

```
char CAR, ARRAYCAR[8];
...
scanf ("%c%8c", &CAR, ARRAYCAR);
        /* ou em alternativa scanf ("%c%8c", &CAR, &ARRAYCAR[0]); */
```

Figura 1.55 - Exemplo da leitura de um carácter.

A Figura 1.56 apresenta um excerto de código em que se utiliza o **scanf** para ler uma variável de tipo carácter, mas, de forma equivalente ao **readln** do Pascal.

```
#include  <stdio.h>
int main (void)
{
  char CAR,                                /* escolha da opção do menu */
       CARX;        /* variável auxiliar para teste de fim de linha */
  ...
  do
  {
    printf ("1 - Opção_1\n");
    printf ("2 - Opção_2\n");
    printf ("3 - Opção_3\n");
    printf ("Qual é a sua escolha? ");
    scanf ("%c", &CAR);    /* ler o carácter que representa a opção */
    if (CAR != '\n')
       do      /* descartar todos os eventuais restantes caracteres */
       {             /* da linha, incluindo o carácter fim de linha */
          scanf ("%c", &CARX);
       } while (CARX != '\n');
  } while ((CAR < '1') || (CAR > '3'));
  ...
}
```

Figura 1.56 - Leitura de um carácter equivalente à instrução **readln** do Pascal.

Depois da leitura da variável de tipo carácter, caso tenha sido lido um carácter diferente de *fim de linha* então vão ser lidos todos os caracteres que eventualmente existam no armazenamento tampão de entrada até à leitura do carácter *fim de linha* inclusive, usando para o efeito uma variável auxiliar de tipo carácter.

1.8.1.2 Leitura de uma cadeia de caracteres

Uma cadeia de caracteres (*string*) é tipicamente um agregado de caracteres, terminado com o carácter especial '\0'. Este carácter especial serve de indicador de fim da cadeia de caracteres, e obviamente, ocupa uma posição de armazenamento do agregado. Pelo que, se necessitarmos de armazenar uma cadeia de caracteres com MAX_CAR caracteres, devemos definir um agregado de caracteres com o tamanho MAX_CAR+1. O subdimensionamento de uma cadeia de caracteres é um dos erros mais frequentemente cometido por programadores que se estão a iniciar na utilização da linguagem C.

A leitura de mais do que um carácter usando o especificador de conversão **%c** não é muito prática, pelo que, é preferível efectuá-la como a leitura de uma cadeia de caracteres. Para a leitura de uma sequência de caracteres que não contenha um carácter separador como elemento constituinte, no fundo, para a leitura de uma palavra, deve-se usar o seguinte especificador de conversão.

%*número máximo de caracteres a ler***s**

O processo de leitura elimina todas as instanciações do carácter separador que surjam no princípio e agrupa os caracteres seguintes até ao número máximo de caracteres indicados para a leitura, ou até à ocorrência de um carácter separador, o que significa que não é possível ler-se uma cadeia de caracteres nula.

O ponteiro da variável correspondente tem que referenciar uma região de memória com capacidade para o armazenamento dos caracteres lidos e do carácter '\0', que é automaticamente colocado no fim.

Assim, é fundamental incluir sempre no especificador de conversão a largura máxima de campo, caso contrário, corre-se o risco de, se a palavra for demasiado longa, ultrapassar-se a região de armazenamento estabelecida e afectar indevidamente o valor de outras variáveis. Note-se que o valor do número máximo de caracteres a ler tem que ser, pelo menos, inferior a uma unidade ao comprimento do agregado, para garantir que o carácter terminador da cadeia de caracteres é sempre armazenado.

A Figura 1.55 apresenta um exemplo de leitura de uma cadeia de caracteres, com capacidade para armazenar 7 caracteres úteis. A invocação da função para uma linha de dados introduzida pelo teclado e constituída pelos caracteres joana'\n', resulta no armazenamento dos caracteres 'j', 'o', ... , 'a', '\0' no agregado NOME. No caso de uma linha de dados constituída pelos caracteres universidade'\n', resulta no armazenamento dos caracteres 'u', 'n', ... , 's', '\0' no agregado NOME.

```
char NOME[8];
...
scanf ("%7s", NOME);/* ou em alternativa scanf ("%7s", &NOME[0]); */
```

Figura 1.57 - Exemplo da leitura de uma cadeia de caracteres com o especificador de conversão **%s**.

Alternativamente, a leitura de uma sequência de caracteres pode ser realizada dividindo dicotomicamente os caracteres em duas classes, a classe de caracteres admissíveis para a sequência e a classe de caracteres não admissíveis, e descrevendo uma delas no especificador de conversão **%[**lista de caracteres**]**, nas suas duas variantes que se apresentam a seguir. Entre parênteses rectos descreve-se extensivamente numa ordem qualquer os caracteres pertencentes à classe de caracteres admissíveis ou alternativamente os caracteres pertencentes à classe de caracteres não admissíveis precedidos pelo carácter **^**.

%*número máximo de caracteres a ler* **[***classe de caracteres admissíveis***]**
%*número máximo de caracteres a ler* **[^***classe de caracteres não admissíveis***]**

O processo de leitura termina quando for lido o número máximo de caracteres a ler ou ocorrer um conflito. Se o conflito acontecer logo no princípio, a variável correspondente não é afectada, o que significa que não é possível ler-se uma cadeia de caracteres nula. De novo, o ponteiro da variável correspondente tem que referenciar uma região de memória com capacidade para o armazenamento dos caracteres lidos e do carácter '\0', que é automaticamente colocado no fim. Pelas razões apontadas anteriormente, a largura máxima de campo deverá também ser sempre incluída no especificador de conversão.

A Figura 1.58 apresenta dois exemplos da utilização do especificador de conversão **%[ ]** para a leitura de cadeias de caracteres. No primeiro caso pretende-se ler um número de telefone, que é uma sequência de 9 caracteres exclusivamente composta por caracteres numéricos. No segundo exemplo, pretende-se efectuar a leitura de uma sequência de caracteres quaisquer até ao aparecimento do carácter *fim de linha*, no máximo de 79 caracteres.

```
char NUMTEL[10];
...
scanf ("%9[0123456789]", NUMTEL);
       /* ou em alternativa scanf ("%9[0123456789]", &NUMTEL[0]); */
```

```
char FRASE[80];
...
scanf ("%79[^\n]", FRASE);
              /* ou em alternativa scanf ("%79[^\n]", &FRASE[0]); */
```

Figura 1.58 - Exemplos da leitura de cadeias de caracteres com o especificador de conversão **%[ ]**.

Um dos erros mais frequentes dos programadores que se estão a iniciar na utilização da linguagem C, consiste em usar simultaneamente os dois especificadores de conversão de cadeia de caracteres, inventando o especificador de conversão **%s[ ]**. Estes dois especificadores de conversão são alternativos, e, portanto, não podem ser combinados.

A Figura 1.59 apresenta um excerto de código em que se utiliza o **scanf** para ler uma variável de tipo cadeia de caracteres, mas, de forma equivalente ao **readln** do Pascal. A função é invocada para ler uma sequência de caracteres quaisquer, constituída no máximo por 79 caracteres. Depois descartam-se todos os caracteres que eventualmente existam no armazenamento tampão de entrada até à leitura do carácter *fim de linha*. De seguida lê-se e descarta-se o carácter *fim de linha*. Estas duas acções não podem ser combinadas numa só, porque caso não existam caracteres extras, então a instrução de leitura terminaria abruptamente sem efectuar a leitura do carácter *fim de linha*, que ficaria no armazenamento tampão de entrada disponível para posteriores invocações da função. Caso não tenha sido lido qualquer carácter para a variável FRASE, o que pode ser indagado aferindo o resultado devolvido pelo **scanf** e armazenado na variável T, que é zero nesse caso, então constrói-se

uma cadeia de caracteres nula, colocando o carácter terminador na primeira posição da cadeia de caracteres.

```
#include  <stdio.h>

int main (void)
{
  char FRASE[80];                                  /* frase a ser lida */
  int T;                           /* sinalização do estado de leitura */
  ...
  printf ("Escreva a frase: ");
  T = scanf ("%79[^\n]", FRASE);                   /* leitura da frase */
  scanf ("%*[^\n]");          /* descartar todos os outros caracteres */
  scanf ("%*c");                 /* descartar o carácter fim de linha */
                         /* construir uma cadeia de caracteres nula */
  if (T == 0) FRASE[0] = '\0';
  ...
}
```

Figura 1.59 - Leitura de uma cadeia de caracteres equivalente à instrução **readln** do Pascal.

1.8.1.3 Leitura de valores numéricos

Na leitura de valores numéricos, a introdução do especificador de dimensão no especificador de conversão permite atribuir os valores lidos a variáveis de tipo ***short*** ou de tipo ***long***, em vez de tipo ***int***, ou a variáveis de tipo ***double*** ou de tipo ***long double***, em vez de tipo ***float***. O recurso à largura máxima de campo só deve ser efectuado quando se pretende ler valores com um formato bem definido. Caso contrário, não deve ser usado para possibilitar uma leitura em formato livre. Se os valores lidos forem demasiado grandes para poderem ser armazenados nas variáveis que lhes estão associadas, vai ocorrer *overflow*.

O processo de leitura termina quando for lido um número de caracteres válidos igual à largura máxima de campo, caso ela tenha sido especificada, surgir entretanto um carácter separador, ou qualquer outro carácter não válido, que funciona neste caso também como carácter separador, ou ocorrer um conflito. Nesta última situação, a variável correspondente não é afectada. A Figura 1.60 apresenta as sequências de caracteres admissíveis para cada carácter de conversão.

Carácter de conversão	Sequência admissível	Observação
	tipos inteiros	
i (tipo **signed**)	[+/-]#...#	# - algarismo decimal
	[+/-]0x#...#	# - algarismo hexadecimal
	[+/-]0#...#	# - algarismo octal
d (tipo **signed**)	[+/-]#...#	# - algarismo decimal
u (tipo **unsigned**)	[+/-]#...#	# - algarismo decimal
x (tipo **unsigned**)	[+/-]0x#...#	# - algarismo hexadecimal
o (tipo **unsigned**)	[+/-]0#...#	# - algarismo octal
	tipos reais	
f,e,g	[+/-]#...#	# - algarismo decimal
	[+/-][#...#][.]#...#	
	[+/-][#...#][.]#...#[e/E[+/-]#...#]	

Figura 1.60 - Tabela de sequências admissíveis em função do carácter de conversão.

A Figura 1.61 apresenta alguns exemplos de leituras de valores numéricos. A execução da função para a linha de dados introduzida pelo teclado e constituída pelos seguintes caracteres -123 0xF2'\t'-1.47e1'\n', resulta no armazenamento das quantidades -123 na variável I, 242 na variável S e -14.7 na variável D. No caso de uma linha de dados constituída pelos caracteres '\t'12 -023 e-4'\n', resulta no armazenamento das quantidades 12 na variável I, -19 na variável S e nenhum valor na variável D, porque ocorreu um conflito. O conflito deve-se ao facto de que a sequência e-4 não ser uma sequência de caracteres admissível para representar uma quantidade real em notação científica. Para corrigir este conflito deve-se usar a sequência de caracteres 1e-4.

```
int I;
short S;
double D;
...
scanf ("%d%hi%lf", &I, &S, &D);
```

Figura 1.61 - Exemplo de leitura de valores numéricos.

O uso do carácter separador no formato de leitura deve ser evitado porque, ou é irrelevante, quando colocado no princípio, ou entre especificadores de conversão, já que o carácter separador funciona como elemento de separação de sequências numéricas, ou tem efeitos colaterais nocivos, quando colocado no fim, já que obriga à detecção de um carácter distinto no fluxo de texto de entrada, antes da conclusão da instrução. Procure descobrir o que vai acontecer quando forem executadas as funções **scanf** apresentadas na Figura 1.62, para as duas linhas de dados introduzidas pelo teclado e constituídas pelos seguintes caracteres 1 2'\n' e 3'\n'.

```
int A, B, C;
...
printf ("Valores 1 e 2? ");
scanf ("%d%d\n", &A, &B);
printf ("Valor 3? ");
scanf ("%d", &C);
```

Figura 1.62 - Exemplo incorrecto de leitura de valores numéricos.

Por causa do literal '\n' no fim da cadeia de caracteres de definição, a primeira invocação da função **scanf** só termina quando se tecla o carácter 3, pelo que, a mensagem da segunda invocação da função **printf** só é escrita no monitor depois de introduzida a segunda linha de dados, quando deveria aparecer antes.

A Figura 1.63 apresenta um excerto de código em que se utiliza o **scanf** para ler uma variável de tipo numérica, mas, de forma equivalente ao **readln** do Pascal. A função é invocada para ler uma variável numérica, que neste caso é inteira de tipo ***int***. Depois descartam-se todos os caracteres que eventualmente existam no armazenamento tampão de entrada até à leitura do carácter *fim de linha*. De seguida lê-se e descarta-se o carácter *fim de linha*. Estas duas acções não podem ser combinadas numa só, porque caso não existam caracteres extras, então a instrução de leitura terminaria abruptamente sem efectuar a leitura do carácter *fim de linha*, que ficaria no armazenamento tampão de entrada disponível para posteriores invocações da função. Caso não tenha sido lido qualquer valor numérico para a variável VAL, o que pode ser indagado aferindo o resultado devolvido pelo **scanf** e armazenado na variável T, que é zero nesse caso, então repete-se o processo até que seja efectivamente lido um valor numérico.

```
#include  <stdio.h>

int main (void)
{
  int VAL;                                          /* valor a ser lido */
  int T;                          /* sinalização do estado de leitura */
  ...
  printf ("Valor? ");
  do
  {
    T = scanf ("%d", &VAL);                          /* ler o valor */
    scanf ("%*[^\n]");      /* descartar todos os outros caracteres */
    scanf ("%*c");             /* descartar o carácter fim de linha */
  } while (T == 0);
        /* repetir enquanto um valor numérico não tiver sido lido */
  ...
}
```

Figura 1.63 - Leitura de uma variável de tipo numérica equivalente à instrução **readln** do Pascal.

A Figura 1.64 apresenta o código necessário para ler uma informação horária válida, de tipo HH:MM:SS. O processo de leitura tem que ler três quantidades numéricas positivas daí o tipo de dados utilizado ser ***unsigned***. Como os valores não excedem 23 para as horas e 60 para os minutos e segundos, então podem ser de tipo ***short***. A leitura é repetida enquanto houver um erro de formato e enquanto houver valores fora dos limites.

```
#include  <stdio.h>

int main (void)
{
  unsigned short H, M, S;  /* hora, minuto e segundo a serem lidos */
  int T;                          /* sinalização do estado de leitura */
  ...
  do
  {
    do
    {                                /* leitura da informação horária */
      printf ("Informação horária HH:MM:SS? ");
      T = scanf ("%2hu:%2hu:%2hu", &H, &M, &S);
      scanf ("%*[^\n]");    /* descartar todos os outros caracteres */
      scanf ("%*c");           /* descartar o carácter fim de linha */
    } while (T != 3);/* repetir enquanto houver um erro de formato */
  } while ((H > 23) || (M > 59) || (S > 59));
               /* repetir enquanto houver valores fora dos limites */
  ...
}
```

Figura 1.64 - Leitura de uma informação horária.

O especificador de conversão **%n** permite contar a número de caracteres lidos até ao seu aparecimento na cadeia de caracteres de definição. É utilizado quando queremos verificar se o formato dos dados está de acordo com a cadeia de caracteres de definição que se está a utilizar. A Figura 1.65 apresenta um exemplo da sua aplicação. Vamos considerar que queremos ler um valor numérico mas, que este se encontra a seguir a outro valor, e que este primeiro valor deve ser descartado porque não nos interessa. Se por algum motivo existir um erro de formato e existir apenas um valor para ser lido, esse valor é descartado e não é lido nem armazenado na variável NUM. Para detectar se há ou não um erro no formato esperado, podemos determinar o número de caracteres lidos antes da leitura da variável NUM. Se o valor de N for nulo é sinal que não havia nenhum valor inicial, e, portanto, o formato previsto dos dados de entrada não se verifica.

```
int NUM,                                          /* valor a ser lido */
    N;                                 /* contador de caracteres lidos */
...
N = 0;            /* inicialização do contador de caracteres lidos */
scanf ("Numero = %*d%n%d", &N, &NUM);
                        /* leitura do valor e utilização do contador */
```

Figura 1.65 - Exemplo da utilização do especificador de conversão **%n**.

1.8.2 A função *printf*

Na linguagem C, a saída de dados é implementada pela função **printf** cuja sintaxe se apresenta na Figura 1.66. A função **printf** escreve sucessivamente sequências de caracteres no fluxo de texto de saída (*stdout*), representativas de texto e dos valores das expressões que formam a lista de expressões, segundo as regras impostas pelo formato de escrita.

O texto é especificado no formato de escrita pela introdução de literais, que são copiados sem modificação para o fluxo de texto de saída. O modo como o valor das expressões é convertido, é descrito pelos especificadores de conversão. Deve, por isso, existir uma relação de um para um entre cada especificador de conversão e cada expressão da lista de expressões. Se o número de expressões for insuficiente, o resultado da operação não está definido. Se, ao contrário, esse número for demasiado grande, as expressões em excesso são ignoradas.

O tipo da expressão e o especificador de conversão devem ser compatíveis, já que a finalidade deste último é indicar, em cada caso, o formato da sequência convertida. Quando o especificador de conversão não é válido, o resultado da operação não está definido.

O processo de escrita só termina quando o formato de escrita se esgota, ou ocorreu um erro. A função devolve o número de caracteres escritos no fluxo de texto de saída, ou o valor -1, se ocorreu um erro.

Como o formato de escrita pode conter literais que são directamente copiados sem modificação para o fluxo de texto de saída, a instrução **printf** é muito geral, podendo ser tornada semanticamente equivalente à instrução **write** de Pascal, quando o formato de escrita não contém o carácter '\n', ou à instrução **writeln**, quando este é precisamente o último carácter presente. De um modo geral, a inclusão de literais no formato de escrita possibilita melhorar a compreensão dos dados impressos e organizar a sua apresentação de um modo mais regular. A impressão do carácter **%**, que não é um literal, é obtida através do especificador de conversão **%%**.

A largura mínima de campo indica o número mínimo de caracteres usados na representação do valor da expressão. Quando a expressão pode ser representada com menos caracteres, são introduzidos espaços ou zeros até perfazer o número indicado. Quando a largura mínima de campo é insuficiente, o campo é alargado para conter o valor convertido.

int printf **(** *formato de escrita* **,** *lista de expressões* **)**

formato de escrita ::= **"***cadeia de caracteres de definição***"**

cadeia de caracteres de definição ::= *especificador de conversão* | *literal* |
cadeia de caracteres de definição *especificador de conversão* |
cadeia de caracteres de definição *literal*

especificador de conversão ::= **%***carácter de conversão* |
%*modificador de conversão* *carácter de conversão*

carácter de conversão ::=
i ‖ d ‖ escreve uma quantidade inteira decimal (***signed***)
u ‖ escreve uma quantidade inteira em decimal (***unsigned***)
o ‖ escreve uma quantidade inteira em octal (***unsigned***)
x ‖ escreve uma quantidade inteira em hexadecimal (**unsigned**)
f ‖ escreve uma quantidade real em notação PI**.**PF
e ‖ E ‖ escreve uma quantidade real em notação científica. O carácter indicador de expoente é **e** ou **E** conforme se usa o carácter de conversão e ou E
g ‖ G ‖ escreve uma quantidade real no formato *f* ou *e* dependente do valor. O carácter indicador de expoente é **e** ou **E** conforme se usa o carácter de conversão *g* ou *G*
c ‖ escreve um caracter
s ‖ escreve uma cadeia de caracteres
p ‖ escreve o valor de um ponteiro para **void**

modificador de conversão ::= *modificador de formato* | *especificador de precisão* |
especificador de dimensão |
modificador de formato *especificador de precisão* |
modificador de formato *especificador de dimensão* |
especificador de precisão *especificador de dimensão* |
modificador de formato *especificador de precisão* *especificador de dimensão*

modificador de formato ::= *modificador de aspecto* | *largura mínima de campo* |
modificador de aspecto *largura mínima de campo*

modificador de aspecto ::= qualquer combinação de **espaço**, **−**, **+**, **#**, **0**

largura mínima de campo ::= valor decimal positivo que indica o número mínimo de caracteres a serem escritos

especificador de precisão ::= **.** | **.** valor decimal positivo

especificador de dimensão ::=
h ‖ com i, d, u, o, x, indica tipo **short**
l ‖ com i, d, u, o, x, indica tipo **long**
L ‖ com f, e, g, indica tipo **long double**

literal ::= carácter diferente de **%** | *literal* carácter diferente de **%**

lista de expressões ::= *expressão de tipo aritmético* | *expressão de tipo ponteiro para* ***void*** |
lista de expressões **,** *expressão de tipo aritmético* |
lista de expressões **,** *expressão de tipo ponteiro para* ***void***

Figura 1.66 - Definição formal da função **printf**.

O papel do modificador de aspecto é controlar a apresentação dos valores convertidos. Assim, tem-se que:

- O carácter **–**, significa justificação da sequência convertida à esquerda no campo. Por defeito, ela é justificada à direita.

- O carácter **0**, significa que quando for necessário introduzir caracteres extra para perfazer a largura mínima de campo na representação de valores numéricos, são introduzidos zeros, em vez de espaços, para uma justificação à direita.

- O carácter **+**, significa que todos os valores numéricos passam a ter sinal, os positivos, o sinal **+** e os negativos, o sinal **–**. Por defeito, só os valores negativos são representados com sinal.

- O carácter **espaço**, significa que os valores numéricos positivos passam a ser representados com um espaço na posição do sinal. Por defeito, só os valores negativos são representados com sinal.

- O carácter **#**, significa que os valores numéricos no sistema hexadecimal ou octal são representados precedidos de '0x' ou '0', respectivamente. Os valores numéricos reais são representados com o separador **.**, mesmo que não haja parte fraccionária. Além disso, quando for usado o carácter de conversão **g**, os zeros à direita na parte fraccionária são representados.

O papel do especificador de precisão está associado de alguma forma com a precisão com que os valores convertidos são expressos. Assim, tem-se que:

- Para **valores inteiros**, indica o número mínimo de algarismos usados na representação do valor. Se surgir apenas o carácter **.**, significa um número zero de algarismos.

- Para **valores reais**:

    - Com os caracteres de conversão **e** ou **f** indica o número de algarismos usados na representação da parte fraccionária. Se surgir apenas o carácter **.**, significa um número zero de algarismos. Quando não está presente, esse número é de 6.

    - Com o carácter de conversão **g** indica o número máximo de algarismos significativos. Se surgir apenas o carácter **.**, significa um número zero de algarismos.

    - Quando o valor não puder ser expresso no número de algarismos indicado pelo especificador de precisão, então o valor será arredondado.

- Para **valores de tipo cadeia de caracteres**, indica o número máximo de caracteres a imprimir.

É preciso ter em atenção que não existe uma total simetria nos formatos de conversão das funções **scanf** e **printf**. Assim, a função **scanf** exige o formato de conversão **%lf** para ler um valor ***double***, enquanto que a função **printf** usa o formato de conversão **%f** para escrever uma expressão de tipo ***double***. Esta assimetria deve-se ao facto de a linguagem C converter automaticamente expressões reais para ***double*** e portanto, não existir de facto a impressão de expressões de tipo ***float***. Esta assimetria é um dos erros mais frequentemente cometido por programadores que se estão a iniciar na utilização da linguagem C.

A Figura 1.67 apresenta alguns exemplos da aplicação de vários formatos de escrita de valores numéricos e as respectivas sequências de caracteres escritas no monitor.

```
printf ("->%12d\n", 675);                    /* ->         675 */
printf ("->%012d\n", 675);                   /* ->000000000675 */
printf ("->%12.4d\n", 675);                  /* ->        0675 */

printf ("->%12f\n", 675.95);                 /* ->  675.950000 */
printf ("->%12.4f\n", 675.95);               /* ->    675.9500 */
printf ("->%12.1f\n", 675.95);               /* ->       676.0 */

printf ("->%12.6g\n", 675.0000);             /* ->         675 */
printf ("->%#12.6g\n", 675.0000);            /* ->     675.000 */
printf ("->%12.3g\n", 675.0);                /* ->         675 */
printf ("->%12.2g\n", 675.0);                /* ->     6.8e+02 */
printf ("->%12.1g\n", 675.0);                /* ->       7e+02 */
printf ("->%#12.1g\n", 675.0);               /* ->      7.e+02 */
printf ("->%#12.1G\n", 675.0);               /* ->      7.E+02 */
```

Figura 1.67 - Exemplos de formatos de escrita.

1.8.2.1 Escrita de caracteres

Para a escrita de um carácter, deve-se utilizar o seguinte especificador de conversão.

%–*número de colunas de impressão***c**

Sendo que o modificador de aspecto **–**, é usado quando se pretender a justificação do carácter à esquerda dentro do número de colunas de impressão indicado. Não é possível escrever-se mais do que um carácter de cada vez. A expressão argumento de tipo ***int*** é convertida para tipo ***unsigned char*** e o valor obtido é impresso. A Figura 1.68 apresenta alguns exemplos de escrita do carácter *. No primeiro caso não se especifica a largura mínima de campo, pelo que, o carácter é escrito numa única coluna de impressão. Nos segundo e terceiro casos, especifica-se cinco colunas de impressão, com justificação à direita, que é a situação por defeito, e com justificação à esquerda usando o modificador de aspecto **–**.

```
printf ("->%c<-\n", '*');                        /* ->*<- */
printf ("->%5c<-\n", '*');                   /* ->    *<- */
printf ("->%-5c<-\n", '*');                  /* ->*    <- */
```

Figura 1.68 - Escrita de um carácter com vários formatos.

1.8.2.2 Escrita de cadeias de caracteres

Para a escrita de uma cadeia de caracteres, deve-se utilizar o seguinte especificador de conversão.

%– *número mínimo de colunas de impressão***.***número máximo de colunas de impressão***s**

Sendo que o modificador de aspecto **–**, é usado quando se pretender a justificação da cadeia de caracteres à esquerda dentro do número mínimo de colunas de impressão indicado. A expressão argumento deve referenciar a região de memória onde a sequência de caracteres, terminada necessariamente pelo carácter '\0', está armazenada, sendo tipicamente um agregado de caracteres. O número de caracteres a imprimir é igual ao número de caracteres da cadeia de caracteres, se o especificador de precisão não tiver sido usado, ou se tiver um valor maior ou igual ao comprimento da cadeia de caracteres, ou, alternativamente, igual ao valor do especificador de precisão, em caso contrário.

A Figura 1.69 apresenta alguns exemplos de escrita de uma cadeia de caracteres. No primeiro caso não se usa qualquer formato, pelo que, a escrita ocupa o número de colunas de impressão igual ao comprimento da cadeia de caracteres. Nos segundo e terceiro casos como o número mínimo de colunas é insuficiente, então é ignorado. No quarto caso apenas é imprimido o número de caracteres indicado pelo especificador de precisão. No quinto caso não é impresso nada, uma vez que o especificador de precisão é nulo. Nos sexto e sétimo casos usa-se um número mínimo de colunas de impressão igual ao número máximo de colunas de impressão, pelo que, a cadeia de caracteres é impressa com esse número de caracteres. Como a cadeia de caracteres tem um número de colunas inferior ao pretendido, então são acrescentados espaços à direita ou à esquerda, conforme a justificação pedida.

```
char FRASE[30] = "pim pam pum, cada bola ...";
...
printf ("->%s<-\n", FRASE);      /* ->pim pam pum, cada bola ...<- */
printf ("->%11s<-\n", FRASE);    /* ->pim pam pum, cada bola ...<- */
printf ("->%-11s<-\n", FRASE);   /* ->pim pam pum, cada bola ...<- */
printf ("->%.11s<-\n", FRASE);                  /* ->pim pam pum<- */
printf ("->%.s<-\n", FRASE);                              /* -><- */
printf ("->%8.8s<-\n", "pim");                  /* ->     pim<- */
printf ("->%-8.8s<-\n", "pim");                 /* ->pim     <- */
```

Figura 1.69 - Escrita de uma cadeia de caracteres com vários formatos.

1.8.2.3 Escrita de valores numéricos

A introdução do especificador de dimensão no especificador de conversão permite caracterizar o tipo do valor que vai ser convertido, o tipo ***short*** ou o tipo ***long*** em vez do tipo ***int***, ou o tipo ***long double*** em vez do tipo ***double***. Note-se o significado distinto atribuído ao especificador de precisão, tratando-se de valores inteiros ou de valores reais. No primeiro caso significa o número mínimo de algarismos da representação. No segundo caso significa o número de algarismos da parte fraccionária, quando são usados os caracteres de conversão *f* ou *e*, ou o número máximo de algarismos significativos, que é 6 por defeito, quando é usado o carácter de conversão *g*. Segundo a norma ANSI, a selecção do estilo de representação, quando é usado o carácter de conversão *g*, baseia-se na regra seguinte. A notação científica é utilizada, quando o expoente resultante for inferior a -4 ou maior ou igual à precisão estabelecida. Em todos os outros casos é utilizada a representação em parte inteira, a vírgula, que nos computadores é o **.** e a parte fraccionária.

A Figura 1.70 apresenta alguns exemplos da aplicação de vários formatos de escrita de valores numéricos inteiros e as respectivas sequências de caracteres escritas no monitor.

```
printf ("->%ld***%ld<-\n", 1095L, -1095L);     /* ->1095***-1095<- */
printf ("->%+li***%+li<-\n", 1095L, -1095L); /* ->+1095***-1095<- */
printf ("->% .2ld***% .2ld<-\n", 1095L, -1095L);
                                              /* -> 1095***-1095<- */
printf ("->%5.5hd***%5.hd<-\n", -13505, -13505);
                                            /* ->-13505***-13505<- */
printf ("->%05u***%5u<-\n", 17, 17);          /* ->00017***   17<- */
printf ("->%-5.2x***%#-5.2x<-\n", 17, 17);    /* ->11   ***0x11<- */
printf ("->%5.2ho***%#5.2o<-\n", 17, 17);     /* ->   21***  021<- */
```

Figura 1.70 - Escrita de valores numéricos inteiros com vários formatos.

A Figura 1.71 apresenta alguns exemplos da aplicação de vários formatos de escrita de valores numéricos reais e as respectivas sequências de caracteres escritas no monitor.

```
printf ("->%f***%e<-\n", 13.595, 13.595);
                                    /* ->13.595000***1.359500e+01<- */
printf ("->%.f***%.e<-\n", 13.595, 13.595);
                                              /* ->14***1e+01<- */
printf ("->%#.f***%#.e<-\n", 13.595, 13.595);
                                           /* ->14.***1.e+01<- */
printf ("->%g***%g<-\n", -13.595, -0.000011);
                                         /* ->-13.595***-1.1e-05<- */
printf ("->%8.2f***%10.2e<-\n", -13.345, -13.345);
                                       /* ->  -13.35*** -1.33e+01<- */
printf ("->%08.2f***%010.2e<-\n", -13.345, -13.345);
                                       /* ->-0013.35***-01.33e+01<- */
printf ("->%8.2g***%#08.2g<-\n", -13.345, -13.345);
                                        /* ->     -13***-000013.<- */
```

Figura 1.71 - Escrita de valores numéricos reais com vários formatos.

1.9 Bibliotecas de execução ANSI

Já referimos a biblioteca de execução ANSI ***stdio***, que descreve as funções de acesso aos dispositivos de entrada e de saída. Agora vamos apresentar mais algumas das bibliotecas de execução ANSI.

1.9.1 Biblioteca ctype

A biblioteca ***ctype*** contém um conjunto de funções para processamento de caracteres. Tem a particularidade de as rotinas poderem ser tornadas independentes do código usado na representação dos caracteres. Para utilizar as suas funções é preciso fazer a inclusão do ficheiro de interface *ctype.h* com a seguinte directiva do pré-processador.

#include <ctype.h>

As funções dividem-se funcionalmente em dois grupos. O grupo das **funções de teste de caracteres**, que determinam se um dado carácter pertence ou não a uma dada classe. O grupo das **funções de conversão**, que permitem a conversão de um carácter do alfabeto maiúsculo num carácter do alfabeto minúsculo e vice-versa.

As funções de teste de caracteres têm o seguinte mecanismo de comunicação com o exterior.

int nome da função (**int** carácter);

O valor devolvido é não nulo, ou seja, verdadeiro, se o carácter em argumento pertencer à classe associada e é nulo, ou seja, falso, caso contrário. Se o valor em argumento não puder ser representado como um ***unsigned int***, o resultado é indefinido.

As funções de conversão têm o seguinte mecanismo de comunicação com o exterior.

int nome da função (**int** carácter);

O valor devolvido é o do carácter em argumento, se o carácter já estiver convertido, ou não se tratar de um carácter alfabético e é o do carácter convertido, no caso contrário.

A Figura 1.72 apresenta as funções de teste de caracteres e as funções de conversão.

funções de teste de caracteres

Nome da função	Classe de pertença
isalnum	caracteres alfabéticos e algarismos decimais
isalpha	caracteres alfabéticos
iscntrl	caracteres de controlo
isdigit	algarismos decimais
isgraph	todos os caracteres com representação gráfica
islower	caracteres alfabéticos minúsculos
isprint	todos os caracteres com representação gráfica mais o carácter espaço
ispunct	todos os caracteres com representação gráfica menos os caracteres alfabéticos e os algarismos decimais
isspace	espaço, '\f', '\n', '\r', '\t' e '\v'
isupper	caracteres alfabéticos maiúsculos
isxdigit	algarismos hexadecimais

funções de conversão

Nome da função	Tipo de conversão
tolower	do alfabeto maiúsculo para o alfabeto minúsculo
toupper	do alfabeto minúsculo para o alfabeto maiúsculo

Figura 1.72 - Funções da biblioteca ***ctype***.

1.9.2 Biblioteca math

A biblioteca *math* contém um conjunto de funções matemáticas. Para utilizar as suas funções é preciso fazer a inclusão do ficheiro de interface *math.h* com a seguinte directiva do pré-processador.

#include <math.h>

Quando da invocação destas funções podem ocorrer dois tipos de erros. Erros de domínio, quando o valor do argumento está fora da gama permitida. Erros de contradomínio (ERANGE), quando o valor devolvido não pode ser representado por uma variável de tipo ***double***. Quando tal acontece, a variável global **errno** assume o valor EDOM ou ERANGE.

A norma IEEE 754, usada na caracterização dos tipos inteiros no computador utilizado na disciplina, permite a detecção destas situações por inspecção do valor devolvido. Se o erro for EDOM e o valor devolvido for **nan**, então isso significa que não é um número. Se o erro for ERANGE podemos ter os seguintes valores devolvidos: **inf** que significa que é um número positivo muito grande; **-inf** que significa que é um número negativo muito grande; e **0** que significa que é um número muito pequeno.

Durante a compilação de um programa que referencia a biblioteca *math.h*, para que o processo de ligação funcione, a biblioteca tem que ser explicitamente referenciada no comando de compilação. Para tal, é preciso acrescentar ao comando de compilação a opção de compilação **-lm**, da seguinte forma.

cc programa.c -o programa ... -lm

As funções da biblioteca matemática dividem-se funcionalmente em três grupos: **funções trigonométricas e hiperbólicas**; **funções exponenciais e logarítmicas**; e **funções diversas**.

A Figura 1.73 apresenta as funções trigonométricas e hiperbólicas.

Nome da função	Significado
`double acos (double x);`	$x \in [-1, 1] \Rightarrow$ arco coseno(x) $\in [0,\pi]$
`double asin (double x);`	$x \in [-1, 1] \Rightarrow$ arco seno(x) $\in [-\pi/2,\pi/2]$
`double atan (double x);`	arco tangente(x) $\in [-\pi/2,\pi/2]$
`double atan2 (double y, double x);`	$x \neq 0$ ou $y \neq 0 \Rightarrow$ arco tangente(y/x) $\in]-\pi,\pi]$
`double cos (double x);`	coseno(x)
`double sin (double x);`	seno(x)
`double tan (double x);`	tangente(x)
`double cosh (double x);`	$(e^x+e^{-x})/2$
`double sinh (double x);`	$(e^x-e^{-x})/2$
`double tanh (double x);`	$(e^x-e^{-x})/(e^x+e^{-x})$

Figura 1.73 - Funções trigonométricas e hiperbólicas.

A Figura 1.74 apresenta as funções exponenciais e logarítmicas.

Nome da função	Significado
`double exp (double x);`	e^x
`double frexp (double x, int *y);`	x = frexp (x,y) * 2^{*y}, com frexp (x,y) $\in]0.5,1]$
`double ldexp (double x, int y);`	x * 2^y
`double log (double x);`	$\log_e$(x), com x>0
`double log10 (double x);`	$\log_{10}$(x), com x>0
`double modf (double x, double *y);`	x = modf (x,y) + *y, com |modf (x,y)| $\in [0,1[$
`double pow (double x, double y);`	x^y, com $x \neq 0$ ou $y>0$ e $x \geq 0$ ou então y não tem parte fraccionária
`double sqrt (double x);`	$\sqrt{x}$

Figura 1.74 - Funções exponenciais e logarítmicas.

A Figura 1.75 apresenta as funções diversas.

Nome da função	Significado
`double ceil (double x);`	truncatura por excesso
`double fabs (double x);`	|x|
`double floor (double x);`	truncatura por defeito
`double fmod (double x, double y);`	resto da divisão de x por y (y≠0)

Figura 1.75 - Funções diversas.

1.9.3 Biblioteca errno

Algumas funções da biblioteca de execução ANSI, particularmente as funções matemáticas, para além de devolverem um valor específico em situações de erro, afectam uma variável global de tipo ***int***, designada por **errno**, com um código de erro. A biblioteca ***errno*** contém a alusão à variável **errno** e a definição dos códigos de erro. Para utilizar as suas funções é preciso fazer a inclusão do ficheiro de interface *errno.h* com a seguinte directiva do pré-processador.

#include <errno.h>

A sua aplicação prática vem muitas vezes associada com o envio da mensagem de erro correspondente para o dispositivo convencional de saída de erro, que normalmente é o mesmo que o dispositivo convencional de saída, ou seja, o monitor.

Isto é feito usando a função **perror**, descrita no ficheiro de interface *stdio.h*, que imprime no dispositivo convencional de saída de erro uma combinação de duas mensagens, separadas pelo carácter **:**.

mensagem definida pelo programador **:** *mensagem associada ao valor de* ***errno***

A Figura 1.76 apresenta um exemplo da utilização da função **perror**.

```
#include <stdio.h>
#include <errno.h>
...
errno = 0;
Z = log (X);
if (errno != 0)                    /* ocorreu uma situação de erro */
{
   perror ("erro no cálculo de um logaritmo na rotina ...");
   ...
}
...
```

Figura 1.76 - Exemplo da utilização da função **perror**.

1.9.4 Biblioteca stdlib

A biblioteca ***stdlib*** contém funções de utilidade geral e define quatro tipos de dados. Para utilizar as suas funções é preciso fazer a inclusão do ficheiro de interface *stdlib.h* com a seguinte directiva do pré-processador.

#**include** <stdlib.h>

As funções dividem-se funcionalmente em vários grupos. As **funções para geração de sequências pseudo-aleatórias**, que permitem fazer uma aproximação à geração de uma sequência de valores que segue uma função densidade de probabilidade uniformemente distribuída. As **funções para realização de operações inteiras elementares**, como a obtenção do valor absoluto e do quociente e do resto de divisões inteiras. As **funções para conversão de cadeias de caracteres em quantidades numéricas**.

A Figura 1.77 apresenta as funções para geração de sequências pseudo-aleatórias.

Nome da função	Significado
int rand (**void**);	rand() ∈ {0, 1, ... , RAND_MAX}
void srand (**unsigned int** s);	após a invocação desta função, sucessivas invocações de rand() produzem uma sequência de valores pseudo-aleatórios determinada por ***s***, sendo ***s*** a semente de geração

Figura 1.77 - Funções para geração de sequências pseudo-aleatórias.

A Figura 1.78 apresenta as funções para realização de operações inteiras elementares.

Nome da função	Significado
`int abs (int x);`	|x|
`long labs (long x);`	|x|
`div_t idiv (int x, int y);`	quociente e resto da divisão inteira de x por y
`ldiv_t ldiv (long x, long y);`	quociente e resto da divisão inteira de x por y

Figura 1.78 - Funções para realização de operações inteiras elementares.

Para as duas últimas funções, os valores do quociente e do resto da divisão de operandos de tipo ***int*** ou de tipo ***long*** são devolvidos conjuntamente numa variável de tipo complexo, *div_t* ou *ldiv_t*, assim definida para o primeiro caso.

```
typedef struct
        {
          int quot;    /* quociente */
          int rem;     /* resto */
        } div_t;
```

A definição do tipo *ldiv_t* é formalmente semelhante, substitui-se apenas ***int*** por ***long*** em todas as ocorrências.

A Figura 1.79 apresenta as funções para conversão de cadeias de caracteres em quantidades numéricas.

Nome da função	Significado
`double atof (const char *str);`	converte a cadeia de caracteres apontada por str numa quantidade real de tipo ***double***
`int atoi (const char *str);`	converte a cadeia de caracteres apontada por str numa quantidade inteira de tipo ***int***

Figura 1.79 - Funções para conversão de cadeias de caracteres em quantidades numéricas.

Para além dos tipos *div_t* e *ldiv_t*, a biblioteca ***stdlib*** define ainda o tipo *size_t* que é o tipo do resultado do operador **sizeof**. Contém também dois identificadores que podem ser usados pela função **exit** para indicar o estado de execução de um programa. A constante EXIT_FAILURE, que vale 1, pode ser usada para indicar a finalização do programa sem sucesso. A constante EXIT_SUCCESS, que vale 0, pode ser usada para indicar a finalização do programa com sucesso.

Esta biblioteca contém ainda as funções para gestão de memória, que permitem adjudicar e libertar memória dinâmica, que serão apresentadas mais tarde.

1.10 Leituras recomendadas

- 3º, 4º, 5º e 6º capítulos do livro "C A Software Approach", 3ª edição, de Peter A. Darnell e Philip E. Margolis, da editora Springer-Verlag, 1996.

Capítulo 2

COMPLEMENTOS SOBRE C

Sumário

Este capítulo é dedicado aos aspectos mais avançados da linguagem C. Começamos por apresentar o conceito de função generalizada, que é o único tipo de subprograma existente e como através dela se implementam funções e procedimentos. Explicamos a passagem de parâmetros às funções e introduzimos os primeiros conceitos sobre ponteiros.

De seguida, apresentamos o modo como se definem os tipos de dados estruturados, nomeadamente, os agregados unidimensionais, bidimensionais e tridimensionais, as cadeias de caracteres e a biblioteca ***string*** que providencia funções para a sua manipulação e os registos, que na linguagem C se designam por estruturas. Apresentamos também como se definem tipos de dados enumerados.

Devido à interacção entre os agregados e os ponteiros, o que se costuma designar por dualidade ponteiro agregado, e à necessidade de implementar programas mais eficientes recorrendo a este novo tipo de dados, vamos expondo as suas características mais avançadas à medida que explicamos os tipos de dados estruturados. Apresentamos, designadamente, os seus operadores específicos, a aritmética de ponteiros, a dualidade ponteiro agregado, para agregados unidimensionais e multidimensionais e a construção de estruturas de dados versáteis envolvendo agregados e ponteiros.

Apresentamos também as classes de armazenamento das variáveis e redefinimos os conceitos de objectos locais e globais, bem como dos níveis de visibilidade dos objectos aplicados ao facto da linguagem C permitir a construção de aplicações distribuídas por vários ficheiros fonte.

2.1 Funções

No estabelecimento de soluções de problemas complexos é essencial controlar o grau de complexidade da descrição. A descrição deve ser organizada de uma forma hierarquizada, também designada de decomposição do topo para a base (*Top-Down Decomposition*), de modo a que corresponda, a cada nível de abstracção considerado, uma decomposição num número limitado de operações, com recurso a um número também limitado de variáveis.

Só assim é possível:

- Minimizar a interacção entre as diferentes operações, impondo regras estritas na formulação das dependências de informação.
- Desenvolver metodologias que, de uma forma simples, conduzam à demonstração da correcção dos algoritmos estabelecidos.
- Conceber um desenho da solução que:
    - Promova a compactação através da definição de operações reutilizáveis em diferentes contextos.
    - Enquadre a mudança ao longo do tempo, possibilitando a alteração de especificações com um mínimo de esforço.

A transcrição da solução numa linguagem de programação deve, depois, procurar reflectir a hierarquia de decomposição que foi explicitada na descrição. Isto consegue-se fazendo um uso intensivo dos mecanismos de **encapsulamento de informação**, presentes na linguagem utilizada, para implementar cada operação como uma secção autónoma de código, ou seja, com um **subprograma** e promover, assim, uma visibilidade controlada dos detalhes de implementação.

Desta forma, é possível:

- Separar a especificação da operação, ou seja, a descrição da sua funcionalidade interna e do mecanismo de comunicação com o exterior, constituída pelas variáveis de entrada e de saída, da sua implementação.
- Introduzir características de robustez e de desenho para o teste no código produzido.
- Validar de uma maneira controlada e integrar de um modo progressivo os diferentes componentes da aplicação
- Planear com eficácia a distribuição de tarefas pelos diversos membros da equipa de trabalho.

Na linguagem Pascal existem dois tipos de subprogramas. O **procedimento**, que se apresenta na Figura 2.1, é um mecanismo de encapsulamento de informação que permite a construção de operações mais complexas a partir de uma combinação de operações mais simples. Após a sua definição, a nova operação é identificada por um nome e por uma lista opcional de parâmetros de comunicação. Os parâmetros de entrada, que representam valores necessários à realização da operação, e os parâmetros de saída, que representam valores produzidos pela realização da operação.

Figura 2.1 - Esquema de um procedimento.

A **função**, que se apresenta na Figura 2.2, é um mecanismo de encapsulamento de informação que permite a descrição de um processo de cálculo complexo a partir de uma combinação de operações mais simples. Após a sua definição, a nova operação é identificada por um nome e por uma lista opcional de parâmetros de comunicação. Os parâmetros de entrada, que representam valores necessários à realização da operação e pela indicação do tipo do resultado de saída, ou seja, do valor calculado e devolvido pela função.

Figura 2.2 - Esquema de uma função.

Na linguagem C só temos um tipo de subprograma que é a **função generalizada**, que se apresenta na Figura 2.3. A função generalizada é um mecanismo de encapsulamento de informação que combina as características apresentadas pelo procedimento e pela função. Semanticamente, pode ser encarada como um procedimento que devolve o estado de realização da operação. Pode ser visto como uma caixa preta que recebe informação à entrada e que produz informação à saída, sendo que os detalhes associados com a implementação da operação são invisíveis externamente e não originam qualquer interacção com o exterior. A informação de entrada, quando é necessária à realização da operação, é fornecida à função generalizada através de um conjunto de **parâmetros de entrada**. A informação de saída, quando é produzida pela realização da operação, é fornecida através de um conjunto de **parâmetros de saída** e/ou pela produção de um **resultado de saída**.

Figura 2.3 - Esquema de uma função generalizada da linguagem C.

Após a sua definição, a nova operação é identificada por um nome, por uma lista opcional de parâmetros de comunicação, ou seja, os parâmetros de entrada, os parâmetros de saída, e pela indicação do tipo do valor devolvido.

A Figura 2.4 apresenta a definição da função de conversão de distâncias de milhas para quilómetros no Pascal e na linguagem C.

```
function CONVERTE_DISTANCIA (ML: real): real;          (* no Pascal *)
const MIL_QUI = 1.609;
begin
   CONVERTE_DISTANCIA := MIL_QUI * ML
end;
```

```
#define MIL_QUI 1.609                              /* na linguagem C */
...
double CONVERTE_DISTANCIA (double ML)        /* definição da função */
{
   return ML * MIL_QUI;
}
```

Figura 2.4 - Exemplo da função da conversão de distâncias de milhas para quilómetros.

Uma função pode aparecer num programa, de três formas diferentes:

- A **definição** é a declaração que define o que a função faz, bem como o número e tipo de parâmetros e o tipo de resultado de saída. Normalmente é colocada depois da função **main**, ou num ficheiro de implementação.
- A **invocação** invoca a execução da função.
- A **alusão**, ou protótipo é a declaração que define a interface da função, ou seja, o tipo, o nome e a lista de parâmetros. Normalmente é colocada no início do ficheiro fonte logo após as directivas de *include*, ou num ficheiro de interface, que tem a extensão **.h**.

2.1.1 Definição de uma função

Tal como já foi mostrado para a função **main** na Figura 1.3, qualquer função na linguagem C, supõe a especificação do seu **cabeçalho** e do seu **corpo**. A Figura 2.5 apresenta detalhadamente os vários componentes da função CONVERTE_DISTANCIA.

No cabeçalho, indica-se o **tipo do valor devolvido**, o **nome da função**, e entre parênteses curvos, a **lista de parâmetros** de comunicação. Se a função não tiver lista de parâmetros é utilizada a declaração de **void**. Se a função não tiver resultado de saída é utilizada a declaração de tipo **void**, como sendo o tipo do valor devolvido.

O corpo da função é delimitado pelos separadores **{** e **}**. Para devolver o valor calculado pela função, utiliza-se a palavra reservada **return** seguido da expressão a devolver. O valor de retorno pode ser de qualquer tipo, excepto um agregado ou uma função. É possível devolver uma estrutura ou união, mas não é recomendado porque é ineficiente. Uma função pode conter qualquer número de instruções **return**. Se não houver instrução de **return**, a execução da função termina quando atingido o separador **}**. Nesse caso, o valor devolvido é indefinido. O valor devolvido na instrução de **return** deve ser compatível em termos de atribuição com o tipo de saída da função. Para determinar se o valor devolvido é aceitável, o compilador usa as mesmas regras de compatibilidade da instrução de atribuição.

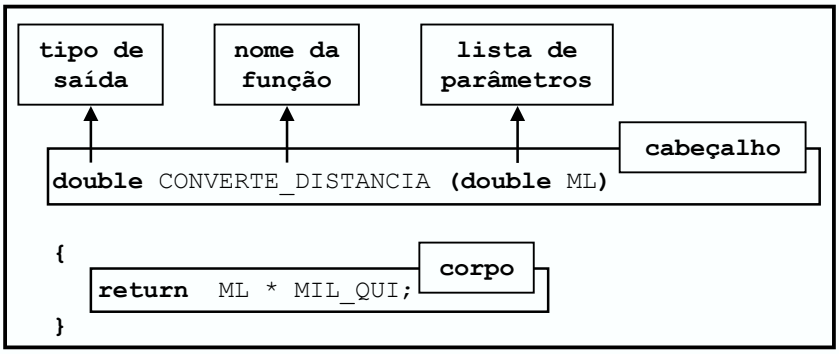

Figura 2.5 - A função CONVERTE_DISTANCIA.

Normalmente os parâmetros de uma função são parâmetros de entrada e como tal são passados por valor. Ou seja, a função não altera o valor da variável passada à função. A passagem por valor consiste em referir apenas a variável, ou a expressão, na invocação da função.

Mas, por vezes é necessário usar parâmetros de entrada-saída numa função. Ou seja, o valor da variável necessita de ser alterada durante a execução da função. Nesses casos estamos perante a passagem de parâmetros por referência, também designada por endereço. Nesta situação é passado o endereço, ou seja, a localização da variável na memória. Isso é feito usando um ponteiro para a variável na invocação da função.

2.1.2 Introdução aos ponteiros

Uma característica saliente da linguagem C é permitir referenciar de um modo muito simples a localização de variáveis em memória e, a partir dessa localização, aceder ao seu conteúdo. Para isso existem os dois operadores unários apresentados na Figura 2.6 que são aplicados directamente a uma variável.

localização em memória da variável ::= **&** identificador de variável
referência indirecta à variável ::= ***** identificador de variável de tipo ponteiro

Figura 2.6 - Definição formal dos operadores para manipulação de variáveis através de ponteiros.

O **operador endereço**, cujo símbolo é o **&**, dá a localização em memória da variável. À localização de uma variável chama-se habitualmente **ponteiro para a variável**, já que o valor que lhe está associado aponta para, ou referencia, a região de memória onde essa variável está armazenada. Mais especificamente, o valor que está associado a um ponteiro representa o endereço do primeiro *byte* da região de armazenamento da variável. Assim, um ponteiro, embora assuma o valor de um endereço, não é propriamente um endereço. O que está subjacente ao conceito não é a localização de um *byte*, que define a divisão física da memória, mas de uma variável, ou seja, da divisão lógica da memória. Portanto, não faz sentido falar-se em ponteiros em abstracto, porque um ponteiro está sempre associado a um tipo de dados bem definido. Como nos processadores actuais, o barramento de endereços tem uma dimensão de 32 *bits*, o tamanho do formato do tipo ponteiro é igual a 4 *bytes*. Por isso, tem-se que **sizeof (**ponteiro para um tipo de dados genérico**)** = 4.

O **operador apontado por**, cujo símbolo é o *****, dá acesso à variável, cuja localização em memória está armazenada numa variável de tipo ponteiro, ou seja, é uma referência indirecta à variável. Este operador tem outro significado quando é usado na declaração de

uma variável. Nessa situação indica que a variável é de tipo ponteiro. E quando é utilizado na lista de parâmetros, precedendo um parâmetro, indica que o parâmetro é um parâmetro de entrada-saída, ou seja, que é uma passagem por referência.

A Figura 2.7 apresenta um exemplo da manipulação de uma variável através de um ponteiro. A primeira linha de código declara uma variável A de tipo ***int*** e uma variável PA de tipo ponteiro para ***int***. A atribuição PA = &A, coloca na variável PA o endereço da variável A, daí que, a variável PA fica a apontar para a variável A. A atribuição *PA = 23 coloca o valor 23 na variável A, através do ponteiro PA.

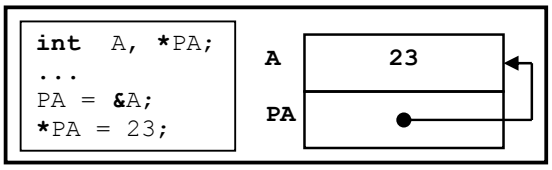

Figura 2.7 - Exemplo de manipulação de uma variável através de um ponteiro.

2.1.3 Invocação de uma função

Aquando da invocação de uma função, o controlo do programa passa para a função. A invocação de uma função é uma expressão e como tal pode aparecer em qualquer sítio onde possa aparecer uma expressão. A função devolve sempre um valor, a não ser quando o valor devolvido é **void**. No entanto, é possível invocar uma função sem utilizar o valor devolvido.

A Figura 2.8 apresenta a invocação da função CONVERTE_DISTANCIA. Podemos invocar a função numa expressão de atribuição do seu valor à variável QUILOMETROS, ou em alternativa podemos invocá-la directamente na instrução de saída de dados **printf**. A definição da função deve ser colocada após a função **main**.

```
#include <stdio.h> /* interface com a biblioteca de entrada/saída */
#define MIL_QUI 1.609          /* definição do factor de conversão */
double CONVERTE_DISTANCIA (double);             /* alusão à função */
int main ( void )
{
  double MILHAS,                    /* distância expressa em milhas */
         QUILOMETROS;          /* distância expressa em quilómetros */
  do     /* leitura com validação da distância expressa em milhas */
  {
     printf ("Distância em milhas? ");
     scanf ("%lf", &MILHAS);
  } while (MILHAS < 0.0);
                                                /* invocação da função */
  QUILOMETROS = CONVERTE_DISTANCIA (MILHAS);
                /* impressão da distância expressa em quilómetros */
  printf ("Distância em quilómetros é %8.3f\n", QUILOMETROS);
  return 0;
}
...
```

Figura 2.8 - Programa da conversão de distâncias utilizando a função CONVERTE_DISTANCIA.

2.1.4 Alusão de uma função

A alusão de uma função, permite ao compilador assegurar que o número correcto de parâmetros são passados na invocação. Proíbe a passagem de parâmetros que não possam ser convertidos para o tipo correcto. Por outro lado, converte pacificamente os parâmetros quando isso é possível. Permite também verificar se a atribuição do resultado da função está de acordo com o tipo do valor devolvido pela função. O código apresentado na Figura 2.8 faz a alusão à função CONVERTE_DISTANCIA na terceira linha. Na lista de parâmetros indica-se apenas o tipo dos parâmetros e omite-se os seus nomes, uma vez que o compilador necessita apenas de saber o tipo dos parâmetros, para verificar se a passagem de parâmetros está correcta quando se invoca a função.

2.1.5 Procedimentos na linguagem C

Na linguagem C um **procedimento** é implementado por uma função que não devolve qualquer valor, ou seja, cujo tipo do resultado de saída é de tipo **void**. No corpo de uma função que devolve o tipo **void**, não é necessário utilizar a instrução **return**. A não ser que, por razões de algoritmo haja a necessidade de abandonar a função em sítios diferentes. Nesse caso usa-se a instrução **return** sem expressão.

A Figura 2.9 apresenta o procedimento TROCA em Pascal e a sua implementação com uma função na linguagem C.

```
procedure TROCA (var X, Y: integer);                    (* no Pascal *)
var TEMP : integer;
begin
  TEMP := X; X := Y; Y := TEMP
end;
```

```
void TROCA (int *X, int *Y)                          /* na linguagem C */
{
  int TEMP;
  TEMP = *X; *X = *Y; *Y = TEMP;
}
```

Figura 2.9 - Procedimento na linguagem C.

Na definição da função os dois parâmetros X e Y são parâmetros de entrada-saída, pelo que, são passados por referência através de ponteiros para as variáveis. Para indicar que o parâmetro é um parâmetro de entrada-saída, coloca-se o **operador apontado por** atrás do parâmetro. Na invocação da função é necessário passar a localização em memória das variáveis actuais, através de ponteiros para as variáveis. Para passar a localização da variável, coloca-se o **operador endereço** atrás da variável, tal como se mostra na Figura 2.10.

```
void TROCA (int *X, int *Y);                        /* alusão à função */
...
int A, B;                                      /* definição das variáveis */
...
TROCA (&A, &B);                                   /* invocação da função */
```

Figura 2.10 - Exemplo da invocação da função TROCA.

2.2 Agregados (*arrays*)

O construtor agregado (***array***) é um mecanismo usado por muitas linguagens de programação para criar **tipos de dados estruturados**, ou seja, tipos de dados mais complexos a partir de tipos de dados mais simples, designados de **tipo base**. O tipo agregado é concebido como um conjunto ordenado de objectos do tipo base. Cada objecto do tipo base, designado genericamente por elemento, é referenciado pela posição que ocupa no agregado. Estamos perante um **acesso por indexação**.

A Figura 2.11 apresenta graficamente um agregado unidimensional com 11 elementos e um agregado bidimensional com 15 elementos.

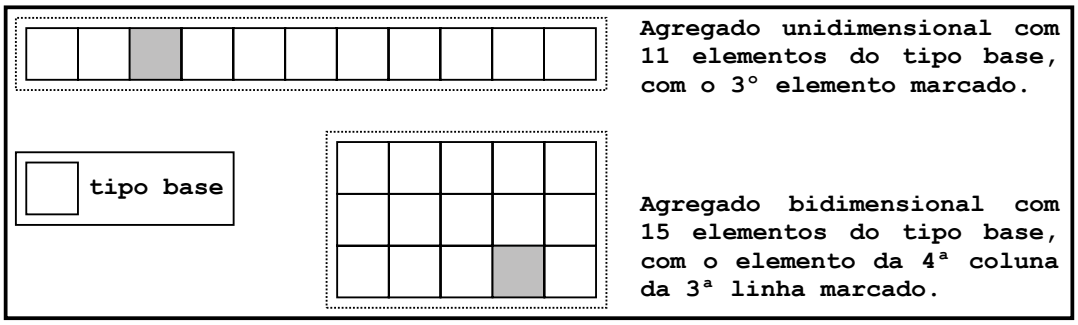

Figura 2.11 - Visualização gráfica de agregados unidimensionais e bidimensionais.

A declaração de variáveis de tipo agregado segue a regra geral de declaração de variáveis da linguagem C e a sua definição formal é apresentada na Figura 2.12.

tipo de dados lista de variáveis de tipo agregado **;**

tipo de dados ::= qualquer tipo de dados válido na linguagem C

lista de variáveis de tipo agregado ::= *identificador de variável de tipo agregado* |
lista de variáveis de tipo agregado **,** *identificador de variável de tipo agregado*

identificador de variável de tipo agregado ::= *identificador de variável descritor de dimensão* |
identificador de variável descritor de dimensão **=** *expressão de inicialização*

identificador de variável ::= qualquer identificador válido na linguagem C

descritor de dimensão ::= **[ ]** | **[** constante numérica inteira positiva **]** |
descritor de dimensão **[ ]** |
descritor de dimensão **[** constante numérica inteira positiva **]**

expressão de inicialização ::= **{** *listas de constantes do tipo base* **}**

Figura 2.12 - Definição formal da declaração de um tipo agregado.

A Figura 2.13, a Figura 2.41 e a Figura 2.42 apresentam respectivamente exemplos da declaração de agregados unidimensionais, bidimensionais e tridimensionais, bem como da sua colocação na memória. Uma variável de tipo agregado distingue-se de uma variável simples, devido aos descritores de dimensão, ou seja, aos parênteses rectos a seguir ao nome da variável. A constante numérica inteira positiva, incluída no descritor de dimensão, indica o tamanho da dimensão. Se se tratar de um agregado unidimensional, isto é equivalente a indicar o número de elementos do agregado. Para um agregado multidimensional, o número de elementos do agregado é calculado pelo produto dos tamanhos das diferentes dimensões.

Quando o descritor de dimensão não inclui a constante numérica positiva, diz-se que se está perante uma **definição incompleta**, já que a reserva de espaço de armazenamento não é feita. O seu uso é, por isso, bastante restrito e é geralmente usado quando se faz a inicialização no momento da declaração. A inicialização é feita através de listas de constantes do tipo base, separadas pela vírgula e inseridas entre chavetas. O número de listas de inicialização e de níveis de chavetas depende do número de dimensões do agregado.

Um elemento particular do agregado é referenciado através da indicação do nome da variável, seguido do valor de um ou mais índices, tantos quantas as dimensões do agregado, colocados individualmente entre parênteses rectos. Na linguagem C, os índices são variáveis numéricas inteiras positivas e o índice do elemento localizado mais à esquerda em cada dimensão é o índice zero.

2.2.1 Agregados unidimensionais

A Figura 2.13 apresenta exemplos da declaração e inicialização de agregados unidimensionais, bem como, da sua colocação na memória.

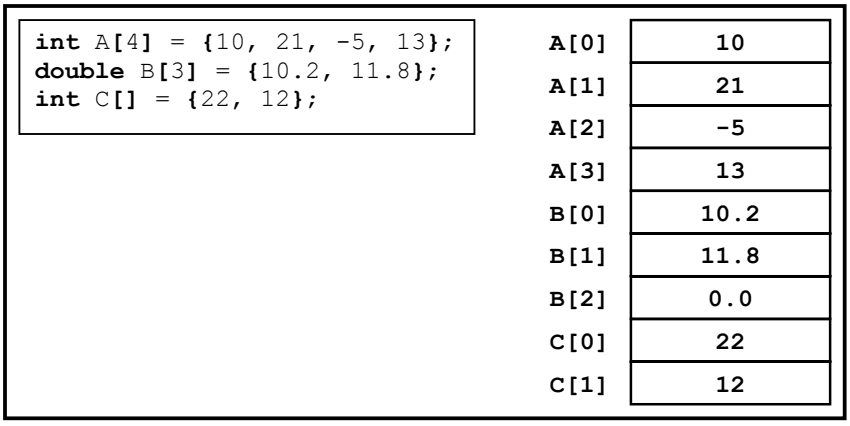

Figura 2.13 - Declaração e inicialização de agregados unidimensionais.

A expressão de inicialização para variáveis de tipo agregado unidimensional de um tipo base consiste numa lista de constantes do tipo base, separadas pela vírgula e colocadas entre chavetas. A atribuição das constantes aos elementos do agregado processa-se da esquerda para a direita e é realizada até ao esgotamento da lista.

Assim, a constante numérica positiva, incluída no descritor de dimensão, terá que ser maior, ou quando muito igual, ao número de constantes da lista. Se for igual, como é o caso do agregado A, todos os elementos do agregado serão inicializados com os valores indicados. Se for maior, como é o caso do agregado B, só os K primeiros elementos, em que K é igual ao número de constantes da lista, serão inicializados com os valores indicados, sendo que os restantes elementos são inicializados a zero. Alternativamente, como é o caso do agregado C, o descritor de dimensão pode ser omitido e então será reservado espaço, e serão inicializados, tantos elementos quanto o número de constantes da lista de inicialização.

2.2.2 Dualidade ponteiro agregado

Em geral, quando numa instrução se referencia o nome de uma variável, pretende-se aludir como um todo à região de memória que está reservada para armazenamento dos seus valores. Tal não se passa, contudo, quando se referencia o nome de uma variável de tipo agregado unidimensional. O significado atribuído é, neste caso, o de um **ponteiro para o primeiro elemento do agregado**.

$$\text{TIPO\_BASE A[N]; } \Rightarrow \text{ A} \equiv \&\text{A[0]}$$

Na linguagem C existe aquilo que se costuma designar por **dualidade ponteiro agregado**, já que, quer uma variável de tipo agregado unidimensional de um dado tipo base, quer uma variável de tipo ponteiro para o mesmo tipo base, são formas equivalentes de referenciar uma região de memória formada por um conjunto contíguo de variáveis do tipo base. Consequentemente, é possível aceder a um elemento do agregado unidimensional, quer da forma usual em Pascal, quer através do operador apontado por.

$$\text{A+i} \equiv \&\text{A[i]} \Rightarrow *(\text{A+i}) \equiv *\&\text{A[i]} \equiv \text{A[i]}, \text{ com } 0 \leq i < N$$

Ou seja, a localização do elemento de índice **i** do agregado **A** pode ser alternativamente expressa por **&A[i]** ou por **A+i**, e o valor do mesmo elemento por **A[i]** ou por ***(A+i)**.

Por conseguinte, uma variável de tipo ponteiro para um tipo base pode ser encarada como uma variável de tipo agregado unidimensional do mesmo tipo base na referência a uma região de memória formada por um conjunto contíguo de variáveis do tipo base. Pelo que, a seguinte declaração do agregado A e do ponteiro para o agregado PA inicializado com o endereço inicial do agregado, resulta no mapa de reserva de espaço em memória que se apresenta na Figura 2.14 e permite a utilização do ponteiro PA para aceder aos elementos do agregado.

$$\text{TIPO\_BASE A[N], *PA = A;}$$

$$\textbf{PA+i} \equiv \&\text{PA[i]} = \textbf{\&A[i]} \Rightarrow \textbf{*(PA+i)} \equiv \text{PA[i]} = \textbf{A[i]}, \text{ com } 0 \leq i < N$$

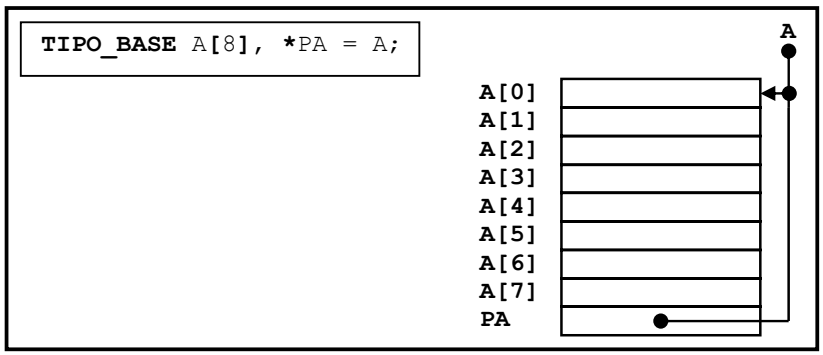

Figura 2.14 - Declaração de um agregado unidimensional e de um ponteiro para o agregado.

Existe, porém, uma diferença subtil entre as variáveis de tipo agregado unidimensional de um tipo base e as variáveis de tipo ponteiro para o mesmo tipo base. Como se verifica do mapa de reserva de espaço em memória apresentado na Figura 2.14, não existe espaço directamente associado com a variável A e, portanto, A é um ponteiro constante, cujo valor não pode ser modificado. Assim, instruções do tipo A = expressão**;** são ilegais.

2.2.3 Agregados como parâmetros de funções

Na linguagem C, dentro de uma função não é possível saber com quantos elementos foi declarado um agregado que foi passado como argumento para essa função. Pelo que, para além do agregado deve ser passado o número de elementos a processar. A Figura 2.15 apresenta o código da definição de uma função que inicializa a zero todos os elementos de um agregado. A função tem um parâmetro de entrada-saída que é o agregado a inicializar e um parâmetro de entrada que é a dimensão do agregado. Para indicar que o parâmetro de entrada-saída é um agregado, usa-se o nome do parâmetro formal seguido dos parênteses rectos. Quando da invocação da função, o endereço inicial do agregado é passado à função usando o nome do agregado.

```
void INICIALIZAR (int A[], int N)          /* definição da função */
{
  int I;
  for (I = 0; I < N; I++)  A[I] = 0;
}
-------------------------------------------------------------------
int X[10];
...
INICIALIZAR (X, 10);                       /* invocação da função */
```

Figura 2.15 - Passagem de um agregado a uma função.

É possível obter o número de elementos de um agregado dividindo o tamanho em *bytes* do agregado pelo tamanho em *bytes* de um dos seus elementos, ou seja, pelo tamanho do tipo base, utilizando para tal a seguinte expressão envolvendo o operador **sizeof**.

sizeof (*agregado* **)** / **sizeof (** *elemento do agregado* **)**

A expressão funciona independentemente do tipo de elementos do agregado. A Figura 2.16 apresenta uma aplicação desta expressão. O agregado C é declarado sem descritor de dimensão, sendo no entanto inicializado. Na altura da invocação da função FUNC usamos a expressão para passar o número de elementos do agregado ao parâmetro de entrada N.

```
void FUNC (int A[], int N);                    /* alusão da função */
...
int C[] = {20, 21, 22, 23};
...                  /* invocação da função FUNC para o agregado C */
FUNC (C, sizeof (C) / sizeof (C[0]));
```

Figura 2.16 - Utilização do operador **sizeof** para calcular o número de elementos de um agregado.

Um dos erros mais frequentemente cometido, por programadores que se estão a iniciar na utilização da linguagem C, é ultrapassar a dimensão de um agregado, nomeadamente em ciclos repetitivos **for**. No exemplo apresentado na Figura 2.17, o **for** vai processar o agregado desde o elemento de índice 0 até ao elemento de índice 10, que não existe no agregado. Acontece que é nesta posição de memória que está armazenada a variável I que controla o **for**, pelo que, o seu valor vai ser reinicializado a zero e portanto o **for** é de novo repetido. Estamos perante um ciclo repetitivo infinito criado por engano.

```
int I, AR[10];
...
for (I = 0; I <= 10; I++) AR[I] = 0;          /* situação de erro */
```

Figura 2.17 - Situação de erro na utilização do ciclo **for** para processar um agregado.

2.3 Cadeias de caracteres (*strings*)

Na linguagem C, uma cadeia de caracteres (***string***) é entendida como um agregado unidimensional de caracteres terminado obrigatoriamente pelo carácter nulo **'\0'**, também reconhecido pelo identificador NUL. A expressão de inicialização para variáveis de tipo cadeia de caracteres, pode por isso ser estabelecida alternativamente segundo as regras anteriores de inicialização de agregados unidimensionais, tendo o cuidado de garantir que a lista de constantes inclui em último lugar o carácter nulo, ou através de uma constante de tipo cadeia de caracteres, ou seja, uma sequência de caracteres delimitada por aspas duplas, onde o compilador coloca automaticamente no fim o carácter nulo. Quando se declara e inicializa uma cadeia de caracteres e o descritor de dimensão é maior do que o número de caracteres de inicialização, então todos os restantes caracteres da cadeia são inicializados com o carácter nulo.

A Figura 2.18 apresenta alguns exemplos de declaração e inicialização de cadeias de caracteres. A variável VOGAIS não é uma cadeia de caracteres porque não foi terminada com o carácter nulo e foi declarada para armazenar apenas as cinco vogais. Pelo que, é apenas um agregado de caracteres. A variável CIDADE também está declarada como um agregado de caracteres, mas, como foi declarado com o tamanho de 10 caracteres, então todos os caracteres não inicializados, que são quatro, são colocados a '\0'. Pelo que, é de facto uma cadeia de caracteres. A variável OLA é uma cadeia de caracteres e está explicitamente declarada com o carácter nulo. A variável CARTA não é uma cadeia de caracteres, porque apesar de ser inicializada com uma constante de tipo cadeia de caracteres, foi declarada com um tamanho insuficiente para armazenar o carácter nulo. A variável TEXTO está declarada sem especificação de tamanho e portanto, é criado com a dimensão exacta para conter a cadeia de caracteres de inicialização mais o carácter nulo, ou seja, com a dimensão de 16 caracteres.

```
char VOGAIS[5] = {'a', 'e', 'i', 'o', 'u'};
                          /* VOGAIS não é uma cadeia de caracteres */
char CIDADE[10] = {'A', 'v', 'e', 'i', 'r', 'o'};
                               /* CIDADE é uma cadeia de caracteres */
char OLA[4] = {'o', 'l', 'a', '\0'};
                                 /* OLA é uma cadeia de caracteres */
char CARTA[4] = "tres";
                           /* CARTA não é uma cadeia de caracteres */
char TEXTO[] = "era uma vez ...";
                                /* TEXTO é uma cadeia de caracteres */
```

Figura 2.18 - Exemplos de declaração e inicialização de cadeias de caracteres.

A Figura 2.19 apresenta o mapa de reserva de espaço em memória da declaração de duas cadeias de caracteres. A variável OLA foi declarada e inicializada com quatro caracteres. A variável TB é inicializada com uma constante de 8 caracteres e o carácter nulo é automaticamente colocado no fim.

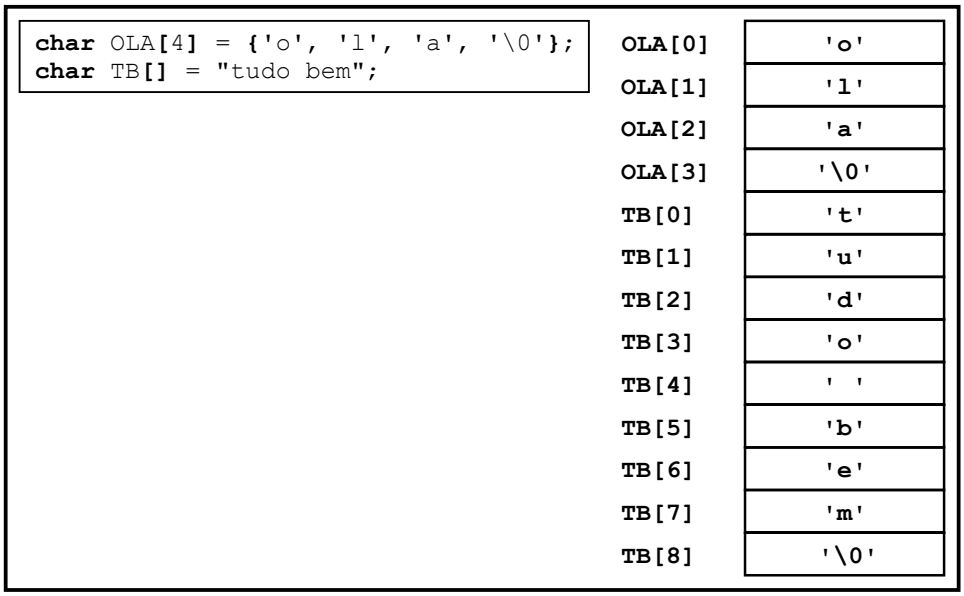

Figura 2.19 - Mapa de reserva de espaço em memória de duas cadeias de caracteres.

É possível criar um ponteiro para uma cadeia de caracteres e simultaneamente iniciá-lo com uma cadeia de caracteres constante, usando para esse efeito a seguinte instrução.

char ****ponteiro* **= "*constante cadeia de caracteres***"**

A Figura 2.20 apresenta o mapa de reserva de espaço em memória de um exemplo deste tipo de declaração. O ponteiro PTS aponta para a cadeia de caracteres **"**Aveiro**"**, que é uma cadeia de caracteres constante, pelo que, não pode ser alterada. O valor de PTS pode ser alterado, por exemplo, atribuindo-lhe outra constante, ou uma variável, de tipo cadeia de caracteres, e caso o seja, perde-se o acesso à cadeia de caracteres **"**Aveiro**"**.

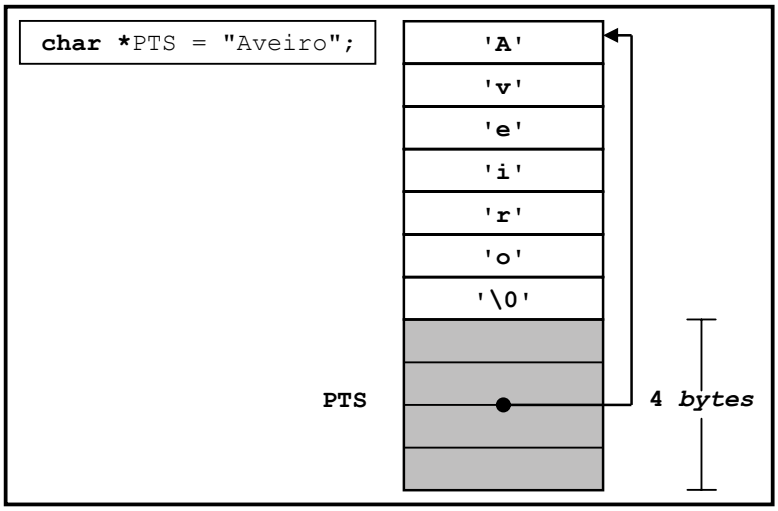

Figura 2.20 - Mapa de reserva de espaço em memória de um ponteiro para uma cadeia de caracteres inicializado com uma cadeia de caracteres constante.

2.3.1 Atribuição de cadeias de caracteres

Uma cadeia de caracteres é um agregado de caracteres, portanto, uma constante de tipo cadeia de caracteres é também um ponteiro para o seu primeiro carácter. Pelo que, quando uma variável de tipo cadeia de caracteres é declarada com o operador **[ ]**, não é possível atribuir-lhe directamente uma constante de tipo cadeia de caracteres, uma vez que o nome da variável cadeia de caracteres é um ponteiro constante, logo não pode ser alterado. Mas, já é possível, atribuir uma constante de tipo carácter a um elemento da variável, indicando para esse efeito o índice do elemento entre parênteses rectos. Para atribuir a uma variável de tipo cadeia de caracteres, uma constante de tipo cadeia de caracteres ou outra variável de tipo cadeia de caracteres, existe a função **strcpy** da biblioteca de execução ANSI ***string***.

Mas já é possível atribuir a uma variável de tipo ponteiro para ***char*** uma constante de tipo cadeia de caracteres ou outra variável de tipo cadeia de caracteres. A Figura 2.21 apresenta um exemplo que mostra esta diferença.

```
char CIDADE[10] = "aveiro";           /* CIDADE é igual a "aveiro" */
char *PTS = "Aveiro";                  /* PTS aponta para "Aveiro" */
...
CIDADE = "Lisboa";       /* ERRO não se pode fazer esta atribuição */
CIDADE[0] = 'A';               /* agora CIDADE é igual a "Aveiro" */
strcpy (CIDADE, "Lisboa");     /* agora CIDADE é igual a "Lisboa" */
PTS = "Lisboa";                 /* agora PTS aponta para "Lisboa" */
```

Figura 2.21 - Exemplo da atribuição de cadeias de caracteres.

2.3.2 Cadeias de caracteres versus caracteres

Uma constante de tipo carácter é colocado entre pelicas, por exemplo **'a'**, enquanto que uma constante de tipo cadeia de caracteres é colocada entre aspas, por exemplo **"a"**. Existe uma diferença importante entre um carácter e uma cadeia de caracteres, mesmo quando esta é apenas constituída por um único carácter. Enquanto que um carácter ocupa apenas um *byte*, uma cadeia de caracteres ocupa tantos *bytes* quantos os caracteres, mais um *byte* para armazenar o carácter nulo **'\0'**. A Figura 2.22 apresenta alguns exemplos da utilização de um ponteiro para carácter. Pode-se atribuir um carácter a um dos elementos de uma cadeia de caracteres através de um ponteiro para carácter, desde que a cadeia de caracteres não seja constante, como é o caso do exemplo apresentado. Portanto, a instrução de atribuição *PTS = 'A' dá erro de execução, devido a violação de memória protegida. A instrução de atribuição *PTS = "A" está errada, uma vez que, não se pode atribuir um ponteiro para carácter a um carácter. É preciso ter em atenção que *PTS é do tipo ***char***, enquanto que, "A" é do tipo ***char****. Como já foi referido, é possível atribuir uma constante de tipo cadeia de caracteres a um ponteiro para carácter, mas não é possível atribuir-lhe um carácter, pelo que, a instrução de atribuição PTS = 'A' está errada.

```
char *PTS = "aveiro";                  /* PTS aponta para "aveiro" */
...
*PTS = 'A'; /* ERRO "Aveiro" é uma cadeia de caracteres constante */
*PTS = "A";              /* ERRO não se pode fazer esta atribuição */

PTS = 'A';               /* ERRO não se pode fazer esta atribuição */
PTS = "A";                             /* agora PTS aponta para "A" */
```

Figura 2.22 - Exemplo da manipulação de um ponteiro para carácter.

2.3.3 Cadeias de caracteres como parâmetros de funções

A passagem de uma cadeia de caracteres a uma função é semelhante à passagem de um agregado. Existem duas maneiras para indicar que um parâmetro de uma função é uma cadeia de caracteres. A notação recomendada pela norma ANSI utiliza o operador **[ ]**, declarando que o parâmetro é um agregado de caracteres. Em alternativa pode-se utilizar o operador *****, declarando que o parâmetro é um ponteiro para carácter. Quando se passa uma cadeia de caracteres a uma função, não é necessário indicar o comprimento da cadeia de caracteres, uma vez que este valor pode ser calculado recorrendo à função **strlen** da biblioteca de execução ANSI ***string***. A Figura 2.23 apresenta um exemplo da definição e invocação de uma função que processa uma cadeia de caracteres.

```
int FUNC (char PST[])                    /* definição da função FUNC */
{                          /* ou em alternativa int FUNC (char *PST) */
  ...
  return ...;
}
-------------------------------------------------------------------------
char ST[] = "bom dia";
...    /* invocação da função FUNC para a cadeia de caracteres ST */
FUNC (ST);
```

Figura 2.23 - Função com um parâmetro de tipo cadeia de caracteres.

2.3.4 Codificação circular de caracteres

Na linguagem C, o tipo ***char*** permite armazenar quer um carácter quer um valor inteiro. Devido a esta polivalência é possível misturar numa mesma expressão caracteres e valores numéricos inteiros, sem ter a necessidade de recorrer a funções de conversão de tipos. Pelo que, a expressão para implementar a codificação circular de um carácter alfabético é simplificada, quando comparada com o Pascal. A Figura 2.24 apresenta o deslocamento circular do carácter 'z' três posições para a frente, sendo transformado no carácter 'c'. Para obtermos o efeito de deslocamento circular num conjunto linear, temos que utilizar o operador resto da divisão, que na linguagem C é o símbolo **%**.

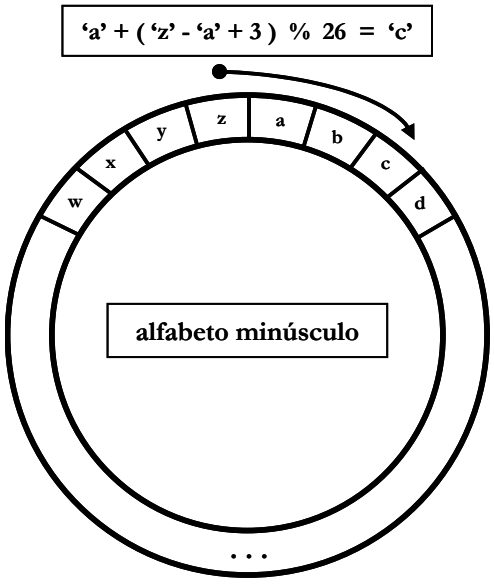

Figura 2.24 - Deslocamento circular de um carácter três posições para a frente.

A Figura 2.25 apresenta a função que codifica um carácter minúsculo. A função não testa se o carácter é ou não minúsculo, pelo que, só deve ser invocada para caracteres minúsculos. Se o valor do deslocamento circular, que é representado pelo parâmetro de entrada K, for positivo então o carácter é codificado K posições para a frente, se for negativo então o carácter é codificado K posições para a trás, e se K for zero o carácter sai inalterado. Apresentam-se as soluções em Pascal e na linguagem C.

```
                                                        (* no Pascal *)
function CODIFICAR_MINUSCULO (CAR: char; K: integer): char;
var CAR_S : char;
begin
  CAR_S := chr(ord('a') + (ord(CAR) - ord('a') + 26 + K) mod 26);
  CODIFICAR_MINUSCULO := CAR_S
end;
```

```
char CODIFICAR_MINUSCULO (char CAR, int K)       /* na linguagem C */
{
  return  'a' + (CAR - 'a' + 26 + K ) % 26;
}
```

Figura 2.25 - Função que faz a codificação circular de um carácter minúsculo.

2.3.5 Biblioteca string

A biblioteca ***string*** contém funções de manipulação de cadeias de caracteres. Para utilizar as suas funções é preciso fazer a inclusão do ficheiro de interface *string.h* com a seguinte directiva do pré-processador.

#**include** <string.h>

As funções dividem-se funcionalmente em cinco grupos: **funções de cópia**; **funções de concatenação**; **funções de comparação**; **funções de busca**; e **funções diversas**.

A Figura 2.26 apresenta as funções de cópia.

```
void *memcpy (void *zd, void *zp, size_t n);
void *memmove (void *zd, void *zp, size_t n);
char *strcpy (char *zd, char *zp);
char *strncpy (char *zd, char *zp, size_t n);
```

Figura 2.26 - Funções de cópia.

Todas as funções copiam *byte* a *byte* o conteúdo da região de memória apontada por **zp** para a região de memória apontada por **zd** e devolvem a localização de **zd**. Está subjacente a qualquer das funções que **zd** referencia uma região de memória, cuja reserva de espaço de armazenamento foi previamente efectuada e que tem tamanho suficiente para que a transferência seja efectivamente possível.

Para as funções **memcpy** e **memmove**, a transferência é realizada sem ser atribuída qualquer interpretação ao conteúdo dos *bytes* transferidos, enquanto que, para **strcpy** e **strncpy**, se supõe que **zp** referencia uma cadeia de caracteres, ou seja, um agregado de caracteres terminado obrigatoriamente pelo carácter nulo.

No caso de **memcpy** e de **memmove**, são transferidos **n** bytes da região apontada por **zp** para a região apontada por **zd**. A diferença entre elas está no facto de que, em **memcpy**, as duas regiões têm que ser disjuntas, enquanto que, em **memmove**, o processo de cópia é feito primeiramente para uma região intermédia, o que possibilita a sobreposição parcelar das duas regiões.

No caso de **strcpy** e de **strncpy**, as regiões apontadas por **zp** e **zd** têm que ser disjuntas. O número de caracteres transferidos por **strcpy** é variável e depende do tamanho da cadeia de caracteres, concretamente, são copiados todos os caracteres que formam a cadeia de caracteres referenciada por **zp**, incluindo o carácter nulo final. Por outro lado, **strncpy** copia um máximo de **n** caracteres, incluindo o carácter nulo final, da cadeia de caracteres referenciada por **zp**. Se o tamanho da cadeia de caracteres for menor do que **n-1**, as posições restantes da região apontada por **zd** serão preenchidas com caracteres nulos. Se, pelo contrário, o tamanho da cadeia de caracteres for maior do que **n-1**, o resultado da cópia para a região de memória apontada por **zd** não constituirá uma cadeia de caracteres, porque falta o carácter nulo final.

A Figura 2.27 apresenta as funções de concatenação.

```
char *strcat (char *zd, char *zp);
char *strncat (char *zd, char *zp, size_t n);
```

Figura 2.27 - Funções de concatenação.

Ambas as funções copiam a cadeia de caracteres referenciada por **zp**, incluindo um carácter nulo final, para o fim da cadeia de caracteres referenciada por **zd** e devolvem a localização de **zd**. O ponto de junção é a localização do carácter nulo final da cadeia de caracteres referenciada por **zd**, que é substituído pelo primeiro carácter da cadeia de caracteres referenciada por **zp**. Está subjacente a qualquer das funções que **zd** aponta para uma região de memória, cuja reserva de espaço de armazenamento tem tamanho suficiente para conter a cadeia de caracteres resultante. No caso da função **strncat**, são copiados no máximo **n** caracteres, excluindo o carácter nulo final, da cadeia de caracteres referenciada por **zp**. As regiões de memória apontadas por **zp** e **zd** têm que ser disjuntas.

A Figura 2.28 apresenta um exemplo da utilização das funções **strcpy** e **strcat**.

```
char MENS1[] = "Ola", MENS2[] = "bom dia", MENS3[15];
...
strcpy (MENS3, MENS1);                    /* MENS3 é igual a "Ola" */
strcat (MENS3, " ");                 /* agora MENS3 é igual a "Ola " */
strcat (MENS3, MENS2);      /* agora MENS3 é igual a "Ola bom dia" */
```

Figura 2.28 - Exemplo da utilização das funções **strcpy** e **strcat**.

A Figura 2.29 apresenta as funções de comparação.

```
int memcmp (void *z1, void *z2, size_t n);
int strcmp (char *z1, char *z2);
int strncmp (char *z1, char *z2, size_t n);
```

Figura 2.29 - Funções de comparação.

As funções comparam *byte* a *byte* o conteúdo das regiões de memória referenciadas por **z1** e **z2**. O processo de comparação termina logo que uma decisão possa ser univocamente tomada.

O conteúdo da região **z1** diz-se menor do que o conteúdo da região **z2** quando, para o primeiro par de posições correspondentes que são distintas, se verifica que o conteúdo da posição de **z1** é menor do que o conteúdo da posição de **z2**, ambos os conteúdos interpretados como ***unsigned char***. Adicionalmente, o conteúdo da região **z1** é menor do que o conteúdo da região **z2** quando o tamanho da região **z1** é menor do que o tamanho da região **z2** e o conteúdo de cada posição de **z1** é igual ao conteúdo da posição correspondente de **z2**.

O conteúdo da região **z1** diz-se igual ao conteúdo da região **z2** quando as duas regiões têm o mesmo tamanho e o conteúdo de cada posição de **z1** é igual ao conteúdo da posição correspondente de **z2**. O conteúdo da região **z1** diz-se maior do que o conteúdo da região **z2** quando se tem em alternativa que o conteúdo da região **z2** é menor do que o conteúdo da região **z1**.

O valor devolvido será positivo, quando o conteúdo da região **z1** for maior do que o conteúdo da região **z2**, zero, quando o conteúdo da região **z1** for igual ao conteúdo da região **z2**, e, negativo, quando o conteúdo da região **z1** for menor do que o conteúdo da região **z2**.

No caso da função **memcmp**, comparam-se sempre **n** bytes sem ser atribuída qualquer interpretação ao seu conteúdo, enquanto que, para **strcmp** e **strncmp**, se supõe que **z1** e **z2** referenciam cadeias de caracteres, ou seja, agregados de caracteres terminados obrigatoriamente pelo carácter nulo, e o processo de comparação decorre até à tomada unívoca de uma decisão, no caso da função **strcmp**, ou até um máximo de **n** caracteres terem sido comparados no caso da função **strncmp**.

A Figura 2.30 apresenta um exemplo que lê e processa frases de texto, até um máximo de NMAX frases ou até ao aparecimento da frase "FIM". Quando se detecta a frase de terminação, usando para esse efeito a função **strcmp**, o ciclo repetitivo **while** é interrompido com a instrução **break**, de forma a evitar o processamento da frase de terminação. Como a função devolve 0 quando as duas cadeias de caracteres são iguais, então usa-se o operador **!** para que a expressão seja verdadeira. Ou seja, a expressão **!**strcmp(FRASE, FIM) é equivalente à expressão booleana strcmp(FRASE, FIM) == 0.

```
#define NMAX 10
...
char FRASE[80], FIM[] = "FIM"; int N = 0;
...
do
{
  N++;
  printf ("Escreva a frase -> ");
  scanf ("%79[^\n]", FRASE);                    /* leitura da frase */
  scanf ("%*[^\n]");       /* descartar todos os outros caracteres */
  scanf ("%*c");           /* descartar o carácter de fim de linha */
  if ( !strcmp (FRASE, FIM) )  break;
  ...                                          /* processar a frase lida */
} while ( N < NMAX );
```

Figura 2.30 - Exemplo da utilização da função **strcmp**.

A Figura 2.31 apresenta as funções de busca.

```
void *memchr (void *z, int c, size_t n);
char *strchr (void *z, int c);
char *strrchr (void *z, int c);
size_t strspn (char *z, char *zx);
size_t strcspn (char *z, char *zx);
char *strpbrk (char *z, char *zx);
char *strstr (char *z, char *zx);
char *strtok (char *z, char *zx);
```

Figura 2.31 - Funções de busca.

A função **memchr** localiza a primeira ocorrência do valor **c**, previamente convertido para ***unsigned char***, entre os **n** primeiros *bytes* da região de memória referenciada por **z**. O valor devolvido é a localização do valor em **z**, ou o **ponteiro nulo**, caso o valor não tenha sido encontrado. O **ponteiro nulo**, que está definido no ficheiro de interface *stddef.h*, é reconhecida pelo identificador **NULL** e representa o endereço da posição de memória 0.

As funções **strchr** e **strrchr** localizam, respectivamente, a primeira e a última ocorrência do valor **c**, previamente convertido para ***char***, na cadeia de caracteres referenciada por **z**. Neste contexto, supõe-se que o carácter nulo final faz parte da cadeia de caracteres. O valor devolvido é a localização do valor em **z**, ou o ponteiro nulo, caso o valor não tenha sido encontrado.

As funções **strspn** e **strcspn** calculam e devolvem o comprimento do segmento inicial máximo da cadeia de caracteres referenciada por **z** que consiste inteiramente de caracteres, respectivamente, presentes e não presentes, na cadeia de caracteres referenciada por **zx**.

A função **strpbrk** localiza a primeira ocorrência, na cadeia de caracteres referenciada por **z**, de qualquer carácter presente na cadeia de caracteres referenciada por **zx**. O valor devolvido é a localização do carácter, ou o ponteiro nulo, caso nenhum carácter tenha sido encontrado.

A função **strstr** localiza a primeira ocorrência, na cadeia de caracteres referenciada por **z**, da sequência de caracteres que constitui a cadeia de caracteres referenciada por **zx**. O valor devolvido é a localização da cadeia de caracteres, ou o ponteiro nulo, caso nenhum carácter tenha sido encontrado. Quando **zx** referencia uma cadeia de caracteres nula, ou seja, formado apenas pelo carácter nulo, o valor devolvido é **z**.

A função **strtok** permite decompor por invocações sucessivas a cadeia de caracteres referenciada por **z** num conjunto de subcadeias de caracteres. A decomposição baseia-se no princípio de que a cadeia de caracteres original é formado por uma sequência de palavras delimitadas por um ou mais caracteres separadores descritos na cadeia de caracteres referenciada por **zx**. O conjunto dos caracteres separadores pode variar de invocação para invocação.

A primeira invocação de **strtok** procura na cadeia de caracteres referenciada por **z** a primeira ocorrência de um carácter não contido na cadeia de caracteres referenciada por **zx**. Se tal carácter não existe, então não há palavras na cadeia de caracteres e a função devolve um ponteiro nulo. Se existir, esse carácter representa o início da primeira palavra. A seguir, a função procura na cadeia de caracteres a primeira ocorrência de um carácter contido na cadeia de caracteres separadores. Se tal carácter não existe, então a palavra actual estende-se até ao fim da cadeia de caracteres e as invocações subsequentes da função devolvem um

ponteiro nulo. Se existir, esse carácter é substituído pelo carácter nulo e constitui o fim da palavra actual, cuja localização é devolvida pela função. Antes de terminar, porém, a função **strtok** armazena internamente a localização do carácter seguinte ao carácter nulo. Esta referência vai constituir o ponto de partida da próxima pesquisa que, para ser feita, exige a invocação da função com um ponteiro nulo em substituição de **z**.

A Figura 2.32 apresenta um excerto de código da utilização da função **strtok** para decompor uma cadeia de caracteres constituída por palavras separadas pelos caracteres $ e #, e a Figura 2.33 apresenta a sua visualização gráfica. A primeira invocação da função detecta o início da primeira palavra a seguir à ocorrência do carácter #. Enquanto a função não devolver o ponteiro nulo, e o agregado de ponteiros para ***char*** tiver capacidade de armazenamento de informação, a função é invocada para encontrar o início da próxima palavra a seguir a um dos caracteres separadores. A segunda palavra começa a seguir ao carácter #, a terceira palavra começa a seguir ao carácter $ e a quarta palavra começa a seguir ao carácter #. Quando a função acaba de processar a frase, ela ficou decomposta em palavras, pelo que, ficou corrompida.

```
char FRASE[] = "#era$#uma$vez#um$"; char *PALAVRA[4], *PS; int N;
...
N = 0;
PS = strtok (FRASE, "$#");
while ( (PS != NULL) && (N < 4) )
{
  PALAVRA[N] = PS;
  N++;
  PS = strtok (NULL, "$#");
}
```

Figura 2.32 - Exemplo da utilização da função **strtok**.

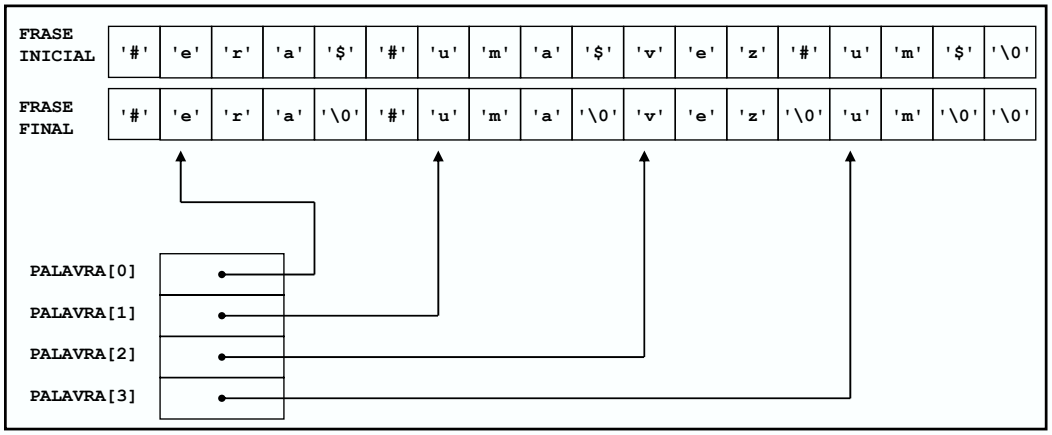

Figura 2.33 - Visualização gráfica do funcionamento da função **strtok.**

A Figura 2.34 apresenta as funções diversas.

```
void *memset (void *z, int c, size_t n);
char *strerror (int errnum);
size_t strlen (char *z);
```

Figura 2.34 - Funções diversas.

A função **memset** copia o valor ***c***, previamente convertido para ***unsigned char***, para os **n** primeiros *bytes* da região de memória referenciada por **z**. A função **strerror** devolve a mensagem associada com a variável global **errno**. A função **strlen** devolve o comprimento da cadeia de caracteres, ou seja, o número de caracteres armazenados até ao carácter nulo.

A Figura 2.35 apresenta um exemplo da utilização da função **strerror**. Esta função devolve uma cadeia de caracteres que pode ser usada para escrever mensagens de erro associadas à variável global de erro. Neste exemplo, logo após a invocação da função matemática raiz quadrada, a variável global de erro é testada para em caso de erro escrever a mensagem respectiva no monitor através do **printf**. No cálculo de uma raiz quadrada, há erro se o valor de X for negativo, pelo que, nesse caso a mensagem indicará que o argumento X está fora da gama permitida para o cálculo da raiz quadrada.

```
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <string.h>
...
errno = 0;
Z = sqrt (X);
if (errno != 0)                /* se ocorreu uma situação de erro */
{
   printf("ERRO -> %s\n", strerror(errno));
   ...                         -> Numerical argument out of domain */
}
```

Figura 2.35 - Exemplo da utilização da função **strerror**.

2.3.6 Conversão de cadeias de caracteres

A biblioteca ***stdio*** contém funções de conversão de cadeias de caracteres. Para utilizar essas funções é preciso fazer a inclusão do ficheiro de interface *stdio.h* com a seguinte directiva do pré-processador.

#include <stdio.h>

A Figura 2.36 apresenta as funções de conversão, que permitem criar e decompor cadeias de caracteres.

```
int sscanf (char *z, const char *formato, lista de ponteiros);
int sprintf (char *z, const char *formato, lista de expressões);
```

Figura 2.36 - Funções de conversão de cadeias de caracteres.

A função **sscanf** decompõe uma cadeia de caracteres referenciada por **z**, segundo as regras impostas pelo formato de leitura indicado por **formato**, armazenando sucessivamente os valores convertidos nas variáveis, cuja localização é indicada na lista de ponteiros de variáveis. As definições do formato e da lista de ponteiros de variáveis são as mesmas que para as funções **scanf** e **fscanf**.

A função **sscanf** é fundamentalmente equivalente a **fscanf**. A diferença principal é que a sequência de caracteres a converter é obtida da cadeia de caracteres referenciada por **z**, em vez de ser lida do ficheiro. Assim, a detecção do carácter nulo, que caracteriza a situação de se ter atingido o fim da cadeia de caracteres, vai corresponder à situação de detecção do carácter de *fim de ficheiro*.

A Figura 2.37 apresenta um exemplo da utilização da função **sscanf** para decompor uma cadeia de caracteres em subcadeias de caracteres. A cadeia de caracteres FRASE, que armazena uma data é decomposta nas suas componentes, CIDADE, DIA, MES e ANO, para posterior impressão no monitor com outro formato.

```
char FRASE[] = "Aveiro, 25 de Fevereiro de 2003";
char CIDADE[10], MES[10]; int DIA, ANO;
...
sscanf (FRASE, "%9[^,],%d de %9s de %d", CIDADE, &DIA, MES, &ANO);
                          /* CIDADE é igual a "Aveiro", DIA = 25, */
                          /* MES é igual a Fevereiro e ANO = 2003 */
...
printf ("(%s) %2d/%s/%4d\n", CIDADE, DIA, MES, ANO);
             /* é impresso no monitor (Aveiro) 25/Fevereiro/2003 */
```

Figura 2.37 - Exemplo da utilização da função **sscanf**.

A função **sprintf** cria uma cadeia de caracteres referenciada por **z**, constituída por texto e pelos valores das expressões que formam a lista de expressões, segundo as regras impostas pelo formato de escrita, indicado por **formato**. As definições do formato e da lista de expressões são as mesmas que para as funções **printf** e **fprintf**.

A função **sprintf** é fundamentalmente equivalente a **fprintf**. A diferença principal é que a sequência de caracteres convertida é armazenada na região de memória referenciada por **z**, em vez de ser escrita no ficheiro. Assim, torna-se necessário garantir que foi reservado previamente espaço de armazenamento suficiente para a sequência de caracteres convertida e para o carácter nulo final.

A Figura 2.38 apresenta um exemplo da utilização da função **sprintf** para a construção dinâmica de um formato de leitura para a função **scanf**. Se pretendermos fazer a leitura da cadeia de caracteres FRASE, que foi declarada com uma dimensão parametrizada pela constante MAX_CAR, não podemos usar o formato **%**MAX_CAR**s**, uma vez que o formato de leitura da função **scanf** só aceita literais. A solução passa pela construção dinâmica do formato de leitura. A cadeia de caracteres FORMATO é constituída pela concatenação do carácter %, através do especificador de conversão **%%**, com o valor de MAX_CAR, através do especificador de conversão **%d**, e com o carácter s, ou seja, armazena o formato de leitura **%40s**.

```
#define MAX_CAR 40
...
char FRASE[MAX_CAR+1], FORMATO[20];
...
sprintf (FORMATO, "%%%ds", MAX_CAR);  /* FORMATO é igual a "%40s" */
...
scanf (FORMATO, FRASE);   /* equivalente a scanf ("%40s", FRASE); */
```

Figura 2.38 - Exemplo da utilização da função **sprintf**.

2.4 Agregados bidimensionais e tridimensionais

Um agregado unidimensional é uma sequência linear de elementos que são acedidos através de um índice. No entanto, existem problemas em que a informação a ser processada é melhor representada através de uma estrutura de dados com um formato bidimensional, como por exemplo, uma tabela com várias colunas de informação, ou uma matriz no caso de aplicações matemáticas. Para esse tipo de aplicações precisamos de um agregado bidimensional.

Um agregado bidimensional é pois um agregado de agregados. A sua definição respeita as mesmas regras de um agregado unidimensional, sendo a única diferença o facto de ter dois descritores de dimensão em vez de um. Pode ser visto como uma estrutura composta por linhas, cujo número é definido pelo primeiro descritor de dimensão, e por colunas, cujo número é definido pelo segundo descritor de dimensão. Mas apesar desta visão bidimensional ele é armazenado na memória de forma linear, em endereços de memória contíguas, de maneira a simplificar o acesso aos seus elementos.

O acesso a um elemento do agregado bidimensional é feito através de dois índices, o primeiro para a linha e o segundo para a coluna, onde se encontra o elemento a que se pretende aceder. É preciso ter sempre em consideração que na linguagem C, os índices dos agregados são variáveis numéricas inteiras positivas e o índice do elemento localizado mais à esquerda em cada dimensão é o zero. Ao contrário do Turbo Pascal onde o acesso pode ser feito indiscriminadamente por A**[**L**][**C**]** ou A**[**L**,**C**]**, na linguagem C deve ser feito obrigatoriamente por A**[**L**][**C**]**.

A Figura 2.39 apresenta um exemplo da declaração de dois agregados, um unidimensional com 3 elementos inteiros e outro bidimensional com 6 elementos inteiros, bem como, da sua colocação na memória.

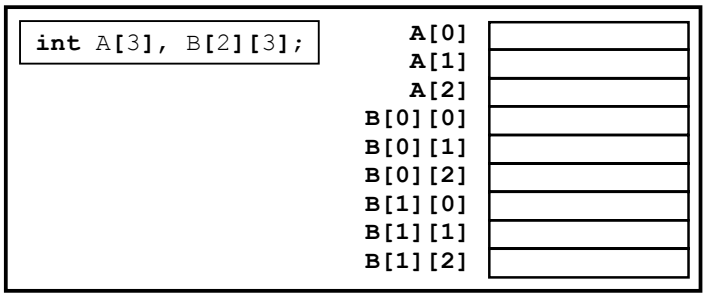

Figura 2.39 - Declaração de agregados unidimensionais e bidimensionais.

A Figura 2.40 apresenta um excerto de código que atribui valores aos agregados A e B. Para aceder a todos os elementos do agregado bidimensional B necessitamos de um duplo ciclo repetitivo **for**. Procure descobrir quais os valores que são armazenados nos agregados.

```
int A[3], B[2][3]; unsigned int I, J;
...
for (I = 0; I < 3; I++)  A[I] = I;

for (I = 0; I < 2; I++)
  for (J = 0; J < 3; J++)  B[I][J] = A[ (I+J)%3 ];
...
```

Figura 2.40 - Manipulação dos agregados.

A expressão de inicialização para variáveis de tipo agregado com N dimensões de um tipo base baseia-se no pressuposto que um tal agregado pode ser entendido como um agregado unidimensional de objectos, que são por sua vez, agregados com N−1 dimensões do tipo base. Assim, a aplicação sistemática deste pressuposto e as regras de inicialização de agregados unidimensionais permitem a construção da expressão de inicialização adequada a cada caso. Cada linha de elementos de inicialização deve ser inserida entre chavetas, de maneira a aumentar a legibilidade da inicialização.

A Figura 2.41 apresenta a declaração e inicialização de dois agregados bidimensionais, bem como da sua colocação na memória.

```
int B[2][3] = {
                {0, 1, 2},
                {3, 4, 5}
              };

int C[3][3] = {
                {10},
                {13, 14}

              };
```

B[0][0]	0
B[0][1]	1
B[0][2]	2
B[1][0]	3
B[1][1]	4
B[1][2]	5
C[0][0]	10
C[0][1]	0
C[0][2]	0
C[1][0]	13
C[1][1]	14
C[1][2]	0
C[2][0]	0
C[2][1]	0
C[2][2]	0

Figura 2.41 - Declaração e inicialização de agregados bidimensionais.

O agregado B, que tem 2×3 elementos, é inicializado com duas listas de inicialização, cada uma delas constituída por três constantes inteiras. Pelo que, todos os seus elementos são inicializados. O agregado C, que tem 3×3 elementos, é inicializado com apenas duas listas de inicialização. A primeira lista de inicialização é constituída apenas por uma constante, pelo que, o elemento C[0][0] é inicializado a 10 e os elementos C[0][1] e C[0][2] são inicializados a 0. A segunda lista de inicialização é constituída por duas constantes, pelo que, os elementos C[1][0] e C[1][1] são inicializados a 13 e a 14 respectivamente, e o elemento C[1][2] é inicializado a 0. Como não existe a terceira lista de inicialização, os elementos C[2][0], C[2][1] e C[2][2] são inicializados a 0.

Para além dos agregados bidimensionais, por vezes existe a necessidade de utilizar agregados tridimensionais. Por exemplo, para a simulação de um campo electromagnético no espaço. A Figura 2.42 apresenta a declaração e inicialização de dois agregados tridimensionais, bem como da sua colocação na memória. Tal como na declaração de agregados unidimensionais é possível omitir o descritor de dimensão, ou seja, fazer uma definição incompleta, mas apenas da primeira dimensão, tal como é feito na declaração do agregado E. A primeira dimensão pode ser inferida pelo compilador a partir da expressão de inicialização, e neste exemplo é 2.

```
int D[2][2][2] = {
                   {
                     {0, 1},
                     {2, 3}
                   },
                   {
                     {4, 5},
                     {6, 7}
                   }
                 };

int E[ ][2][2] = {
                   {
                     {10},
                     {12, 13}
                   },
                   {
                     {14}

                   }
                 };
```

D[0][0][0]	0
D[0][0][1]	1
D[0][1][0]	2
D[0][1][1]	3
D[1][0][0]	4
D[1][0][1]	5
D[1][1][0]	6
D[1][1][1]	7
E[0][0][0]	10
E[0][0][1]	0
E[0][1][0]	12
E[0][1][1]	13
E[1][0][0]	14
E[1][0][1]	0
E[1][1][0]	0
E[1][1][1]	0

Figura 2.42 - Declaração e inicialização de agregados tridimensionais.

O agregado D, que tem 2×2×2 elementos, é inicializado com quatro listas de inicialização, cada uma delas constituída por 2 constantes inteiras. Pelo que, todos os seus elementos são inicializados. O agregado E, que tem a mesma dimensão, é inicializado com apenas três listas de inicialização com um total de 4 constantes. A primeira lista de inicialização é constituída apenas por uma constante, pelo que, o elemento E[0][0][0] é inicializado a 10 e o elemento E[0][0][1] é inicializado a 0. A segunda lista de inicialização é constituída por duas constantes, pelo que, os elementos E[0][1][0] e E[0][1][1] são inicializados a 12 e a 13 respectivamente. A terceira lista de inicialização é constituída por apenas uma constante, pelo que, o elemento E[1][0][0] é inicializado a 14 e o elemento E[1][0][1] é inicializado a 0. Como não existe a quarta lista de inicialização, os elementos E[1][1][0] e E[1][1][1] são inicializados a 0.

Mas, apesar de normalmente não serem necessários agregados com mais de duas ou três dimensões, a linguagem C tal como o Pascal não limita o número de dimensões de um agregado. Segundo a norma ANSI, os compiladores devem suportar pelo menos seis dimensões.

Na passagem de agregados multidimensionais a uma função é preciso passar um ponteiro para o início do agregado, usando para o efeito o nome do agregado seguido dos parênteses rectos, tal como, na passagem de um agregado unidimensional. Mas, como o elemento inicial de um agregado multidimensional é também um agregado, é obrigatório indicar o número de elementos de cada uma das N−1 dimensões à direita. Ou seja, apenas a dimensão mais à esquerda pode ser omitida. A Figura 2.43 apresenta a definição de uma função que inicializa a zero todos os elementos de um agregado tridimensional. A função tem um parâmetro de entrada-saída que é o agregado a inicializar e um parâmetro de entrada que é a primeira dimensão do agregado.

```
void INICIALIZAR (int AM[][M][N], int NE)  /* definição da função */
{
  int I, J, K;
  for (I = 0; I < NE; I++)
    for (J = 0; J < M; J++)
      for (K = 0; K < N; K++)
        AM[I][J][K] = 0;   /* para aceder ao elemento AM[I][J][K] */
}
```

```
int AMULT[L][M][N];
...
INICIALIZAR (AMULT, L);                    /* invocação da função */
```

Figura 2.43 - Passagem de um agregado tridimensional a uma função.

Uma forma alternativa de fazer a passagem do agregado, consiste em passar explicitamente um ponteiro para o primeiro elemento do agregado e indicar todas as dimensões do agregado como parâmetros de entrada auxiliares. A Figura 2.44 apresenta esta forma alternativa. O parâmetro de entrada-saída que representa o início do agregado é passado como sendo um ponteiro para ponteiro para ponteiro para inteiro, através da declaração **int** AM**[ ][ ][ ]**, ou em alternativa **int * * ***AM.

```
void INICIALIZAR (int AM[][][], int X, int Y, int Z)
{                                          /* definição da função */
  int I, J, K;
  for (I = 0; I < X; I++)
    for (J = 0; J < Y; J++)
      for (K = 0; K < Z; K++)
        *((int *) AM + I*Y*Z + J*Z + K) = 0;
}                          /* para aceder ao elemento AM[I][J][K] */
```

```
int AMULT[L][M][N];
...
INICIALIZAR (AMULT, L, M, N);              /* invocação da função */
```

Figura 2.44 - Passagem de um agregado tridimensional a uma função (versão alternativa).

A vantagem desta solução é que a implementação não tem a necessidade de saber quais são as dimensões do agregado e, portanto, a função é genérica. Mas, tem a desvantagem de o programador necessitar de executar a aritmética de ponteiros para aceder aos elementos do agregado. Uma vez que o parâmetro passado à função é do tipo **int * * ***, para aceder aos elementos do agregado usando o operador apontado por, o endereço calculado tem de ser convertido explicitamente num ponteiro para o tipo de elementos do agregado, neste caso o tipo inteiro, daí o uso do operador **cast (int *)**.

2.5 Ponteiros

Como já foi referido anteriormente, um ponteiro embora assuma o valor de um endereço, não é propriamente um endereço, porque ele está sempre associado a um tipo de dados bem definido. Daí que duas variáveis de tipo ponteiro podem conter o mesmo valor, ou seja, o mesmo endereço, e no entanto constituírem de facto entidades distintas.

Esta diferença é apresentada no exemplo da Figura 2.45. Embora as variáveis PI e PD contenham o mesmo valor após a execução das instruções de atribuição, elas constituem de facto entidades distintas. O ponteiro PI referencia a região de memória de armazenamento de uma variável de tipo ***int***, representada na figura pela área mais escura com o tamanho de 4 *bytes*, enquanto que o ponteiro PD referencia a região de memória de armazenamento de uma variável de tipo ***double***, representada na figura pelas duas áreas escura e clara com o tamanho de 8 *bytes*.

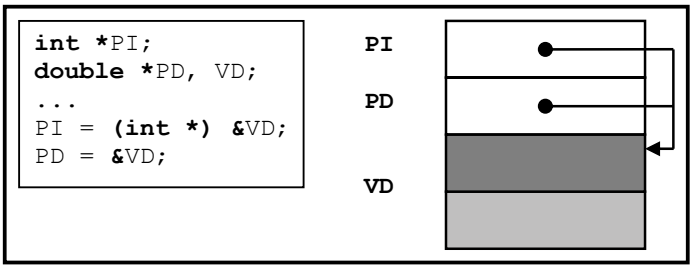

Figura 2.45 - Exemplo da diferença entre um ponteiro e um endereço.

A declaração de variáveis de tipo **ponteiro** (***pointer***) segue a regra geral de declaração de variáveis na linguagem C, cuja definição formal se apresenta na Figura 2.46.

tipo de dados genérico lista de variáveis de tipo ponteiro **;**

tipo de dados genérico ::= qualquer tipo de dados válido na linguagem C

lista de variáveis de tipo ponteiro ::= *identificador de variável de tipo ponteiro* |
lista de variáveis de tipo ponteiro **,** *identificador de variável de tipo ponteiro*

identificador de variável de tipo ponteiro ::= ***** *identificador de variável genérico* |
*****identificador de variável tipo ponteiro*

identificador de variável genérico ::= *identificador de variável* |
identificador de variável **=** *expressão de inicialização*

Figura 2.46 - Definição formal da declaração de variáveis de tipo ***ponteiro***.

Podemos declarar um ponteiro de um qualquer tipo de dados genérico. O tipo de dados genérico inclui os tipos de dados nativos numéricos anteriormente referidos e os tipos de dados definidos pelo utilizador que serão referidos adiante. Além destes, inclui ainda um tipo de dados nativo não numérico, o tipo ***void***, que tem um significado muito especial. Este tipo representa literalmente um tipo de dados que é **coisa nenhuma**. Assim, não é possível definir-se variáveis deste tipo, mas unicamente ponteiros para variáveis deste tipo. Trata-se da forma encontrada pela linguagem C para caracterizar **ponteiros genéricos** e, portanto, o seu uso é bastante restrito.

A Figura 2.47 apresenta um exemplo da utilização de um ponteiro de tipo ***void***. A primeira linha de código declara uma variável A de tipo ***int***. A segunda linha declara uma variável V de tipo ***void***, o que é ilegal e portanto, dá erro de compilação. A terceira linha declara uma variável de tipo ponteiro para ***void***, ou seja, um ponteiro genérico. A atribuição *PV = 5 é uma instrução ilegal, apesar de PV estar a apontar para A, porque o ponteiro PV é de tipo ***void***. Para colocar o valor 5 na variável A que é de tipo ***int***, através do ponteiro PV, é preciso fazer um **cast** do ponteiro PV, como se mostra na última linha.

```
int A;
void V;  /* declaração ilegal porque a variável V não tem sentido */
void *PV;        /* a variável PV representa um ponteiro genérico */
...
PV = &A;
*PV = 5;/* instrução ilegal porque a variável *PV não tem sentido */
*((int *) PV) = 5;                                    /* agora A = 5 */
```

Figura 2.47 - Exemplo da declaração e utilização de um ponteiro de tipo ***void***.

Como se mostra na Figura 2.48, a declaração de variáveis de tipo ponteiro não precisa de ser feita numa linha separada, pelo que, a lista de variáveis pode incluir conjuntamente variáveis do tipo indicado, ponteiros para variáveis do tipo indicado, ponteiros para ponteiros para variáveis do tipo indicado, e assim sucessivamente. A expressão de inicialização para variáveis de tipo ponteiro consiste alternativamente na localização de uma variável, previamente declarada, do tipo para o qual aponta a variável de tipo ponteiro, usando o operador endereço, ou na constante **NULL**, que é o ***ponteiro nulo*** e que representa o valor de um ponteiro que não localiza qualquer região de memória.

```
int A, *PA = &A, **PPA = NULL;
/* A é uma variável de tipo int, PA é uma variável de tipo ponteiro
para int, inicializada com o endereço de A e PPA é uma variável de
tipo ponteiro para ponteiro para int inicializada a NULL */
```

Figura 2.48 - Exemplo da declaração de variáveis e de ponteiros de tipo ***int***.

Dado que ponteiros em abstracto não fazem sentido, na atribuição de um valor a uma variável de tipo ponteiro é obrigatório que o tipo de dados associado à variável e à expressão seja o mesmo. As únicas excepções são a atribuição a uma variável de tipo ponteiro para ***void*** de uma expressão que é um ponteiro para um tipo qualquer, e a atribuição a uma variável que é um ponteiro para um tipo qualquer de uma expressão de tipo ponteiro para ***void***. A Figura 2.49 apresenta um exemplo demonstrativo.

```
int A, *PA = &A, **PPA = &PA;
double *PD;
void *PV, **PPV;
...
PD = PA;              /* incorrecto porque são ponteiros distintos */
PD = (double *) PA;           /* correcto após conversão forçada */
PV = PA;          /* correcto porque PV é um ponteiro de tipo void */
PD = PV;          /* correcto porque PV é um ponteiro de tipo void */
PPV = PPA;           /* incorrecto porque são ponteiros distintos */
```

Figura 2.49 - Exemplos da declaração e utilização de ponteiros de tipos diferentes.

Na Figura 2.49 a variável PPA foi declarada do tipo ponteiro para ponteiro para ***int***. A Figura 2.50 apresenta a sua visualização gráfica e a atribuição do valor 45 à variável A, através de uma dupla referência indirecta com a instrução de atribuição ******PPA = 45.

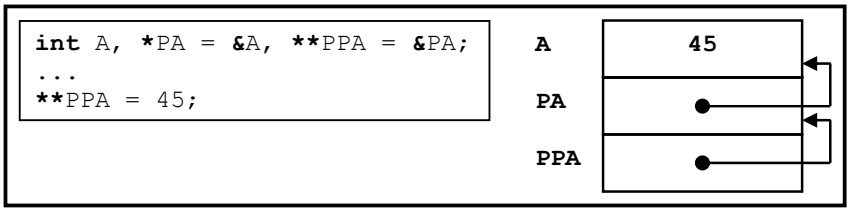

Figura 2.50 - Visualização gráfica de um ponteiro para ponteiro para ***int***.

A Figura 2.51 apresenta um erro frequentemente cometido, por programadores que se estão a iniciar na utilização da linguagem C. Nunca se deve fazer a atribuição de um valor a uma variável apontado por um ponteiro, sem que este tenha sido previamente inicializado. Por uma questão de segurança deve-se sempre inicializar um ponteiro, quanto mais não seja com a constante **NULL**.

```
int A, *PA;
...
*PA = 23;      /* incorrecto porque PA não aponta para lado nenhum */
```

```
int A, *PA = &A;
...
*PA = 23;                                          /* correcto A = 23 */
```

Figura 2.51 - Exemplo de má utilização de um ponteiro devido a falta de inicialização.

2.5.1 Aritmética de ponteiros

Como um ponteiro armazena um valor inteiro, que representa o endereço do primeiro *byte* da região de armazenamento da variável para o qual ele aponta, sobre ele pode ser realizada a operação aritmética da adição de um valor inteiro. O significado atribuído à operação é colocar o ponteiro a referenciar a região de memória que se encontra localizada mais abaixo, mais à frente, se a expressão inteira for positiva, ou mais acima, mais atrás, se a expressão inteira for negativa, e a uma distância que é medida em termos de unidades de armazenamento de variáveis do tipo associado ao ponteiro. Logo, a realização da operação só faz sentido para ponteiros para tipos de dados específicos, pelo que, não pode por isso ser aplicada a expressões de tipo ponteiro para ***void***. O ponteiro avança ou recua o número de *bytes* equivalente ao **sizeof (***tipo de dados***)** por cada unidade adicionada. Sobre um ponteiro, também se pode aplicar a operação de subtracção de um valor inteiro, cujo significado é obviamente o inverso da adição.

Também é possível fazer a subtracção de dois ponteiros do mesmo tipo. O significado atribuído à operação é determinar a distância entre os dois ponteiros, medida em termos de unidades de armazenamento de variáveis do tipo associado. De novo, a realização da operação só faz sentido para ponteiros para tipos de dados específicos, pelo que, não pode ser aplicada a expressões de tipo ponteiro para ***void***.

Sobre ponteiros também é possível fazer operações de comparação, usando os operadores relacionais, sendo que, os dois ponteiros têm de ser do mesmo tipo. A Figura 2.52 apresenta alguns exemplos destas operações.

```
int A[8], *PA = A, N;                          /* PA aponta para A[0] */
...
PA++;      /* agora PA aponta para A[1], ou seja PA avança 4 bytes */
PA += 4; /* agora PA aponta para A[5], ou seja PA avança 16 bytes */
PA -= 2;   /* agora PA aponta para A[3], ou seja PA recua 8 bytes */
...
N = &A[3] - &A[0];                                          /* N = 3 */
N = &A[0] - &A[3];                                         /* N = -3 */
...
if (PA > &A[2]) ...          /* a expresão decisória é verdadeira */
```

Figura 2.52 - Operações aritméticas sobre ponteiros.

2.5.2 Agregados de ponteiros e ponteiros para agregados

A linguagem C potencia a construção de tipos de dados muito versáteis, onde há a combinação do construtor agregado **[ ]** com o operador apontado por ***** na sua definição. A legibilidade resultante é por vezes, contudo, muito pobre e exige uma compreensão rigorosa da associatividade e da precedência dos operadores envolvidos. A Figura 2.53 apresenta a precedência e a associatividade dos novos operadores.

Operadores na classe	Associatividade	Precedência
operadores primários		
( )	esquerda → direita	**maior**
[ ]	esquerda → direita	
referência a campo **->**	esquerda → direita	
acesso a campo **.**	esquerda → direita	
operadores unários		↓
operador **cast**	direita → esquerda	
operador **sizeof**	direita → esquerda	
operador endereço **&**	direita → esquerda	
operador apontado por *****	direita → esquerda	**menor**

Figura 2.53 - Precedência e associatividade entre os operadores.

Os operadores parênteses curvos **( )** e parênteses rectos **[ ]** são designados de operadores primários. Têm associatividade da esquerda para a direita e maior precedência que os operadores unários e binários. Os operadores endereço **&** e apontado por ***** são operadores unários, cuja associatividade é da direita para a esquerda. Têm uma precedência maior do que os operadores binários, mas menor do que os operadores unários anteriormente apresentados, ou seja, menor do que os operadores **cast** e **sizeof**. Os operadores referência a campo **->** e acesso a campo **.** serão explicados no item de acesso a campos de estruturas.

Aplicando as regras de associatividade e precedência dos operadores, temos que a declaração TIPO_BASE *****AP**[**4**];** declara um agregado de 4 ponteiros para TIPO_BASE, cujo mapa de reserva de espaço em memória se apresenta na Figura 2.54.

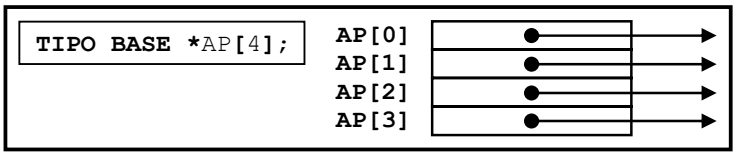

Figura 2.54 - Visualização gráfica de um agregado de ponteiros.

Uma das aplicações dos agregados de ponteiros é a construção de agregados de cadeias de caracteres, que é apresentado na Figura 2.57 e na Figura 2.58.

Aplicando as regras de associatividade e precedência dos operadores, temos que a declaração TIPO_BASE **(***PA**)[**4**];** declara um ponteiro para um agregado de 4 elementos de TIPO_BASE, cujo mapa de reserva de espaço em memória se apresenta na Figura 2.55.

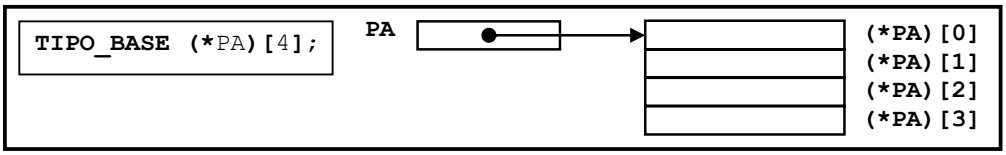

Figura 2.55 - Visualização gráfica de um ponteiro para um agregado.

2.5.3 Dualidade ponteiro agregado

Já referimos que devido à **dualidade ponteiro agregado**, quer uma variável de tipo agregado unidimensional de um dado tipo base, quer uma variável de tipo ponteiro para o mesmo tipo base, são formas equivalentes de referenciar uma região de memória formada por um conjunto contíguo de variáveis do tipo base. Pelo que, a localização do elemento de índice **i** do agregado **A** pode ser alternativamente expressa por **&A[i]** ou por **A+i**, e o valor do mesmo elemento por **A[i]** ou por ***(A+i)**.

Se entendermos um agregado bidimensional do tipo base como um agregado unidimensional de objectos que são, por sua vez, agregados unidimensionais do tipo base, temos que.

TIPO_BASE B[N1][N2]; ⇒ B[i] ≡ &B[i][0] , B ≡ &B[0]

Neste contexto, **B[i]** representa o agregado unidimensional, de índice **i**, de N2 elementos do tipo base ou, o que é equivalente, um ponteiro para o primeiro elemento desse agregado e, **B** representa um ponteiro para o primeiro agregado unidimensional de N2 elementos do tipo base. Ou seja, a localização do elemento, colocado na linha de índice **i** e na coluna de índice **j** do agregado **B**, pode ser alternativamente expressa por **&B[i][j]** ou por **B[i]+j**, e o valor do mesmo elemento por **B[i][j]** ou por ***(B[i]+j)**, ou ainda por ***(*(B+i)+j)**.

Ou seja, uma variável de tipo ponteiro para um agregado unidimensional de N2 elementos do tipo base pode ser encarada como uma variável de tipo agregado bidimensional do mesmo tipo base, com tamanho N2 na segunda dimensão, na referência a uma região de memória formada por um conjunto contíguo de variáveis do tipo base. Pelo que, a seguinte declaração do agregado B e do ponteiro para o agregado PB inicializado com o endereço inicial do agregado, resulta no mapa de reserva de espaço em memória que se apresenta na Figura 2.56 e permite a utilização do ponteiro PB para aceder aos elementos do agregado.

TIPO_BASE B[N1][N2], (*PB)[N2] = B;

PB+i ≡ &PB[i] = **&B[i]** ⇒ ***(PB+i)** ≡ PB[i] = **B[i]** ⇒ ***(PB+i)+j** ≡ PB[i]+j = **&B[i][j]**
***(*(PB+i)+j)** ≡ *(PB[i]+j) ≡ (*(PB+i))[j] ≡ **PB[i][j] = B[i][j]**, com $0 \le i < N1$, $0 \le j < N2$

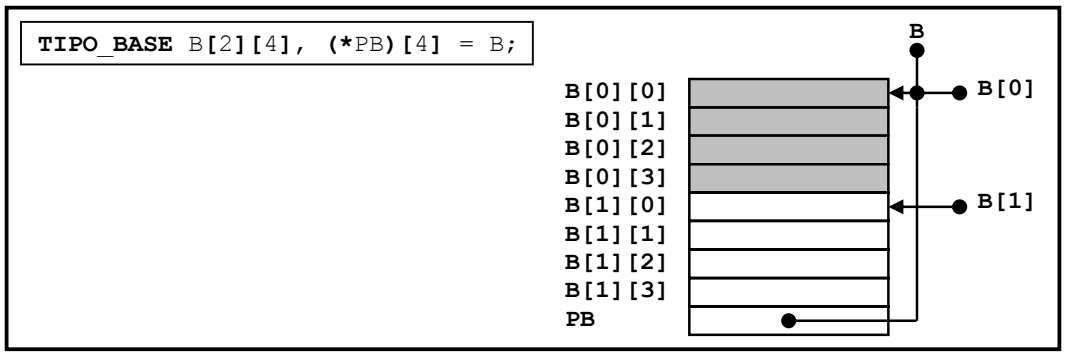

Figura 2.56 - Declaração de um agregado bidimensional e de um ponteiro para o agregado

Existe, porém, uma diferença subtil entre as variáveis de tipo agregado bidimensional de um tipo base e as variáveis de tipo ponteiro para um agregado unidimensional, de tamanho igual à segunda dimensão do agregado bidimensional do mesmo tipo base. Como se verifica do mapa de reserva de espaço em memória apresentado na Figura 2.56, não existe espaço directamente associado com a variável B e, portanto, B, B[0] e B[1] são ponteiros constantes, cujos valores não podem ser modificados. Assim, instruções do tipo B = expressão**;** ou B[i] = expressão**;** são ilegais e tem-se que *(B+i) = B[i].

Perante a declaração TIPO_BASE A**[**N1**]**, B**[**N1**][**N2**]**, C**[**N1**][**N2**][**N3**];** tem-se que:

- A é um ponteiro constante para TIPO_BASE.
- B é um ponteiro constante para um agregado unidimensional de N2 elementos do TIPO_BASE e B[i] é um ponteiro constante para TIPO_BASE, com $0 \leq i < N1$.
- C é um ponteiro constante para um agregado bidimensional de N2×N3 elementos do TIPO_BASE, C[i] é um ponteiro constante para um agregado unidimensional de N3 elementos do TIPO_BASE, com $0 \leq i < N1$, e, C[i][j] é um ponteiro constante para TIPO_BASE, com $0 \leq i < N1$ e $0 \leq j < N2$.

2.5.4 Agregados de cadeias de caracteres (*arrays of strings*)

Uma das aplicações dos agregados de ponteiros é a construção de agregados de cadeias de caracteres (*array of strings*) constantes. A Figura 2.57 apresenta a visualização gráfica do agregado FLORES, declarado da seguinte forma.

char ***FLORES[**4**]** = **{** "rosa", "dahlia", "cravo" **};**

O agregado FLORES é um agregado de ponteiros para ***char***, em que cada elemento aponta para o primeiro carácter de uma cadeia de caracteres. Como o quarto elemento não foi inicializado, então aponta para NULL.

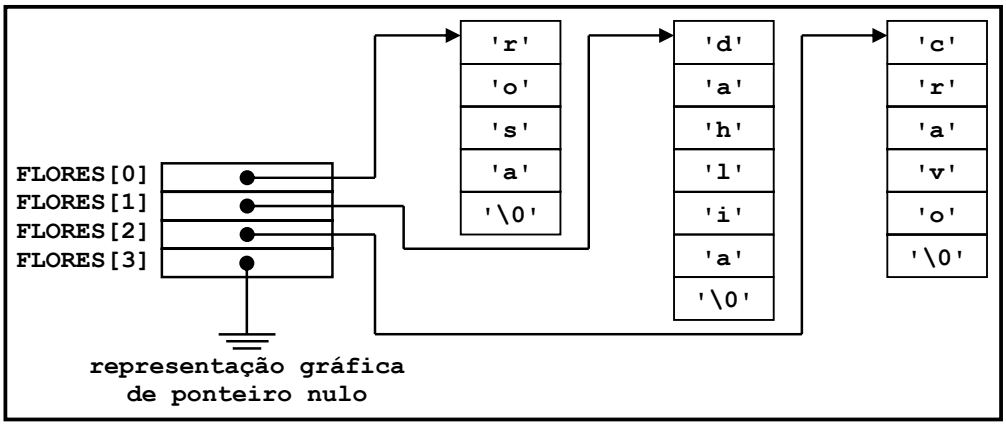

Figura 2.57 - Visualização gráfica de um agregado de cadeias de caracteres.

Uma aplicação de um agregado de cadeias de caracteres constantes é a conversão do mês em numérico para o mês por extenso, cuja função se apresenta na Figura 2.58. Esta é uma situação em que é natural que o agregado comece no índice um, de modo a poupar a operação de subtracção de uma unidade ao número do mês, quando se retira do agregado a cadeia de caracteres pretendida. Pelo que, o agregado tem treze cadeias de caracteres e utiliza-se o índice zero para armazenar uma cadeia de caracteres que vai ser usada para assinalar situações de erro.

Quando a função é invocada para um mês incorrecto, a função devolve a cadeia de caracteres de índice zero, ou seja, "MêsErrado". Quando o número do mês está correcto, a função devolve a cadeia de caracteres, cujo índice é o mês numérico.

```
type TMES = string[9];                                    (* no Pascal *)
...
function MES_EXTENSO (MES: integer): TMES;
begin
  case MES of
      1: MES_EXTENSO := 'Janeiro';    2: MES_EXTENSO := 'Fevereiro';
      3: MES_EXTENSO := 'Março';      4: MES_EXTENSO := 'Abril';
      5: MES_EXTENSO := 'Maio';       6: MES_EXTENSO := 'Junho';
      7: MES_EXTENSO := 'Julho';      8: MES_EXTENSO := 'Agosto';
      9: MES_EXTENSO := 'Setembro';  10: MES_EXTENSO := 'Outubro';
     11: MES_EXTENSO := 'Novembro';  12: MES_EXTENSO := 'Dezembro';
    else MES_EXTENSO := 'MesErrado'
  end
end;
```

```
char *MES_EXTENSO (int MES)                          /* na linguagem C */
{
  char *MES_EXTENSO[13] = { "MêsErrado", "Janeiro", "Fevereiro",
           "Março", "Abril", "Maio", "Junho", "Julho", "Agosto",
                  "Setembro", "Outubro", "Novembro", "Dezembro" };

  if ( (MES < 1)  || (MES > 12) )  return MES_EXTENSO[0];

  return MES_EXTENSO[MES];
}
```

Figura 2.58 - Exemplo da função da conversão do mês em numérico para o mês por extenso.

2.6 Estruturas (*structs*)

Apesar de um agregado ser uma estrutura de dados muito útil, porque permite a manipulação de uma grande quantidade de dados através de um acesso indexado, tem no entanto a limitação de todos os seus elementos serem do mesmo tipo. Em muitas situações precisamos de armazenar informação relacionada entre si, mas que são de tipos diferentes. Por exemplo, o registo de um aluno universitário necessita de conter a informação relativa ao nome do aluno, a informação sobre o seu bilhete de identidade, a morada, a nacionalidade, a data de nascimento, o curso em que está inscrito, o número mecanográfico, o ano que está a frequentar, a lista de disciplinas a que está inscrito, e para cada uma delas o código, o nome e o número de créditos, e o histórico das disciplinas a que já esteve inscrito. Parte desta informação é texto e portanto, pode ser armazenada em cadeias de caracteres, outra é de tipo numérica e a lista de disciplinas e o histórico das disciplinas é de tipo agregado. Para podermos armazenar toda esta informação numa única estrutura de dados, precisamos de um tipo de dados que permita que os seus elementos possam ser de tipos diferentes. Para esse efeito a linguagem C providencia o tipo de dados **estrutura** (***struct***), que é equivalente ao **registo** (***record***) do Pascal.

Uma estrutura distingue-se assim de um agregado pelo facto de permitir que os seus elementos, que se designam por campos, possam ser de tipos diferentes e porque o acesso a cada um dos campos não é feito através da sua localização na estrutura, mas sim através do nome do campo a que se pretende aceder. Estamos perante um **acesso por nomeação**.

2.6.1 Declaração, inicialização e atribuição de estruturas

A Figura 2.59 apresenta a definição formal da declaração de uma estrutura, que recorre ao construtor **struct**. A lista de campos é inserida entre chavetas, e segue a regra geral de declaração de variáveis da linguagem C. Cada lista de campos começa com a declaração do tipo de dados a que pertencem os campos, seguida do nome de um ou mais campos separados por vírgulas, e é terminada com o separador **;**.

struct *identificador da estrutura*
{
lista de campos **;**
} ;

identificador da estrutura ::= identificador válido na linguagem C

lista de campos ::= *elemento campo* | *lista de campos* **;** *elemento campo*

elemento campo ::= *tipo de dados* *lista de nomes de campos*

tipo de dados ::= qualquer tipo de dados válido na linguagem C

lista de nomes de campos ::= *identificador de campo* |
lista de nomes de campos **,** *identificador de campo*

identificador de campo ::= identificador válido na linguagem C

Figura 2.59 - Definição formal da declaração de uma estrutura (***struct***).

A Figura 2.60 apresenta o exemplo da declaração da estrutura **tdados_pessoa**, que permite armazenar os dados pessoais de uma pessoa, composta pelo nome, sexo e data de nascimento, que por sua vez é composta pelo dia, mês e ano.

```
struct tdados_pessoa
{
  char NOME[60];
  char SEXO;
  unsigned int DIA;
  unsigned int MES;
  unsigned int ANO;
};
```

Figura 2.60 - Exemplo da declaração de uma estrutura.

A declaração da estrutura **tdados_pessoa** indica que de agora em diante o compilador reconhece um novo tipo de dados designado por **struct tdados_pessoa**. Pelo que, para declararmos variáveis deste tipo, aplicam-se as mesmas regras da declaração de variáveis dos tipos nativos da linguagem C. A Figura 2.61 apresenta a declaração de uma variável, de um agregado e de um ponteiro deste novo tipo de dados.

```
struct tdados_pessoa PESSOA, GRUPO_PESSOAS[10], *PPESSOA;
```

Figura 2.61 - Exemplo da declaração de variáveis e ponteiros do tipo **struct** tdados_pessoa.

Em alternativa é possível definir o tipo de dados **struct tdados_pessoa** e declarar variáveis desse tipo na mesma instrução, tal como se apresenta na Figura 2.62.

```
struct tdados_pessoa
{
   char NOME[60];
   char SEXO;
   unsigned int DIA;
   unsigned int MES;
   unsigned int ANO;
} PESSOA, GRUPO PESSOAS[10], *PPESSOA;
```

Figura 2.62 - Exemplo da declaração de uma estrutura e da declaração de variáveis desse tipo.

Existe uma outra forma de criar um novo tipo de dados em linguagem C usando para esse efeito a instrução **typedef**, que é equivalente à instrução **type** do Pascal. A Figura 2.63 apresenta esta forma alternativa.

```
typedef struct
{
   char NOME[60];
   char SEXO;
   unsigned int DIA;
   unsigned int MES;
   unsigned int ANO;
} TDADOS_PESSOA;
```

Figura 2.63 - Exemplo da definição de uma estrutura usando o **typedef**.

Enquanto que o identificador tdados_pessoa definia apenas a estrutura, agora TDADOS_PESSOA é um identificador de toda a declaração da estrutura incluindo a palavra reservada **struct**. Para distinguir o tipo de dados do identificador da estrutura, normalmente usa-se o mesmo identificador, mas em caracteres maiúsculos. Quando se define uma estrutura através do **typedef** é possível omitir o identificador da estrutura anteriormente usado, ou seja, o identificador tdados_pessoa.

Na definição de uma estrutura usando o **typedef**, não podem ser declaradas, nem inicializadas, variáveis. Estas têm de ser declaradas à parte. A Figura 2.64 apresenta a declaração de uma variável, de um agregado e de um ponteiro do tipo de dados TDADOS_PESSOA.

```
TDADOS_PESSOA  PESSOA, GRUPO_PESSOAS[10], *PPESSOA;
```

Figura 2.64 - Exemplo da declaração de variáveis e ponteiros do tipo TDADOS_PESSOA.

As definições de tipos de dados usando o **typedef**, são colocados no início dos ficheiros fonte logo após as directivas de *include*, de maneira a tornar o tipo de dados visível por todo o programa, ou num ficheiro de interface, que tem a extensão **.h**, que é depois aludido nos ficheiros fonte onde o tipo de dados é necessário.

É possível inicializar uma estrutura da mesma forma que se inicializa um agregado. A Figura 2.65 apresenta a declaração e inicialização de uma estrutura de dados do tipo TDADOS_PESSOA.

```
TDADOS_PESSOA  PESSOA = { "Vincent Van Gogh", 'M', 30, 3, 1853 };
```

Figura 2.65 - Exemplo da declaração e inicialização de uma variável do tipo TDADOS_PESSOA.

Na linguagem C é possível atribuir uma estrutura a outra estrutura, tal como no Pascal. A Figura 2.66 apresenta alguns exemplos de instruções de atribuição envolvendo estruturas.

```
TDADOS_PESSOA FUNC_PESSOA (void);  /* alusão à função FUNC_PESSOA */
...
TDADOS_PESSOA  PESSOA1 = { "Vincent Van Gogh", 'M', 30, 3, 1853 };
TDADOS_PESSOA  PESSOA2, *PPESSOA;
...
PESSOA2 = PESSOA1;
...
PPESSOA = &PESSOA1;
PESSOA2 = *PPESSOA;
...
PESSOA2 = FUNC_PESSOA ();
```

Figura 2.66 - Exemplos da atribuição de estruturas.

2.6.2 Acesso aos campos de estruturas

Na linguagem C existem duas formas de aceder aos campos de uma estrutura. Se estivermos na presença de uma variável do tipo **struct**, usa-se o nome da variável e o nome do campo separados por um ponto, tal e qual como no Pascal. Vamos designar o operador **.** por operador acesso a campo. Se estivermos na presença de um ponteiro para uma variável do tipo **struct**, então usa-se o nome do ponteiro para a variável e o nome do campo separados pelo operador **->**. Vamos designar o operador **->** por operador referência a campo. Este operador é uma abreviatura da linguagem C, que combina o operador apontado por ***** com o operador acesso a campo **.**, pelo que, a instrução PPESSOA**->**DIA é equivalente a **(***PPESSOA**).**DIA.

A Figura 2.67 mostra a atribuição do valor 30 ao campo DIA, directamente através de uma variável, usando o operador acesso a campo, e por referência indirecta através de um ponteiro, usando o operador referência a campo.

```
TDADOS_PESSOA  PESSOA, *PPESSOA = &PESSOA;
...              /* acesso ao campo DIA através da variável PESSOA */
PESSOA.DIA = 30;
...             /* acesso ao campo DIA através do ponteiro PPESSOA */
PPESSOA->DIA = 30;        /* é equivalente a (*PPESSOA).DIA = 30; */
```

Figura 2.67 - Acesso aos campos de uma estrutura.

Na formação de expressões de acesso aos campos de uma estrutura, é preciso ter em conta que os operadores de acesso aos campos de estruturas são operadores primários, com associatividade da esquerda para a direita e com menor precedência do que os operadores primários **( )** e **[ ]**, tal como se mostra na Figura 2.53.

2.6.3 Estruturas hierárquicas

Tal como no Pascal, também a linguagem C permite que um campo de uma estrutura possa ser uma estrutura, o que se designa por estruturas hierárquicas (*nested structures*). Uma vez que a data constitui um tipo de informação muito frequente em programação, faz sentido que seja previamente declarada como uma estrutura autónoma e depois seja usada na declaração de outras estruturas.

A Figura 2.68 apresenta a decomposição da estrutura TDADOS_PESSOA, autonomizando a data na estrutura TDATA.

```
typedef struct
{
   unsigned int DIA;
   unsigned int MES;
   unsigned int ANO;
} TDATA;

typedef struct
{
   char NOME[60];
   char SEXO;
   TDATA DATA_NASCIMENTO;
} TDADOS_PESSOA;
```

Figura 2.68 - Exemplo da declaração de estruturas hierárquicas.

Esta declaração hierarquizada em dois níveis provoca algumas consequências na inicialização da estrutura TDADOS_PESSOA, bem como, no acesso aos campos da data. Como se mostra na Figura 2.69, agora a inicialização da estrutura TDADOS_PESSOA, implica a inicialização da estrutura TDATA entre chavetas, num segundo nível de inicialização, tal como se de um agregado bidimensional se tratasse.

```
TDADOS_PESSOA  PESSOA = { "Vincent Van Gogh", 'M', {30, 3, 1853} };
```

Figura 2.69 - Exemplo da declaração e inicialização de uma estrutura hierárquica.

A Figura 2.70 apresenta as consequências em termos de acesso aos campos da estrutura TDATA, que agora são campos de um campo da estrutura TDADOS_PESSOA.

```
TDADOS_PESSOA  PESSOA, *PPESSOA = &PESSOA;

...     /* acesso ao campo DIA da DATA através da variável PESSOA */
PESSOA.DATA_NASCIMENTO.DIA = 30;

...    /* acesso ao campo DIA da DATA através do ponteiro PPESSOA */
PPESSOA->DATA_NASCIMENTO.DIA = 30;
```

Figura 2.70 - Acesso aos campos de uma estrutura interna a outra estrutura.

2.6.4 Estruturas ligadas

Uma estrutura não pode conter uma instanciação da própria estrutura, mas pode conter ponteiros para a estrutura, permitindo assim a criação de estruturas ligadas. No entanto, neste caso é necessário usar um identificador para designar a estrutura, que depois vai ser usado como identificador do tipo de dados dos campos de tipo ponteiro, tal como se mostra na Figura 2.71. O tipo TNODO para além dos campos que vão conter a informação a armazenar na estrutura, tem um campo de tipo ponteiro que aponta para a própria estrutura TNODO, através do identificador de tipo **struct** tnodo.

```
typedef struct tnodo
{
   ...;
   ...;
   struct tnodo *PNODO;
} TNODO;
```

Figura 2.71 - Definição de uma estrutura com referência a si própria.

Uma estrutura pode conter ponteiros para estruturas ainda não definidas. A Figura 2.72 apresenta duas estruturas que se referenciam mutuamente. Cada estrutura tem campos que vão conter a informação a armazenar na estrutura, e um campo de tipo ponteiro que aponta para a outra estrutura.

```
typedef struct ts1
{
   ...;
   ...;
   struct ts2 *PST;
} TS1;

typedef struct ts2
{
   ...;
   ...;
   struct ts1 *PST;
} TS2;
```

Figura 2.72 - Definição de duas estruturas que se referenciam mutuamente.

2.6.5 Estruturas como parâmetros de funções

Uma estrutura pode ser passada a uma função por valor ou por referência. Passar uma estrutura por referência é normalmente mais rápido, porque apenas é feito a cópia do ponteiro para a estrutura. Enquanto que a passagem por valor implica a criação de uma cópia local da estrutura na função. Portanto, uma estrutura é passado por valor quando ela não necessita de ser alterada pela execução da função e se quer garantir que a função não altera a informação nela armazenada, ou quando a estrutura é muito pequena. Caso contrário, a estrutura deve ser passada por referência.

A Figura 2.73 apresenta a função de leitura da estrutura TDADOS_PESSOA. Como a estrutura é um parâmetro de entrada-saída da função, tem de ser passada por referência.

```
void LER_DADOS_PESSOA (TDADOS_PESSOA *PESSOA)
{                                          /* definição da função */
   ...
}
-------------------------------------------------------------------
TDADOS_PESSOA  PESSOA;
...
LER DADOS PESSOA (&PESSOA);                /* invocação da função */
```

Figura 2.73 - Definição e invocação de uma função com uma estrutura passada por referência.

A Figura 2.74 apresenta a função de escrita da estrutura TDADOS_PESSOA. Como a estrutura é um parâmetro de entrada da função, pode ser passada por valor.

```
void ESCREVER_DADOS_PESSOA (TDADOS_PESSOA PESSOA)
{                                                  /* definição da função */
   ...
}
```

```
TDADOS_PESSOA  PESSOA;
...
ESCREVER DADOS PESSOA (PESSOA);                    /* invocação da função */
```

Figura 2.74 - Definição e invocação de uma função com uma estrutura passada por valor.

2.6.6 Estruturas como resultado de saída de funções

Uma função pode devolver uma estrutura. No entanto, é mais eficiente devolver um ponteiro para a estrutura. Mas, para isso, temos que declarar a estrutura como sendo de duração permanente, usando para o efeito o qualificativo **static**. Senão, a estrutura deixa de existir assim que a função termina a sua execução, uma vez que por defeito as variáveis das funções são automáticas, ou seja, são alocadas quando a função é invocada e destruídas quando a função termina a execução. A Figura 2.75 apresenta as duas soluções para uma função que devolve um resultado de tipo TDATA.

```
TDATA FUNC_DATA_E (void)
{               /* o resultado de saída da função é de tipo TDATA */
   TDATA DATA;
   ...
   return DATA;                                 /* retorno da estrutura */
}
```

```
TDATA *FUNC_DATA_P (void)
{ /* o resultado de saída da função é de tipo ponteiro para TDATA */
   static TDATA DATA;
   ...
   return &DATA;           /* retorno do ponteiro para a estrutura */
}
```

Figura 2.75 - Devolução de uma estrutura por uma função.

2.6.7 Agregados de estruturas

Uma vez definida uma estrutura é possível declarar agregados de estruturas. A Figura 2.76 apresenta uma função que processa um agregado de estruturas de tipo TDADOS_PESSOA e que calcula o número de pessoas do sexo masculino existentes no agregado. Nesta primeira versão, o acesso aos elementos do agregado é feito através do índice do elemento pretendido.

```
int NUMERO_HOMENS (TDADOS_PESSOA GRUPO_PESSOAS[ ], int N)
{
   int I, NUM = 0;
   for (I = 0; I < N; I++)
     if (GRUPO_PESSOAS[I].SEXO == 'M') NUM++;
   return NUM;
}
```

Figura 2.76 - Exemplo de uma função que processa um agregado de estruturas (1ª versão).

A Figura 2.77 apresenta uma segunda versão, em que o acesso aos elementos do agregado é feito através de um ponteiro que aponta para o elemento pretendido.

```
int NUMERO_HOMENS (TDADOS_PESSOA GRUPO_PESSOAS[ ], int N)
{
  int I, NUM = 0; TDADOS_PESSOA *P = GRUPO_PESSOAS;
  for (I = 0; I < N; P++, I++)
    if (P->SEXO == 'M') NUM++;
  return NUM;
}
```

Figura 2.77 - Exemplo de uma função que processas um agregado de estruturas (2ª versão).

A Figura 2.78 apresenta o programa que invoca a função NUMERO_HOMENS. O número de elementos do agregado é determinado recorrendo ao operador **sizeof**. O tipo de dados TDADOS_PESSOA é o da Figura 2.68.

```
int NUMERO_HOMENS (TDADOS_PESSOA [ ], int);    /* alusão à função */
...
int main (void)
{
  TDADOS_PESSOA PESSOAS[] = {
                           {"Vincent Van Gogh",'M',{30, 3, 1853}},
                           {"Vieira da Silva",'F',{13, 6, 1908}}
                           {"Amedeo Modigliani",'M',{12, 7, 1884}},
                           {"Claude Monet",'M',{14, 11, 1840}},
                           {"Georgia O'Keeffe",'F',{15, 11, 1887}}
                             };
  int NHOMENS, NEST;

  NEST = sizeof (PESSOAS) / sizeof (PESSOAS[0]);
  NHOMENS = NUMERO_HOMENS (PESSOAS, NEST); /* invocação da função */
  ...
  return 0;
}
```

Figura 2.78 - Programa que invoca a função NUMERO_HOMENS.

2.7 Tipo enumerado (*enum*)

Na linguagem C também é possível declarar tipos de dados enumerados (***enum***), cuja definição formal se apresenta na Figura 2.79. É um tipo de dados escalar, em que se enumera a lista de valores associados a uma variável deste tipo. Cada valor é definido como um identificador constante no bloco que contém a definição de tipo, bloco esse que é inserido entre chavetas. Os identificadores constantes são numerados pelo compilador, sendo atribuído ao primeiro identificador da lista o valor 0, ao segundo o valor 1, e assim sucessivamente. Podemos no entanto, alterar esta numeração e indicar para cada identificador um valor específico, de forma a utilizar valores mais explanatórios. Sempre que não se atribui um valor a um dos identificadores, o compilador atribui-lhe o valor do identificador anterior incrementado de uma unidade.

Tal como no caso da definição de estruturas, a definição de enumerados também pode ser feita usando a instrução **typedef**.

enum *identificador do enumerado* **{** *lista de identificadores* **} ;**

identificador do enumerado ::= identificador válido na linguagem C

lista de identificadores ::= *identificador genérico* | *lista de identificadores* **,** *identificador genérico*

identificador genérico ::= *identificador definido pelo utilizador* |
identificador definido pelo utilizador **=** *valor de inicialização*

identificador definido pelo utilizador ::= identificador válido na linguagem C

valor de inicialização ::= valor decimal positivo ou nulo

Figura 2.79 - Definição formal da declaração de um tipo enumerado.

A Figura 2.80 apresenta exemplos da declaração de tipos enumerados. O primeiro exemplo declara o enumerado **t_cor** e na mesma instrução a variável COR. O segundo exemplo define o enumerado **t_dia_util** e numa declaração à parte variáveis desse tipo.

```
enum t_cor { BRANCO, AZUL, VERDE, ROSA, PRETO } COR;
```

```
enum t_dia_util { SEGUNDA=2, TERCA=3, QUARTA=4, QUINTA=5, SEXTA=6 };
...
enum t_dia_util  DIA_SEMANA, DIAS[5], *PDIA;
```

Figura 2.80 - Exemplos da declaração de variáveis de tipos enumerados.

A Figura 2.81 apresenta os mesmos exemplos, mas recorrendo ao **typedef** para definir primeiro os tipos de dados enumerados **T_COR** e **T_DIA_UTIL**, ficando a declaração das variáveis para depois. Como os valores atribuídos aos identificadores da lista do dia da semana são seguidos, então atribui-se apenas o valor inicial ao primeiro elemento da lista.

```
typedef enum { BRANCO, AZUL, VERDE, ROSA, PRETO } T_COR;
...
T_COR  COR;
```

```
typedef enum { SEGUNDA=2, TERCA, QUARTA, QUINTA, SEXTA } T_DIA_UTIL;
...
T_DIA_UTIL  DIA_SEMANA, DIAS[5], *PDIA;
```

Figura 2.81 - Exemplos da definição de tipos enumerados usando o **typedef**.

Normalmente, as definições de tipos de dados enumerados usando o **typedef**, são colocados no início dos ficheiros fonte logo após as directivas de *include*, de maneira a tornar o tipo de dados visível por todo o programa, ou num ficheiro de interface, que é depois aludido nos ficheiros fonte onde o tipo de dados é necessário.

O tipo de dados enumerado é um tipo escalar, cujos identificadores são reconhecidos através do seu valor numérico. Pelo que, é possível ler variáveis deste tipo directamente do teclado, usando o formato decimal e o valor numérico correspondente ao identificador. Também é possível escrever variáveis deste tipo directamente no monitor, se bem que neste caso o valor impresso poderá não ser muito esclarecedor. Também é possível executar operações aritméticas sobre variáveis enumeradas, bem como, operações de comparação utilizando os operadores relacionais. A Figura 2.82 apresenta alguns exemplos.

```
T_DIA_UTIL  DIA;
...
do
{
   printf ("Dia da semana (SEGUNDA=2 ... SEXTA=6)? ");
   scanf ("%1d", &DIA);
} while (DIA<2 || DIA>6);
   /* admitindo que foi introduzido o valor 2, então DIA = SEGUNDA */
DIA += 4;                                        /* agora DIA = SEXTA */
DIA--;                                          /* agora DIA = QUINTA */
...
if (DIA > SEXTA)                               /* a expressão é falsa */
   printf ("*** Aleluia é Fim de Semana ***\n");
```

Figura 2.82 - Exemplos de operações sobre variáveis de tipo enumerado.

2.8 Classes de armazenamento

Normalmente uma variável declarada numa função, designa-se por variável automática, porque a memória é alocada automaticamente quando da invocação da função e libertada quando termina a execução da função. Pelo que, o seu endereço pode ser diferente para cada invocação da função. Como existe apenas durante a execução da função também é designada de variável com duração temporária. No entanto, é possível criar uma variável com duração permanente. Ou seja, alocar um endereço de memória fixo para a variável. Para tal declara-se a variável dentro da função com o qualificativo **static**. A memória, necessária para armazenar uma variável **static**, é alocada no início do programa e o seu endereço é fixo até ao fim do programa. Por outro lado, a variável só é inicializada da primeira vez que a função é invocada, permitindo assim que a variável conserve o seu valor em invocações sucessivas da função. A Figura 2.83 mostra um exemplo que demonstra a diferença entre a variável temporária TEMP e a variável permanente PERM.

```
void INCREMENTAR (void)
{
   int TEMP = 1;  static int PERM = 1;
   TEMP++;
   PERM++;
   printf("Temporaria = %d <-> Permanente = %d\n", TEMP, PERM);
}
```

```
int main (void)
{
   INCREMENTAR ();            /* Temporaria = 2 <-> Permanente = 2 */
   INCREMENTAR ();            /* Temporaria = 2 <-> Permanente = 3 */
   INCREMENTAR ();            /* Temporaria = 2 <-> Permanente = 4 */
   return 0;
}
```

Figura 2.83 - Exemplo demonstrativo da diferença entre variáveis temporárias e permanentes.

Quando a função é invocada pela primeira vez as variáveis são inicializadas a 1 e depois são ambas incrementadas adquirindo o valor 2. Na segunda invocação da função, a variável TEMP é de novo inicializada a 1 e toma o valor 2, enquanto que, a variável PERM é apenas incrementada e como conserva o valor anterior, agora toma o valor 3. Na terceira invocação da função esta actuação é repetida, pelo que, a variável TEMP toma outra vez o valor 2, enquanto que, a variável PERM toma o valor 4.

Uma variável de duração permanente só pode ser inicializada com uma expressão que contenha apenas literais. Se a variável não for inicializada dentro de um função, então fica automaticamente inicializada a zero.

A linguagem C permite que o programador peça ao compilador para colocar o conteúdo de uma variável num registo do processador. As operações envolvendo dados armazenados em registos são mais rápidas que as operações envolvendo dados armazenados em posições de memória, porque, evitam a perda de tempo necessária a que que os dados sejam trazidos da memória para o processador e que o resultado da operação seja colocado de novo na memória. Para esse efeito declara-se a variável com o qualificativo **register**. Infelizmente, o número de registos existentes no processador é limitado, pelo que, não há qualquer garantia que o pedido seja respeitado.

Porque as variáveis declaradas com o qualificativo **register**, podem não existir na memória, o operador endereço não pode ser usado sobre essas variáveis, ou seja, os registos não são endereçáveis. Só podem ser declaradas com o qualificativo **register** variáveis declaradas dentro de funções. Esta prorrogativa deve ser apenas usada para variáveis que são acedidas frequentemente.

A linguagem C também permite declarar variáveis, cujo valor não pode ser alterado depois da sua inicialização. Nesse caso estamos perante uma constante que ao contrário de uma constante definida com a directiva de *define*, que é designada por constante simbólica, ocupa espaço em memória. Para declarar uma constante usa-se o qualificativo **const**. A Figura 2.84 apresenta uma nova versão da função CONVERTE_DISTANCIA em que o factor de conversão é uma constante real local.

```
double CONVERTE_DISTANCIA (double ML)       /* definição da função */
{
   const double  MIL_QUI = 1.609;
   return ML * MIL_QUI;
}
```

Figura 2.84 - Exemplo da utilização de uma constante.

A grande vantagem de declarar uma variável constante é assegurar que a variável está protegida contra escrita e portanto, não perde o seu valor original. Estas variáveis são muitas vezes utilizadas como parâmetros de entrada de funções, como por exemplo em funções da biblioteca de execução ANSI ***string***, de forma a assegurar que o parâmetro não é corrompido pela execução da função. Veja por exemplo o protótipo da função **strcpy**.

char * strcpy (**char *** zd, **const char *** zp);

No caso da declaração de ponteiros, a palavra reservada **const**, pode aparecer de duas formas distintas, tendo por isso significados diferentes. A seguinte declaração, declara um ponteiro constante para um inteiro, ou seja, um ponteiro que aponta sempre para o mesmo endereço de memória.

int *const PCONSTANTE;

No entanto, a seguinte declaração, declara um ponteiro que aponta para um inteiro constante, ou seja, um ponteiro que pode apontar para qualquer endereço, desde que este seja o de uma variável de tipo inteiro e constante.

int const * PINTCONST;

2.9 Visibilidade dos objectos

Uma vez que a linguagem C permite a construção de aplicações distribuídas por vários ficheiros fonte, é muito importante a forma como é possível partilhar informação entre eles, bem como possibilitar a duplicação de identificadores.

O nível de visibilidade de uma variável depende do sítio onde esta é declarada. Existem quatro níveis de visibilidade. As variáveis declaradas antes da função **main** são variáveis globais que têm um alcance para todos os ficheiros fonte da aplicação. Vamos designá-las por **variáveis globalmente globais** (*Program Scope*). As variáveis declaradas antes da função **main** com o qualificativo **static** são variáveis globais que têm um alcance apenas no ficheiro fonte onde estão declaradas. Vamos designá-las por **variáveis localmente globais** (*File Scope*). As variáveis declaradas dentro das funções são variáveis locais que têm o alcance limitado à função onde são declaradas. Vamos designá-las por **variáveis locais** (*Function Scope*). As variáveis declaradas dentro de um bloco, ou seja, dentro de um conjunto de intruções inseridas entre chavetas são variáveis que têm o alcance limitado ao bloco onde estão declaradas. Vamos designá-las por **variáveis de bloco** (*Block Scope*). Por uma questão de bom estilo de programação, estas variáveis devem ser evitadas.

A Figura 2.85 apresenta exemplos de declaração de variáveis com diferentes níveis de visibilidade.

```
#include <stdio.h>

TIPO1 VARIAVEL_1;                    /* Variável Globalmente Global */

static TIPO2 VARIAVEL_2;              /* Variável Localmente Global */

int main (void)
{
   TIPO3 VARIAVEL_3;                                 /* Variável Local */
   ...
}
```

Figura 2.85 - Exemplos de declaração de variáveis com diferentes níveis de visibilidade.

É preciso ter em atenção ao duplo significado do qualificativo **static**. Quando uma variável **static** é declarada fora de uma função, significa que a variável é global mas com alcance circunscrito ao ficheiro. Quando uma variável **static** é declarada dentro de uma função, significa que a variável é local mas de duração permanente.

As variáveis globais são armazenadas na memória RAM, enquanto que as variáveis locais são armazenadas na memória *stack*. Uma função é como uma caixa preta, em que os detalhes associados com a implementação da operação, nomeadamente as variáveis locais, são invisíveis externamente. Mas, por outro lado, todas as variáveis declaradas globalmente são visíveis dentro das funções. Em caso de conflito, a variável instanciada é aquela que está declarada mais perto.

Numa função não é possível declarar uma variável local com o mesmo nome de um parâmetro da função. Mas, como as variáveis locais são invisíveis externamente, pode-se declarar variáveis locais com o mesmo nome em funções diferentes. A Figura 2.86 apresenta exemplos de declaração e utilização de variáveis com diferente visibilidade.

```
static int I, J;                    /* variáveis localmente globais */

int main (void)
{
   int I;                                  /* variável local I do main */
   ...
   FUNC (I);                                       /* variável local I */
   ...
}

int FUNC (int N)
{
   int N;             /* ilegal, porque N é um parâmetro de entrada */
   int I;                           /* variável local I da função FUNC */
   ...
   I = N*J;  /* variável local I = parâmetro N * variável global J */
   ...
}
```

Figura 2.86 - Exemplos de declaração e utilização de variáveis com diferente visibilidade.

Quando temos uma aplicação distribuída por vários ficheiros fonte, é por vezes necessário que uma função de um ficheiro aceda a variáveis globalmente globais que foram declaradas noutros ficheiros. Até agora estivemos a considerar que cada declaração de uma variável produz alocação de memória para a variável. No entanto, a alocação de memória só é feita quando há uma definição de uma variável. Uma variável global pode ser declarada sem produzir alocação de memória. Essa declaração chama-se **alusão** e usa-se para o efeito o qualificativo **extern**. A Figura 2.87 apresenta a alusão a uma variável globalmente global. O objectivo de uma alusão é permitir que o compilador faça a verificação de tipos. As alusões de variáveis são normalmente colocadas em ficheiros cabeçalho, assegurando assim alusões consistentes. Para definir uma variável global, a variável deve ser declarada sem o qualificativo **extern** e deve incluir a inicialização da variável. Para aludir uma variável global ela deve ser declarada com o qualificativo **extern** e não deve incluir qualquer inicialização.

```
int main (void)
{
   int I;                           /* declaração da variável local I */
   extern int J;          /* alusão à variável globalmente global J */
   ...
}
```

Figura 2.87 - Exemplo da alusão a uma variável global externa.

Por vezes também existe a necessidade de que uma função seja apenas visível no ficheiro onde está definida. Desta forma, pode-se definir funções com o mesmo nome em ficheiros diferentes. Para esse efeito, coloca-se o qualificativo **static** antes da definição da função. É normalmente aplicado a funções internas a outras funções, que portanto, não têm a necessidade de serem reconhecidas fora do ficheiro onde estão definidas.

2.10 Leituras recomendadas

- 7º, 8º, 9º e 10º capítulos do livro "C A Software Approach", 3ª edição, de Peter A. Darnell e Philip E. Margolis, da editora Springer-Verlag, 1996.

Capítulo 3

FICHEIROS EM C

Sumário

Em muitas aplicações práticas, como por exemplo na utilização de bases de dados, a quantidade de informação de entrada é tanta, que o simples facto de termos que a introduzir de novo sempre que executamos o programa torna-o pouco funcional. Por outro lado, neste tipo de aplicações os resultados de saída precisam de ser salvaguardados para posterior utilização. O tipo ficheiro é um tipo estruturado que tem como suporte de armazenamento a memória de massa, e não a memória principal do computador, pelo que, um ficheiro não só armazena dados a título definitivo, como também permite a comunicação de informação entre programas. Como um ficheiro é também uma estrutura de dados dinâmica, permite também ultrapassar a limitação das estruturas de dados estáticas, como são os agregados de registos, pelo que, é a estrutura de dados adequada para suportar o processamento de bases de dados.

A norma ANSI da linguagem C cria um modelo uniforme de acesso aos diferentes dispositivos do sistema computacional. Quer se trate dos dispositivos de entrada e de saída, ou seja, dos periféricos, ou de ficheiros localizados na memória de massa, o acesso é sempre efectuado associando um fluxo de comunicação ao dispositivo. Um fluxo de comunicação é uma sequência ordenada de *bytes*, armazenada numa dada região da memória principal, que materializa o fluxo de dados entre o programa e o dispositivo. Portanto, ao contrário da linguagem Pascal, em que um ficheiro é visto como tendo uma estrutura interna ou registo associado, na linguagem C um ficheiro não é mais do que uma sequência ordenada de *bytes*. Ler de, ou escrever para, um dado ficheiro, significa portanto, ler do, ou escrever no, fluxo de comunicação associado ao ficheiro. Na linguagem C existem dois tipos de fluxos de comunicação, que são os fluxos de texto e os fluxos binários. Vamos apresentar os dois através de exemplos de aplicação.

3.1 Fluxos de comunicação

A norma ANSI cria um modelo uniforme de acesso aos diferentes dispositivos de entrada e de saída do sistema computacional. Quer se trate dos dispositivos convencionais de entrada, que é o teclado, de saída que é o monitor, *scanners*, impressoras, ou de ficheiros localizados na memória de massa, o acesso é sempre efectuado associando um **fluxo de comunicação** (*stream*) ao dispositivo. Um fluxo de comunicação é visto como uma sequência ordenada de *bytes*, armazenada numa dada região da memória principal, que materializa o fluxo de dados entre o programa e o dispositivo. A transferência de informação pode ser unidireccional ou bidireccional. Ler de, ou escrever para, um dado dispositivo, significa portanto, ler do, ou escrever no, fluxo de comunicação associado ao dispositivo. É o sistema operativo que se encarrega da transferência propriamente dita entre o fluxo de comunicação e o dispositivo, garantindo-se assim a portabilidade.

Em transferências unidireccionais, a leitura ou a escrita são sequenciais, iniciando-se, respectivamente, no princípio ou no fim da informação aí residente, enquanto que, em transferências bidireccionais, há por vezes a possibilidade de acesso aleatório para leitura e/ou para escrita.

A norma ANSI define no ficheiro de interface ***stdio.h*** uma estrutura de dados, de nome **FILE**, que mantém toda a informação necessária ao controlo de um fluxo de comunicação e que consiste, entre outros elementos:

- Num indicador de posição de leitura ou de escrita.
- Num ponteiro para a localização do armazenamento tampão (*buffer*) associado ao fluxo de comunicação.
- Num sinalizador de erro, que indica se ocorreu um erro de leitura ou de escrita.
- Num sinalizador de fim de ficheiro, que indica que o fim da informação armazenada no dispositivo foi atingido.

Assim, sempre que um programa pretender aceder a um dispositivo tem que declarar uma variável de tipo **ponteiro para FILE** para armazenamento do identificador do fluxo de comunicação que é devolvido pela operação de estabelecimento de comunicação. A partir daí, esse identificador é usado em todas as operações de leitura e/ou de escrita no dispositivo, bem como no controlo e encerramento da comunicação.

A norma ANSI distingue dois tipos de fluxos de comunicação. Os **fluxos de texto** e os **fluxos binários**.

- Nos **fluxos de texto**, a sequência ordenada de *bytes* é interpretada como uma sequência de caracteres, organizada em linhas que consistem em zero ou mais caracteres, com representação gráfica ou com funções de controlo, seguidos do carácter de *fim de linha*, que é o carácter '\n'.
- Nos **fluxos binários**, a sequência ordenada de *bytes* não sofre qualquer interpretação e exprime o modo de representação da informação em memória.

Em geral, quando um programa é posto em execução, é automaticamente estabelecida a comunicação com os dispositivos convencionais de entrada e de saída. No caso da linguagem C, a norma ANSI impõe a inicialização automática dos três fluxos de texto seguintes:

- O fluxo de texto **stdin** está associado com o dispositivo convencional de entrada, que é o teclado.

- O fluxo de texto **stdout** está associado com o dispositivo convencional de saída, que é o monitor.

- O fluxo de texto **stderr** está associado com o dispositivo convencional de saída de erro, que também é o monitor.

Para garantir a portabilidade, o ficheiro de interface ***stdio.h***, define ainda duas constantes. A constante **EOF** (*end of file*), que sinaliza o fim de ficheiro e a constante **NULL**, que é o ponteiro nulo, ou seja o valor de um ponteiro que não localiza qualquer região de memória.

3.2 Abertura ou criação de um fluxo de comunicação

Como já foi referido, para que seja possível ler ou escrever num dispositivo, é necessário associar-se-lhe um fluxo de comunicação. Isso é feito invocando a função **fopen**, que se apresenta na Figura 3.1, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

FILE ***fopen **(const char ***nome do dispositivo **, const char ***modo de acesso **) ;

nome do dispositivo ::= nome válido para um dispositivo genérico

modo de acesso ::= "r" | "rb" | "w" | "wb" | "a" | "ab"
"r+" | "rb+" | "w+" | "wb+" | "a+" | "ab+"

Figura 3.1 - Função **fopen**.

A função **fopen** estabelece a associação de um fluxo de comunicação com o dispositivo pretendido, segundo as regras impostas pelo modo de acesso. A associação é feita criando e inicializando em memória uma estrutura de dados de tipo **FILE**. A função devolve, quando bem sucedida, a localização em memória da estrutura definida, que vai funcionar na prática como identificador do fluxo de comunicação correspondente, ou, quando falha, um ponteiro nulo. No caso de insucesso, a causa é sinalizada na variável global de erro **errno**. O nome do dispositivo é essencialmente uma cadeia de caracteres que identifica um dispositivo específico do sistema computacional ou um ficheiro do sistema de ficheiros. O formato concreto depende do sistema operativo presente. A diferença entre os modos de acesso sem e com o carácter **b**, é que no primeiro caso o fluxo de comunicação associado é de tipo fluxo de texto, enquanto que no segundo caso é de tipo fluxo binário. A função tem os seguintes modos de acesso:

- **r Abertura de um dispositivo já existente para leitura.**

A leitura começa no princípio do conteúdo armazenado no ficheiro, ou da informação que for entretanto introduzida no dispositivo convencional de entrada.

- **r+ Abertura de um dispositivo já existente para leitura e escrita.**

A leitura ou a escrita começam no princípio do conteúdo aí armazenado, por cima do conteúdo anterior. Este modo é usado tipicamente para ficheiros.

- **w Abertura ou criação de um dispositivo para escrita.**

Qualquer conteúdo previamente armazenado é destruído e a escrita começa numa situação de ficheiro vazio, ou de início de comunicação no caso do dispositivo convencional de saída.

- **w+ Abertura ou criação de um dispositivo para leitura e escrita.**

Qualquer conteúdo previamente armazenado é destruído e a leitura ou a escrita começam numa situação de dispositivo vazio. Este modo é usado tipicamente para ficheiros.

- **a Abertura ou criação de um dispositivo para escrita no fim.**

A escrita só pode ser efectuada no fim do conteúdo já existente. Este modo é usado tipicamente para ficheiros.

- **a+ Abertura ou criação de um dispositivo para escrita no fim e leitura.**

A escrita só pode ser efectuada no fim do conteúdo já existente, a leitura pode ocorrer em qualquer ponto. Este modo é usado tipicamente para ficheiros.

A Figura 3.2 sumaria as permissões dos modos de acesso.

Modo de acesso						
	r	**w**	**a**	**r+**	**w+**	**a+**
Existência prévia do dispositivo	*			*		
Dispositivo aberto ou criado sem conteúdo		*			*	
Leitura do fluxo permitida	*			*	*	*
Escrita no fluxo permitida		*	*	*	*	*
Escrita permitida só no fim do fluxo			*			*

Figura 3.2 - Permissões dos modos de acesso.

As condições em que se processa a comunicação com o dispositivo podem ser monitorizadas através de um conjunto de funções que testam os elementos, sinalizador de erro e sinalizador de fim de ficheiro da estrutura de dados **FILE**, e a variável global de erro **errno**. Estas funções, cuja descrição está contida no ficheiro de interface ***stdio.h***, são apresentadas na Figura 3.3.

void clearerr **(** FILE ***fluxo** **) ;**
int ferror **(** FILE ***fluxo** **) ;**
int feof **(** FILE ***fluxo** **) ;**
void perror **(const char ***mensagem definida pelo programador **) ;**

fluxo ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen**, **freopen**, ***stdin***, ***stdout*** ou ***stderr***

mensagem definida pelo programador :: = texto elucidativo da situação de ocorrência de erro

Figura 3.3 - Funções que testam situações de erro.

A função **clearerr** limpa os indicadores de fim de ficheiro e de erro associados com o identificador do fluxo fornecido. A função **ferror** testa o erro associado com o identificador do fluxo fornecido, ocorrido numa instrução de leitura ou de escrita anterior. A função **perror** imprime no dispositivo convencional de saída de erro, a combinação de uma mensagem fornecida pelo programador, com a mensagem de erro associada à variável global de erro **errno**, separadas pelo carácter **:**. A função **feof** testa o elemento sinalizador de fim de ficheiro da estrutura de dados **FILE**, correspondente ao identificador do fluxo fornecido, e devolve 0 (zero), se a sinalização não foi activada, ou seja, se o fim de ficheiro não foi atingido, ou um valor diferente de 0, em caso contrário. No entanto, esta função só detecta o carácter de *fim de ficheiro* após uma tentativa de leitura falhada, pelo que, em determinadas situações produz resultados incorrectos, nomeadamente quando se tenta processar um ficheiro vazio.

A Figura 3.4 apresenta um exemplo de abertura de um fluxo de comunicação para leitura de um ficheiro de texto. Começa-se por declarar uma variável de tipo ponteiro para **FILE** e uma cadeia de caracteres para o nome do ficheiro. Deve-se assegurar que se lê de facto um nome válido para nome do ficheiro, para evitar situações anormais na tentativa de abertura do ficheiro. Quando se abre um ficheiro, seja para leitura, seja para escrita deve-se verificar sempre se a abertura ou a criação do ficheiro foi bem sucedida e caso contrário imprimir uma mensagem de erro. A mensagem de erro deve ser enviada para o dispositivo convencional de saída de erro, uma vez que, o dispositivo convencional de saída pode estar redireccionado para um ficheiro e então perder-se-ia a mensagem de erro. Em caso de erro o programa deve ser terminado com indicação de erro, o que pode ser feito através da função **exit**. Após a leitura e processamento do conteúdo do ficheiro este deve ser fechado, usando a função **fclose** que se apresenta a seguir.

```
...
FILE *FPENT;  char NOMEFICH[81];  int ST;
...
do
{
   printf ("Nome do ficheiro de entrada ");
   ST = scanf ("%80s", NOMEFICH);    /* leitura do nome do ficheiro */
   scanf ("%*[^\n]");       /* descartar todos os outros caracteres */
   scanf ("%*c");           /* descartar o carácter de fim de linha */
} while (ST == 0);
                                              /* abertura do ficheiro */
if ( (FPENT = fopen (NOMEFICH, "r") ) == NULL )
{
   fprintf (stderr, "O Ficheiro %s não existe\n", NOMEFICH);
   exit (EXIT_FAILURE);
}
...          /* leitura do conteúdo do ficheiro e seu processamento */
fclose (FPENT);                                 /* fecho do ficheiro */
...
```

Figura 3.4 - Exemplo da abertura de um ficheiro de texto para leitura.

3.3 Fecho de um fluxo de comunicação

Após a comunicação ter tido lugar, é necessário dissociar-se o fluxo de comunicação do dispositivo, para se garantir o encerramento da comunicação. Isso é feito invocando a função **fclose**, que se apresenta na Figura 3.5, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

int fclose **(** FILE ***fluxo** **) ;**

fluxo ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen** ou **freopen**, ***stdin***, ***stdout*** ou ***stderr***

Figura 3.5 - Função **fclose**.

A função **fclose** dissocia o fluxo de comunicação do dispositivo, encerra a comunicação e fecha o dispositivo. Se se tratar de um fluxo de comunicação, cujo modo de acesso de abertura pressupunha escrita, é garantido que o sistema operativo transfere para o dispositivo, toda a informação nova existente no armazenamento tampão associado ao fluxo de comunicação. A função devolve 0 (zero), quando bem sucedida, e **EOF**, em caso contrário. No caso de insucesso, a causa é sinalizada na variável global de erro **errno**.

3.4 Fluxos de texto

Os fluxos de texto estão organizados em linhas, cada linha contendo um número variável de caracteres com representação gráfica, ou com funções de controlo. Portanto, mesmo quando o dispositivo associado ao fluxo de comunicação supõe armazenamento interno de informação, só é possível escrever-se no fim da informação lá existente e ler-se a partir do princípio da informação lá existente. Os fluxos de texto são, por isso, quase sempre abertos ou criados nos modos de leitura ou escrita apenas, com os modos de acesso r, w ou a, e supõem um acesso puramente sequencial.

3.4.1 Leitura de um fluxo de texto

A operação básica para leitura de fluxos de texto é a função **fscanf**, cuja definição formal se apresenta na Figura 3.6, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

int fscanf **(** *identificador do fluxo de entrada* **,** *formato de leitura* **,** *lista de ponteiros de variáveis* **) ;**

identificador do fluxo de entrada ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen**, nos modos de acesso **r**, **r+**, **w+** ou **a+**, ou ***stdin***

Figura 3.6 - Definição formal da função **fscanf**.

A função **fscanf** lê do fluxo de texto, cujo nome é indicado pelo identificador do fluxo de entrada, sequências de caracteres e processa-as segundo as regras impostas pelo formato de leitura, armazenando sucessivamente os valores convertidos nas variáveis, cuja localização é indicada na lista de ponteiros de variáveis. As definições de formato de leitura e de lista de ponteiros de variáveis são as mesmas que para a função **scanf**.

Salvo em duas situações especiais, deve existir uma relação de um para um entre cada especificador de conversão e cada variável da lista de ponteiros de variáveis. Se o número de variáveis da lista de ponteiros de variáveis for insuficiente, o resultado da operação não está definido. Se, pelo contrário, o número de variáveis for demasiado grande, as variáveis em excesso não são afectadas. O tipo da variável e o especificador de conversão devem ser compatíveis, já que a finalidade deste último é indicar, em cada caso, que tipos de sequências de caracteres são admissíveis e como devem ser tratadas. Quando o especificador de conversão não é válido, o resultado da operação não está definido. O processo de leitura só termina quando o formato de leitura se esgota, quando é lido o carácter de *fim de ficheiro*, ou quando existe um conflito de tipo entre o que está indicado no formato de leitura e a correspondente quantidade a ser lida. Neste último caso, o carácter que causou o conflito é mantido no fluxo de texto. A função devolve o número de valores lidos e armazenados, ou o valor **EOF** (*end of file*), se o carácter de *fim de ficheiro* é lido antes que qualquer conversão tenha lugar. Se, entretanto, ocorreu um conflito, o valor devolvido corresponde ao número de valores lidos e armazenados até à ocorrência do conflito.

O maior problema na leitura de um ficheiro é a correcta detecção do carácter de *fim de ficheiro*. Na linguagem Pascal a detecção do carácter de *fim de ficheiro* é feito em avanço quando se lê o último carácter útil do ficheiro, daí que se pode ler um ficheiro com um ciclo repetitivo **repeat until**. Como na linguagem C o carácter de *fim de ficheiro* só é detectado após uma tentativa frustrada de leitura, então o ciclo de leitura não pode ser do tipo **do while**, tem que ser do tipo **while**. Por outro lado, a forma mais eficaz de o detectar, consiste em aproveitar o facto da função **fscanf** devolver o valor **EOF** quando o carácter de *fim de ficheiro* é lido antes que qualquer conversão tenha lugar.

Vamos apresentar um exemplo de leitura de um ficheiro de texto, que se supõe ter um formato bem definido, composto por linhas com NITEMS de informação com formatos bem definidos. O ficheiro deve obviamente ser lido até que seja atingido o fim do ficheiro, pelo que, é importante a sua correcta detecção. O processo de leitura também deve prever a situação de que uma linha possa eventualmente não estar de acordo com o formato esperado. O que fazer nesta situação? Temos duas soluções que vamos apresentar de seguida. Vamos considerar que o ficheiro foi aberto com sucesso e que FPENT é o ponteiro para FILE identificador do fluxo de texto.

A Figura 3.7 apresenta a solução mais simples, que consiste em terminar a leitura assim que se detecte uma linha desformatada, ou seja, uma linha que não contenha os NITEMS que era esperado ler. Nesse caso é assinalada a situação de erro e fecha-se o ficheiro.

```
#define NITEMS 4
...
FILE *FPENT;  int NLIDOS;
...
     /* leitura da linha completa de acordo com o formato esperado */
while ((NLIDOS = fscanf (FPENT, "formato\n", ponteiros)) == NITEMS)
{
  ... ;                          /* processamento da informação lida */
}
                      /* verificação da causa de paragem da leitura */
if ( NLIDOS != EOF )                  /* EOF ou linha desformatada? */
  fprintf (stderr, "Leitura parada devido a linha desformatada\n");
...
fclose (FPENT);                                  /* fecho do ficheiro */
...
```

Figura 3.7 - Leitura de um ficheiro de texto até à detecção de uma linha desformatada.

A Figura 3.8 apresenta a segunda solução, que consiste em assinalar com uma mensagem de erro sempre que se lê uma linha desformatada, e tenta passar por cima dela de forma a continuar a ler, se possível, o ficheiro até ao fim.

```
#define NITEMS 4
...
FILE *FPENT;  int NLIDOS;
...
     /* leitura da linha completa de acordo com o formato esperado */
while ( (NLIDOS = fscanf (FPENT, "formato\n", ponteiros)) != EOF )
{
   /* não ocorreu EOF mas é preciso aferir a consistência da linha */
  if ( NLIDOS != NITEMS )      /* lido o número esperado de items? */
    fprintf (stderr, "Linha desformatada\n");
  else ... ;                     /* processamento da informação lida */
  ...
}
...
fclose (FPENT);                                  /* fecho do ficheiro */
...
```

Figura 3.8 - Leitura de um ficheiro de texto ignorando linhas desformatadas.

A Figura 3.9 apresenta a solução normalmente apresentada em livros de programação que recorre à função **feof**, e que em determinadas situações produz resultados incorrectos, nomeadamente quando se tenta processar um ficheiro vazio. Pelo que, esta solução nunca deve ser usada.

```
#define NITEMS 4
...
FILE *FPENT;  int NLIDOS;
...
while ( !feof(FPENT) )
{
     /* leitura da linha completa de acordo com o formato esperado */
  NLIDOS = fscanf (FPENT, "formato\n", ponteiros)
   /* não ocorreu EOF mas é preciso aferir a consistência da linha */
  if ( NLIDOS != NITEMS )      /* lido o número esperado de items? */
    fprintf (stderr, "Linha desformatada\n");
  else ... ;                   /* processamento da informação lida */
}
...
fclose (FPENT);                                  /* fecho do ficheiro */
...
```

Figura 3.9 - Leitura de um ficheiro de texto recorrendo à função **feof**.

3.4.2 Escrita num fluxo de texto

A operação básica para escrita em fluxos de texto é a função **fprintf**, cuja definição formal se apresenta na Figura 3.10, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

int fprintf **(** *identificador do fluxo de saída* **,** *formato de escrita* **,** *lista de expressões* **) ;**

identificador do fluxo de saída ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen**, nos modos de acesso **r+**, **w**, **w+**, **a** ou **a+**, ou ***stdout*** ou ***stderr***

Figura 3.10 - Definição formal da função **fprintf**.

A função **fprintf** escreve sucessivamente no fluxo de texto, cujo nome é indicado pelo identificador do fluxo de saída, sequências de caracteres, representativas de texto e dos valores das expressões que formam a lista de expressões, segundo as regras impostas pelo formato de escrita. As definições de formato de escrita e de lista de expressões são as mesmas que para a função **printf**.

O texto é especificado no formato de escrita pela introdução de literais, que são copiados sem modificação para o fluxo de saída. O modo como o valor das expressões é convertido, é descrito pelos especificadores de conversão. Salvo numa situação especial, deve existir uma relação de um para um entre cada especificador de conversão e cada expressão da lista de expressões. Se o número de expressões for insuficiente, o resultado da operação não está definido. Se, pelo contrário, esse número for demasiado grande, as expressões em excesso são ignoradas.

O tipo da expressão e o especificador de conversão devem ser compatíveis, já que a finalidade deste último é indicar, em cada caso, o formato da sequência convertida. Quando o especificador de conversão não é válido, o resultado da operação não está definido. O processo de escrita só termina quando o formato de escrita se esgota, ou quando ocorre um erro. A função devolve o número de caracteres escritos no fluxo de saída, ou o valor −1, se ocorreu um erro.

A Figura 3.11 apresenta um exemplo em que vamos ler um ficheiro de texto formatado e escrever a informação lida noutro ficheiro de texto formatado, eventualmente com um formato diferente. Vamos considerar que o ficheiro de entrada foi aberto com sucesso, sendo FPENT o ponteiro para FILE identificador do fluxo de texto de entrada, e que o ficheiro para escrita foi aberto com sucesso, sendo FPSAI o ponteiro para FILE identificador do fluxo de texto de saída. Só as linhas que estão de acordo com o formato de leitura esperado serão escritas no ficheiro de saída com o formato pretendido.

```
#define NITEMS 4
...
FILE *FPENT, *FPSAI;  int NLIDOS;
...
     /* leitura da linha completa de acordo com o formato esperado */
while ( (NLIDOS = fscanf (FPENT, "formato\n", ponteiros)) != EOF )
{
   /* não ocorreu EOF mas é preciso aferir a consistência da linha */
  if ( NLIDOS != NITEMS )      /* lido o número esperado de items? */
    fprintf (stderr, "Linha desformatada\n");

  else
  {
    ... ;                      /* processamento da informação lida */
   /* escrita da linha completa de acordo com o formato pretendido */
    fprintf (FPSAI, "formato\n", expressões);
  }
  ...
}
...
fclose (FPENT);                    /* fecho do ficheiro de entrada */
fclose (FPSAI);                      /* fecho do ficheiro de saída */
...
```

Figura 3.11 - Escrita num ficheiro de texto.

3.4.3 Considerações finais sobre fluxos de texto

As instruções seguintes de leitura do fluxo de texto associado com o dispositivo convencional de entrada são equivalentes.

int fscanf **(stdin ,** *formato de leitura* **,** *lista de ponteiros de variáveis* **) ;**
int scanf **(** *formato de leitura* **,** *lista de ponteiros de variáveis* **) ;**

As instruções seguintes de escrita no fluxo de texto associado com o dispositivo convencional de saída são equivalentes.

int fprintf **(stdout ,** *formato de escrita* **,** *lista de expressões* **) ;**
int printf **(** *formato de escrita* **,** *lista de expressões* **) ;**

As instruções seguintes de escrita no fluxo de texto associado com o dispositivo convencional de saída de erro são aproximadamente equivalentes.

int fprintf **(stderr ,** "*mensagem definida pelo programador* **:**%s\n" **,** strerror (**errno**) **) ;**
void perror **(** *mensagem definida pelo programador* **) ;**

Existem ainda as duas funções, que se apresentam na Figura 3.12, para leitura e escrita de linhas de texto, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

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

Figura 3.12 - Funções para leitura e escrita de linhas de texto.

A função **fgets** lê caracteres do fluxo de texto especificado por **fluxo**, para a cadeia de caracteres referenciada por **s**. O fluxo de texto foi inicializado pela invocação prévia de **fopen**, nos modos de acesso que permitam a leitura. Os caracteres são lidos até que seja detectado o carácter de *fim de linha*, ou o carácter de *fim de ficheiro*, ou até terem sido lidos **n−1** caracteres. A função adiciona automaticamente o carácter nulo final. Se bem sucedida, a função devolve um ponteiro para a cadeia de caracteres **s**. Se foi encontrado o carácter de *fim de ficheiro* antes da leitura de qualquer carácter, a cadeia de caracteres **s** fica inalterada e é devolvido o ponteiro nulo. Se, entretanto, ocorreu um erro, é devolvido o ponteiro nulo, mas o conteúdo da cadeia de caracteres é imprevisível.

A função **fputs** escreve a cadeia de caracteres referenciada por **s**, no fluxo de texto especificado por **fluxo**. A cadeia de caracteres tem de ser obrigatoriamente terminada com o carácter nulo final, carácter esse que não é escrito no fluxo de texto. É preciso ter em atenção que, a função não escreve o carácter de *fim de linha*. Se bem sucedida, a função devolve o valor 0 (zero), ou um valor diferente de 0, em caso contrário.

3.5 Passagem de argumentos na linha de comando

Quando se coloca em execução um programa desenvolvido na linguagem C, a função **main**, que se comporta como o programa principal, é a primeira função a ser invocada e a partir da qual todas as restantes funções são invocadas. É assim possível, e desejável em certas situações, passar-lhe informação de entrada, tal como para qualquer outra função. A responsabilidade de lhe passar essa informação é do sistema operativo através da linha de comando, o que se designa por **passagem de argumentos na linha de comando**.

Este mecanismo permite à função **main** comunicar directamente com o exterior e receber os argumentos que foram escritos na linha de comando. Para tal, acrescenta-se ao cabeçalho da função, dois parâmetros de entrada com a seguinte sintaxe.

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

- O primeiro parâmetro chama-se normalmente **argc**, é de tipo ***int*** e indica o número de argumentos que foram escritos na linha de comando, número esse que inclui o nome do programa invocado.

- O segundo parâmetro chama-se normalmente **argv**, é de tipo ***char * [ ]***, ou ***char ****, ou seja, é um agregado de cadeias de caracteres. Cada cadeia de caracteres armazena um argumento da linha de comando, sendo que o primeiro argumento de índice zero é o nome do programa invocado.

Este mecanismo de comunicação é muito útil, principalmente quando queremos desenvolver programas que vão manipular ficheiros, porque simplifica o código necessário à obtenção dos nomes dos ficheiros. Em vez do programa ter de ler o nome de cada ficheiro, necessitando para tal de uma variável para o armazenar e de assegurar que não lê uma cadeia de caracteres nula, pode-se passar a informação ao invocar o programa. Mas, a utilização deste mecanismo implica alguns cuidados. Nomeadamente é preciso assegurar

que todos os argumentos necessários à execução do programa foram de facto passados na linha de comando.

Vamos considerar que queremos escrever um programa que utiliza dois ficheiros. Neste caso o programa tem de receber pelo menos três argumentos, que são, o nome do programa e os nomes dos dois ficheiros. A Figura 3.13 apresenta o excerto de código que valida os argumentos de entrada. Se o número de argumentos for inferior a três, o programa escreve no dispositivo convencional de saída de erro, identificado por **stderr**, como é que o programa deve ser invocado e termina a execução do programa com indicação de finalização sem sucesso.

```
...
int main (int argc, char *argv[])
{
  if ( argc < 3 )              /* número de argumentos suficiente? */
  {
    fprintf (stderr, "Uso: %s ficheiro ficheiro\n", argv[0]);
    exit (EXIT_FAILURE);
  }
  ...                                  /* processamento dos ficheiros */
  return EXIT_SUCESS;
}
```

Figura 3.13 - Utilização da passagem de argumentos na linha de comando.

3.6 Exemplos de processamento de ficheiros de texto

Pretende-se escrever um programa que leia um ficheiro de texto e que escreva noutro ficheiro de texto, todo o texto alfabético em minúsculas. Ou seja, que codifique todos os caracteres alfabéticos maiúsculos do ficheiro de entrada em caracteres alfabéticos minúsculos no ficheiro de saída. Apesar de um ficheiro de texto estar organizado como um conjunto de linhas, o seu processamento pode ser feito como uma sequência de caracteres, uma vez que o carácter de *fim de linha* pode ser tratado como um carácter normal. O que torna a organização do ficheiro em linhas transparente para o programador.

A Figura 3.14 apresenta o programa. Temos dois ficheiros de texto, logo necessitamos de dois ponteiros para FILE. Vamos designar o ponteiro para o ficheiro de entrada por FPENT e o ponteiro para o ficheiro de saída por FPSAI. Os nomes dos ficheiros vão ser passados na linha de comando. Vamos considerar que o primeiro argumento **argv[1]**, é o nome do ficheiro de entrada e o segundo argumento **argv[2]**, é o nome do ficheiro de saída. O número de argumentos **argc**, bem como, a abertura dos ficheiros é validada e em qualquer situação de erro o programa termina com indicação de finalização sem sucesso. No caso da abertura sem sucesso do ficheiro de saída, o programa deve fechar o ficheiro de entrada, que entretanto foi aberto, antes de terminar. A leitura do ficheiro de entrada é feita carácter a carácter até ser atingido o fim do ficheiro. Cada carácter é escrito no ficheiro de saída, transformado para minúsculo através da função **tolower**. Como a função só converte o carácter se ele for um carácter alfabético maiúsculo, então não há a necessidade de testar previamente o carácter. Se o carácter for o carácter de *fim de linha*, o '\n', ele é escrito no ficheiro de saída provocando uma mudança de linha. Assim transporta-se o formato do ficheiro de entrada para o ficheiro de saída, com excepção dos caracteres alfabéticos maiúsculos que passam para caracteres alfabéticos minúsculos. O carácter '\' é o carácter continuador de linha, que se utiliza para decompor linhas compridas.

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

int main (int argc, char *argv[])
{
  FILE *FPENT, *FPSAI;  char CAR;

  if ( argc < 3 )          /* o número de argumentos é suficiente? */
  {
    fprintf (stderr, "Uso: %s ficheiro ficheiro\n", argv[0]);
    exit (EXIT_FAILURE);
  }

           /* abertura do ficheiro de entrada cujo nome é argv[1] */
  if ( (FPENT = fopen (argv[1], "r") ) == NULL )
  {
    fprintf (stderr, "Não foi possível abrir o ficheiro %s\n"\
                  , argv[1]);
    exit (EXIT_FAILURE);
  }

              /* criação do ficheiro de saída cujo nome é argv[2] */
  if ( (FPSAI = fopen (argv[2], "w") ) == NULL )
  {
    fprintf (stderr, "Não foi possível criar o ficheiro %s\n"\
                  , argv[2]);
    fclose (FPENT);               /* fecho do ficheiro de entrada */
    exit (EXIT_FAILURE);
  }

       /* leitura do carácter do ficheiro de entrada e escrita do */
       /* carácter convertido para minúsculo no ficheiro de saída */
  while ( fscanf (FPENT, "%c", &CAR) != EOF )
    fprintf (FPSAI, "%c", tolower(CAR));

  fclose (FPENT);                 /* fecho do ficheiro de entrada */
  fclose (FPSAI);                   /* fecho do ficheiro de saída */

  return EXIT_SUCCESS;
}
```

Figura 3.14 - Programa de codificação do ficheiro.

Pretende-se escrever um programa que lê um ficheiro de texto que armazena informação relativa a um conjunto de pessoas e que determina as pessoas que nasceram no dia 29 de Fevereiro, escrevendo no monitor o nome da pessoa e o seu número de telefone. O ficheiro é constituído por linhas, sendo que cada linha contém a informação relativa a uma pessoa, com o seguinte formato. O número de registo da pessoa, que é do tipo inteiro positivo, o nome que é uma cadeia de caracteres com 40 caracteres no máximo, a data de nascimento constituída pelo dia, mês em numérico e ano, e o número de telefone que é uma cadeia de 9 caracteres. Estes elementos de informação estão separados pelo carácter **:**.

A Figura 3.15 apresenta o programa. Uma vez que o nome de uma pessoa é constituído por nomes separados pelo espaço, não se pode usar o formato de leitura %s. Tem que se utilizar o formato alternativo para a leitura de cadeias de caracteres que é o %[]. Como o número de registo da pessoa e o ano de nascimento não são precisos, a leitura pode descartar esses itens, não havendo a necessidade de declarar variáveis para esses itens. A linha deve ser lida de forma a colocar o indicador de posição do ficheiro no início da linha seguinte, pelo que, o carácter de *fim de linha* também deve ser lido. Portanto, o formato apropriado deve ser "%*****d**:**%40[**^:**]**:**%d**:**%d**:**%*****d**:**%9s**\n**". O nome do ficheiro vai ser passado na linha de comando. O número de argumentos e a abertura do ficheiro são

validados, e em qualquer situação de erro o programa termina com indicação de finalização sem sucesso. A leitura do ficheiro de entrada é feita linha a linha até ser atingido o fim do ficheiro e se for detectada uma linha desformatada, o programa ignora essa linha e continua a leitura do ficheiro.

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

#define NITEMS 4

int main (int argc, char *argv[])
{
  FILE *FPENT;  char NOME[41], TEL[10];  int NLIDOS, DIA, MES;

  if ( argc < 2 )          /* o número de argumentos é suficiente? */
  {
    fprintf (stderr, "Uso: %s nome do ficheiro\n", argv[0]);
    exit (EXIT_FAILURE);
  }

           /* abertura do ficheiro de entrada cujo nome é argv[1] */
  if ( (FPENT = fopen (argv[1], "r") ) == NULL )
  {
    fprintf (stderr, "Não foi possível abrir o ficheiro %s\n"\
                  , argv[1]);
    exit (EXIT_FAILURE);
  }

  printf ("Aniversariantes a 29 de Fevereiro\n");

                       /* leitura da linha do ficheiro de entrada */
  while ( (NLIDOS = fscanf (FPENT, "%*d:%40[^:]:%d:%d:%*d:%9s\n"\
                                 , NOME, &DIA, &MES, TEL)) != EOF )
    if ( NLIDOS != NITEMS )    /* lido o número esperado de items? */
      fprintf (stderr, "Linha desformatada\n");
                /* escrita no monitor dos aniversariantes a 29/Fev */
    else if ( DIA == 29 && MES == 2)
           printf ("NOME -> %-40.40s TELEFONE -> %s\n", NOME, TEL);

  fclose (FPENT);                              /* fecho do ficheiro */
  return EXIT_SUCCESS;
}
```

Figura 3.15 - Processamento de um ficheiro constituído por linhas com um formato específico.

Como exercício de treino, altere o programa para escrever a informação de saída num ficheiro de texto, cujo nome é passado na linha de comando.

3.7 Fluxos binários

Nos fluxos binários, a transferência de informação processa-se *byte* a *byte* segundo o formato usado para representação dos valores em memória. Quando o dispositivo associado ao fluxo de comunicação supõe armazenamento interno da informação, é comum ter-se uma organização em duas partes, o cabeçalho e o corpo. O **cabeçalho** contém informação genérica relativa ao fluxo como um todo. O **corpo** é concebido como uma sequência de registos de um tipo previamente definido. A leitura e a escrita do cabeçalho ou de um registo particular podem ocorrer em qualquer ponto da sequência de operações, já que é possível determinar-se com rigor a sua localização no ficheiro. Os fluxos binários são, por isso, muitas vezes abertos ou criados nos modos de leitura ou escrita mistos, com os modos de acesso rb+, wb+ ou ab+, e supõem um acesso aleatório.

3.7.1 Leitura e escrita de fluxos binários

As operações básicas para leitura e escrita em fluxos binários são respectivamente as funções **fread** e **fwrite**, cuja definição formal se apresenta na Figura 3.16, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

int fread **(** *localização* **,** *tamanho do elemento* **,** *número de elementos* **,** *identificador do fluxo* **) ;**

int fwrite **(** *localização* **,** *tamanho do elemento* **,** *número de elementos* **,** *identificador do fluxo* **) ;**

localização ::= variável de tipo ponteiro para o tipo elemento que localiza o início da região de armazenamento em memória

tamanho do elemento ::= tamanho em *bytes* de uma variável de tipo elemento

número de elementos ::= número de elementos a serem lidos ou escritos

identificador do fluxo ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen** ou **freopen**

Figura 3.16 - Definição formal das funções **fread** e **fwrite**.

A função **fread** lê do fluxo binário, cujo nome é indicado pelo identificador do fluxo, o número de elementos indicado, todos com o mesmo tamanho em *bytes*, e armazena-os na região de memória referenciada pela localização indicada. A região de memória referenciada tem que ter capacidade para o armazenamento da informação solicitada. O processo de leitura termina quando é lido o número total de *bytes* pretendido, ou quando é lido o carácter de *fim de ficheiro*, ou quando ocorreu um erro. A função devolve o número de elementos que foram efectivamente lidos e armazenados, que pode ser menor do que o número de elementos pretendidos, se o carácter de *fim de ficheiro* é lido antes, ou se ocorreu um erro. Se o número de elementos lidos for zero, então o valor devolvido é 0 (zero) e, quer a região de memória referenciada, quer o fluxo binário não são afectados.

A função **fwrite** escreve no fluxo binário, cujo nome é indicado pelo identificador do fluxo, o número de elementos indicado, todos com o mesmo tamanho em *bytes*, que estão armazenados na região de memória referenciada pela localização indicada. O processo de escrita termina quando é escrito o número total de *bytes* pretendido, ou quando ocorre um erro. A função devolve o número de elementos que foram efectivamente escritos no fluxo de saída, que pode ser menor do que o número de elementos pretendidos, se ocorreu um erro. Se o número de elementos escritos for zero, o valor devolvido é 0 (zero) e o fluxo binário não é afectado.

A Figura 3.17 exemplifica a utilização das funções **fread** e **fwrite** para ler e escrever uma estrutura de tipo TDADOS_PESSOA em ficheiros binários.

```
TDADOS_PESSOA  PESSOA;
...                                     /* leitura de uma estrutura */
fread (&PESSOA, sizeof (TDADOS_PESSOA), 1, FPENT);
...                                     /* escrita de uma estrutura */
fwrite (&PESSOA, sizeof (TDADOS_PESSOA), 1, FPOUT);
```

Figura 3.17 - Utilização das funções **fread** e **fwrite**.

3.8 Funções para colocação do indicador de posição

Quando o dispositivo associado ao fluxo de comunicação supõe o armazenamento interno de informação, ou seja, estamos perante um ficheiro, o ponto onde se efectua a próxima operação de leitura ou de escrita pode ser modificado através da manipulação do indicador de posição do ficheiro. As operações básicas para controlo e monitorização do indicador de posição do ficheiro são as funções **fseek**, **ftell** e **rewind**, que se apresentam na Figura 3.18, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

int fseek **(** FILE ***fluxo**, **long** deslocamento**, int** referência **) ;**

long ftell **(** FILE ***fluxo **) ;**

void rewind **(** FILE ***fluxo **) ;**

fluxo ::= variável de tipo ponteiro para **FILE**, inicializada pela invocação prévia de **fopen** ou **freopen**

deslocamento ::= número de *bytes* a deslocar o indicador de posição do ficheiro

referência ::= SEEK_SET | SEEK_CUR | SEEK_END

Figura 3.18 - Funções para colocação do indicador de posição do ficheiro.

A função **fseek** posiciona o indicador de posição do ficheiro indicado, no *byte* referenciado pelo deslocamento pretendido a partir do ponto de referência. O ponto de referência pode ser o princípio da informação armazenada, reconhecido pelo identificador SEEK_SET, a posição actual do indicador, reconhecido pelo identificador SEEK_CUR ou o fim da informação armazenada, reconhecido pelo identificador SEEK_END. A função devolve 0 (zero), em caso de sucesso, e −1, em caso de erro. No caso de insucesso, a causa é sinalizada na variável global de erro **errno**.

Embora a sua aplicação possa ser feita com qualquer tipo de fluxo de comunicação, o seu uso com fluxos de texto é muito restrito. A norma ANSI impõe que, neste caso, o ponto de referência é obrigatoriamente SEEK_SET e o deslocamento ou é 0 (zero), ou o valor devolvido por uma invocação prévia da função **ftell**.

A função **ftell** devolve o valor do indicador de posição do ficheiro indicado. Para os fluxos binários, o valor devolvido representa a distância em *bytes* a partir do princípio da informação armazenada. Para os fluxos de texto, o valor devolvido contém um valor, que depende da implementação e que pode ser usado posteriormente por uma invocação de **fseek** para posicionamento do indicador de posição do ficheiro na mesma posição. A função devolve o valor do indicador de posição do ficheiro, em caso de sucesso, e o valor −1L, em caso de erro. No caso de insucesso, a causa é sinalizada na variável global de erro **errno**.

A função **rewind** posiciona o indicador de posição do ficheiro no início da informação armazenada. A função é equivalente a invocar a função **fseek** para o início do ficheiro, mas, com a vantagem de limpar os indicadores de erro e de fim de ficheiro, e sem devolver qualquer valor.

3.9 Esvaziamento do armazenamento tampão

Por vezes é necessário garantir que a última informação produzida e colocada no armazenamento tampão associado ao fluxo de comunicação é enviada para o dispositivo. Isso é feito invocando a função **fflush**, que se apresenta na Figura 3.19, e cuja descrição está contida no ficheiro de interface ***stdio.h***.

```
int fflush ( FILE *fluxo ) ;
```

Figura 3.19 - Função **fflush**.

A função **fflush** permite o esvaziamento do armazenamento tampão associado ao fluxo de comunicação indicado, transferindo toda a informação nova aí existente para o dispositivo. Se bem sucedida, a função devolve o valor 0 (zero), ou um valor diferente de 0, em caso contrário.

3.10 Funções para operar sobre ficheiros

A Figura 3.20 apresenta as funções que a biblioteca ***stdio*** fornece para operar directamente sobre ficheiros.

```
int remove ( const char *nome do ficheiro ) ;
int rename ( const char *nome antigo, const char *nome novo ) ;
FILE *tmpfile ( void ) ;
char *tmpnam ( char *nome do ficheiro ) ;
```

Figura 3.20 - Funções para operar sobre ficheiros.

A função **remove** remove do sistema de ficheiros o ficheiro cujo nome é indicado por nome do ficheiro. A função devolve 0 (zero), se tiver sucesso, e um valor não nulo, em caso contrário.

A função **rename** altera o nome do ficheiro cujo nome é indicado por nome antigo para o nome novo. A função devolve 0 (zero), se tiver sucesso, e um valor não nulo, em caso contrário.

A função **tmpfile** cria um ficheiro binário temporário que é automaticamente apagado quando é fechado, ou quando o programa termina. O ficheiro é criado no modo de acesso wb+.

A função **tmpnam** gera um nome válido de ficheiro que é distinto de qualquer nome já existente. Até TMP_MAX vezes, no mínimo 25, é garantido que os nomes sucessivamente gerados são diferentes. Se o nome do ficheiro for um ponteiro nulo, a função devolve a localização de uma região de memória interna onde está armazenado o nome, caso contrário, deverá apontar para uma região de memória com capacidade de armazenamento de L_tmpnam caracteres, e será esse o valor devolvido. O valor de L_tmpnam está definido no ficheiro de interface ***limits.h***.

3.11 Exemplo da manutenção de uma base de dados

Pretende-se desenvolver um programa que crie e mantenha uma agenda telefónica. A agenda deve ser concebida como uma sequência de registos que contenham a seguinte informação. O nome de uma pessoa, com 40 caracteres no máximo, o seu número de telefone, com 15 caracteres no máximo e a sua data de aniversário, constituída pelo dia, mês e ano. O programa deve funcionar de forma repetitiva, apresentando um menu que permita as seguintes opções: a introdução de um registo novo na agenda; a eliminação de um registo preexistente da agenda, sendo que o registo a eliminar é identificado pelo número de telefone; a listagem do conteúdo de todos os registos da agenda, por ordem alfabética do nome; e terminar a execução do programa. A Figura 3.21 apresenta o algoritmo em pseudocódigo e linguagem natural da agenda telefónica.

```
nome: Base de dados da agenda telefónica manuseada num ficheiro
begin
  Abrir ou criar o ficheiro da agenda telefónica;
  do
     Escrita do menu de operações disponíveis e leitura da operação;
     Processamento da operação escolhida;
  while  não for seleccionada a operação para terminar;
  Fechar o ficheiro da agenda telefónica;
end
```

Figura 3.21 - Algoritmo da agenda telefónica manuseada num ficheiro.

Uma vez que se pretende efectuar a listagem do ficheiro por ordem alfabética do nome, então temos que manter a agenda telefónica sempre ordenada inserindo cada novo registo no sítio certo. O que implica pesquisar o ficheiro à procura do local de inserção do novo registo e depois deslocar todos os registos que se encontram abaixo do local de inserção, um registo para baixo, de forma a abrir espaço para escrever o novo registo. A Figura 3.22 apresenta a estrutura do ficheiro binário, que é constituído por um cabeçalho que indica o número de registos presentes no ficheiro e o corpo que é um conjunto de registos idênticos que estão ordenados por ordem alfabética do nome, para facilitar a listagem do ficheiro.

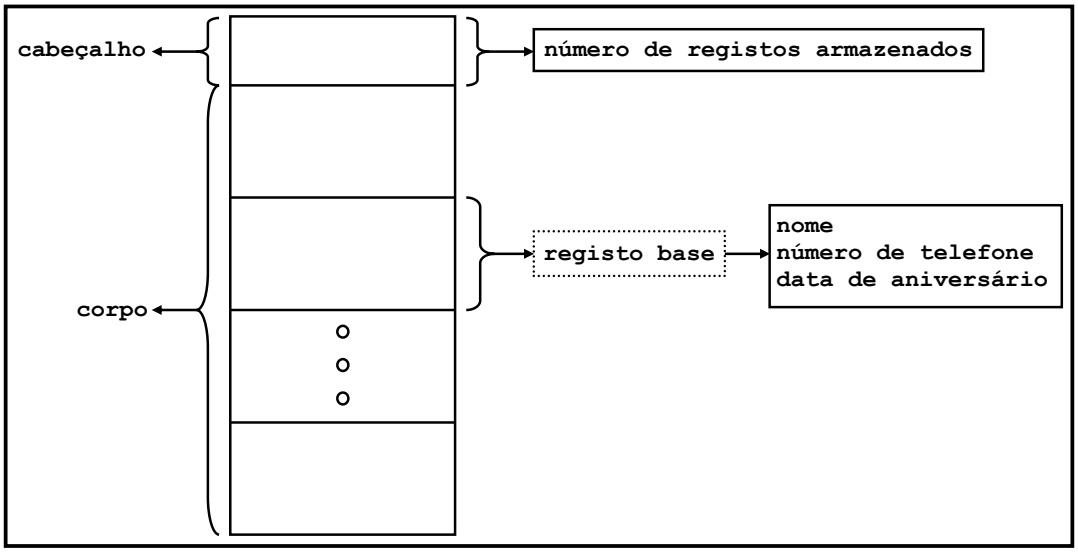

Figura 3.22 - Formato do ficheiro da agenda telefónica.

A Figura 3.23 apresenta o programa da agenda telefónica, ou seja, a função main. Para a sua implementação o programa tem como variáveis de entrada, o nome do ficheiro e a opção pretendida pelo utilizador. Como o ficheiro binário vai ser utilizado como estrutura de dados de suporte da agenda telefónica, então o ficheiro, ou melhor o fluxo binário associado com o ficheiro, é uma variável interna do programa.

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

typedef struct                              /* definição do tipo TDATA */
{
  unsigned int DIA, MES, ANO;
} TDATA;

#define  NMX  41         /* número máximo de caracteres do nome + 1 */
#define  TMX  16     /* número máximo de caracteres do telefone + 1 */

typedef struct                               /* definição do tipo TREG */
{
  char NOME[NMX];                                            /* nome */
  char NTELEF[TMX];                          /* número de telefone */
  TDATA ANIVERSARIO;                        /* data de aniversário */
} TREG;
                                             /* alusão a funções */
void ESCREVER_MENU_E_LER_OPCAO (int *);
void LER_NOME_FICHEIRO (char []);
void ABRIR_CRIAR_FICHEIRO (char [], FILE **);
void FECHAR_FICHEIRO (FILE *);
void INSERIR_REGISTO (FILE *);
void ELIMINAR_REGISTO (FILE *);
void LISTAR_FICHEIRO (FILE *);

int main (void)
{
  char NOMEFICH[NMX];                          /* nome do ficheiro */
  int OPCAO;                                   /* escolha da opção */
  FILE *FP;                        /* ponteiro para o fluxo binário */

  LER_NOME_FICHEIRO (NOMEFICH);           /* ler o nome do ficheiro */
  ABRIR_CRIAR_FICHEIRO (NOMEFICH, &FP);  /* abrir/criar o ficheiro */
  do                                                /* processamento */
  {
                        /* apresentação do menu e escolha da opção */
    ESCREVER_MENU_E_LER_OPCAO (&OPCAO);
    switch (OPCAO)                          /* realização da operação */
    {
      case 1: INSERIR_REGISTO (FP); break;
      case 2: ELIMINAR_REGISTO (FP); break;
      case 3: LISTAR_FICHEIRO (FP);
    }
  } while (OPCAO != 4);
  FECHAR_FICHEIRO (FP);                       /* fechar o ficheiro */
  return EXIT_SUCCESS;
}
```

Figura 3.23 - Programa da agenda telefónica.

Vamos agora apresentar as funções necessárias para a implementação do programa. A função ESCREVER_MENU_E_LER_OPCAO, que se apresenta na Figura 3.24, como o nome indica escreve o menu de operações e lê a escolha do utilizador, que é o parâmetro de saída da função. A validação destina-se a assegurar que a opção escolhida é uma das opções permitidas, pelo que, o programa principal não tem que se proteger contra dados de entrada incorrectos.

```
void ESCREVER_MENU_E_LER_OPCAO (int *OP)
{
   do
   {
      fprintf (stdout, "\nAgenda de datas de aniversário\n\n");
      fprintf (stdout, "1 - Introduzir um registo novo\n");
      fprintf (stdout, "2 - Eliminar um registo preexistente\n");
      fprintf (stdout, "3 - Listar a agenda telefónica\n");
      fprintf (stdout, "4 - Saída do programa\n");
      fprintf (stdout, "\nQual é a sua escolha? ");
      fscanf (stdin, "%1d", OP);                    /* ler a opção */
      fscanf (stdin, "%*[^\n]");     /* ler e descartar outros dados */
      fscanf (stdin, "%*c");       /* ler e descartar o fim de linha */
   } while (*OP < 1 || *OP > 4);
}
```

Figura 3.24 - Função para escrever o menu de operações e ler a opção pretendida.

A função LER_NOME_FICHEIRO, que se apresenta na Figura 3.25, é um procedimento que tem como parâmetro de saída o nome do ficheiro que é lido do teclado.

```
void LER_NOME_FICHEIRO (char NOMEF[])
{
   fprintf (stdout, "\nNome do ficheiro? ");
   fscanf (stdin, "%40s", NOMEF);           /* ler NMX-1 caracteres */
   fscanf (stdin, "%*[^\n]");  /* ler e descartar outros caracteres */
   fscanf (stdin, "%*c");         /* ler e descartar o fim de linha */
}
```

Figura 3.25 - Função para ler o nome do ficheiro.

A função ABRIR_CRIAR_FICHEIRO, que se apresenta na Figura 3.26, admite que o ficheiro, cujo nome é o parâmetro de entrada NOMEF, existe e tenta abri-lo. Caso ele não exista, pergunta ao utilizador se o pretende criar. No caso da criação de um ficheiro novo, o cabeçalho é inicializado com zero registos. O ponteiro para o fluxo associado ao ficheiro binário é um parâmetro de saída, pelo que, tem de ser passado por referência. Mas, como ele é um ponteiro para FILE, então tem de ser passado um ponteiro para o ponteiro para FILE, daí a declaração FILE **FP. Em caso de situação de erro na criação do ficheiro ou na escrita do cabeçalho são escritas mensagens de erro no dispositivo convencional de saída de erro.

A função FECHAR_FICHEIRO, que se apresenta na Figura 3.27, fecha o ficheiro, cujo ponteiro para o fluxo associado é o parâmetro de entrada FP. O fecho do ficheiro é validado e em caso de situação de erro é escrita uma mensagem de erro no dispositivo convencional de saída de erro.

A vantagem da utilização da função **perror**, na escrita de mensagens de erro, deve-se ao facto da função acrescentar à mensagem do utilizador, a mensagem associada à variável global de erro **errno**, informando-nos da causa do erro.

```
void ABRIR_CRIAR_FICHEIRO (char NOMEF[], FILE **FP)
{
  char OPC;                                    /* escolha da opção */
  unsigned int ZERO = 0;      /* sinalização do número de registos */

  if ( (*FP = fopen (NOMEF, "rb+")) != NULL ) return;

  do
  {
    fprintf (stdout, "\nO ficheiro não existe.\n"\
                     "Deseja criá-lo (s/n)? ");
    fscanf (stdin, "%c", &OPC);                    /* ler a opção */
    fscanf (stdin, "%*[^\n]");       /* ler e descartar caracteres */
    fscanf (stdin, "%*c");       /* ler e descartar o fim de linha */
  } while ( OPC != 's' && OPC != 'n');

  if (OPC == 'n') exit (EXIT_SUCCESS);

  if ( (*FP = fopen (NOMEF, "wb+")) == NULL )
  {
    perror ("erro na criação do ficheiro"); exit (EXIT_FAILURE);
  }

  if (fwrite (&ZERO, sizeof (ZERO), 1, *FP) != 1)
  {
    fprintf (stderr, "erro na inicialização do ficheiro\n");
    exit (EXIT_FAILURE);
  }
}
```

Figura 3.26 - Função para abrir ou criar o ficheiro da agenda telefónica.

```
void FECHAR_FICHEIRO (FILE *FP)
{
  if ( fclose (FP) == EOF )
  {
    perror ("erro no fecho do ficheiro"); exit (EXIT_FAILURE);
  }
}
```

Figura 3.27 - Função para fechar o ficheiro da agenda telefónica.

A Figura 3.28 apresenta o algoritmo em linguagem natural da função INSERIR_REGISTO. Após a leitura do conteúdo do novo registo digitado no teclado pelo utilizador recorrendo à função LER_REGISTO_TECLADO, é preciso determinar o seu local de inserção no ficheiro, em função do número de registos actualmente armazenados no ficheiro. A função LER_NUMREG determina o número de registos. A função POS_INSERCAO implementa a operação de pesquisa e devolve o índice do registo do ficheiro onde o novo registo deve ser colocado. O processo de comparação é rudimentar, usando para o efeito a função de comparação de cadeias de caracteres **strcmp**, e por isso, supõe-se que o nome é escrito em maiúsculas, com o número mínimo de espaços a separar as palavras. A inserção do registo no ficheiro exige deslocar para baixo os registos do ficheiro a partir do elemento que foi detectado como local de inserção do novo registo, o que é feito pela função DESLOCAR_PARA_BAIXO. A não ser que o local de inserção do registo novo seja o fim do ficheiro. Finalmente é preciso actualizar o número de registos armazenados no ficheiro, o que é feito pela função ESCREVER_NUMREG.

A Figura 3.29 apresenta a implementação da função INSERIR_REGISTO. As funções auxiliares serão posteriormente apresentadas.

```
nome: Inserir um registo novo na agenda telefónica
begin
  Leitura do conteúdo do novo registo do teclado;
  Obtenção do número de registos armazenados no ficheiro;
  Determinação do ponto de inserção do novo registo no ficheiro;
  Escrita do novo registo no ficheiro;
  Actualização do número de registos armazenados no ficheiro;
end
```

Figura 3.28 - Algoritmo da inserção de um registo novo na agenda telefónica.

```
void INSERIR_REGISTO (FILE *FP)
{
   TREG REGISTO;              /* registo novo a ser lido do teclado */
   unsigned int NREG,             /* número de registos armazenados */
                PREG;          /* ponto de inserção do registo novo */
                                                  /* alusão a funções */
   void LER_REGISTO_TECLADO (TREG *);
   void LER_NUMREG (FILE *, unsigned int *);
   void ESCREVER_NUMREG (FILE *, unsigned int);
   unsigned int POS_INSERCAO (FILE *, unsigned int, TREG);
   void DESLOCAR_PARA_BAIXO (FILE *, unsigned int, unsigned int);
   void ESCREVER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG);
                 /* leitura do conteúdo do novo registo do teclado */
   LER_REGISTO_TECLADO (&REGISTO);
         /* obtenção do número de registos actualmente armazenados */
   LER_NUMREG (FP, &NREG);
                              /* determinação do ponto de inserção */
   PREG = (NREG == 0) ? 0 : POS_INSERCAO (FP, NREG, REGISTO);
                            /* escrita do novo registo no ficheiro */
   if (PREG != NREG) DESLOCAR_PARA_BAIXO (FP, NREG, PREG);
   ESCREVER_REGISTO_FICHEIRO (FP, PREG, REGISTO);
                 /* actualização do número de registos armazenados */
   ESCREVER_NUMREG (FP, NREG+1);
}
```

Figura 3.29 - Função para inserir um registo novo na agenda telefónica.

A Figura 3.30 apresenta o algoritmo em linguagem natural da função ELIMINAR_REGISTO. Para eliminar um registo preexistente no ficheiro, em primeiro lugar temos que verificar se existem de factos registos armazenados no ficheiro, o que é feito pela função LER_NUMREG. Depois é preciso determinar se o registo pretendido existe no ficheiro e em que posição se encontra. Para pesquisar o registo, lê-se do teclado a chave de pesquisa, que é o número de telefone, com a função LER_TELEFONE. A função POS_ELIMINACAO implementa a operação de pesquisa e devolve o índice do primeiro registo do ficheiro, cujo número de telefone é igual ao número indicado. Caso não haja qualquer registo nestas condições, será devolvido o número de registos do ficheiro. A eliminação do registo no ficheiro exige deslocar para cima os registos do ficheiro a partir do elemento que foi detectado como local de eliminação do novo registo, o que é feito pela função DESLOCAR_PARA_ACIMA. A não ser que o registo seja o último registo do ficheiro. Finalmente é preciso actualizar o número de registos armazenados no ficheiro.

A Figura 3.31 apresenta a implementação da função ELIMINAR_REGISTO. As funções auxiliares serão posteriormente apresentadas.

```
nome: Eliminar um registo preexistente da agenda telefónica
begin
  Obtenção do número de registos armazenados no ficheiro;
  se o ficheiro está vazio, então sair da função;
  Leitura da chave de pesquisa;
  Determinação do ponto de eliminação do registo no ficheiro;
  Eliminação do registo do ficheiro;
  Actualização do número de registos armazenados no ficheiro;
end
```

Figura 3.30 - Algoritmo da eliminação de um registo preexistente da agenda telefónica.

```
void ELIMINAR_REGISTO (FILE *FP)
{
   char NUMTEL[TMX];                          /* número de telefone */
   unsigned int NREG,                 /* número de registos armazenados */
                PREG;                 /* ponto de eliminação do registo */
                                                    /* alusão a funções */
   void LER_TELEFONE (char []);
   void LER_NUMREG (FILE *, unsigned int *);
   void ESCREVER_NUMREG (FILE *, unsigned int);
   unsigned int POS_ELIMINACAO (FILE *, unsigned int, char []);
   void DESLOCAR_PARA_CIMA (FILE *, unsigned int, unsigned int);
          /* obtenção do número de registos actualmente armazenados */
   LER_NUMREG (FP, &NREG);
   if (NREG == 0)
   {
      fprintf (stdout, "\nO ficheiro está vazio!\n");  return;
   }
   LER_TELEFONE (NUMTEL);              /* leitura da chave de pesquisa */
                              /* determinação do ponto de eliminação */
   PREG = POS_ELIMINACAO (FP, NREG, NUMTEL);
   if (PREG == NREG)
   {
      fprintf (stdout, "\nNão existe qualquer registo com esse "\
                       "número de telefone no ficheiro!\n");

      return;
   }
                                                /* eliminação do registo */
   if (PREG < NREG-1) DESLOCAR_PARA_CIMA (FP, NREG, PREG);
                    /* actualização do número de registos armazenados */
   ESCREVER_NUMREG (FP, NREG-1);
}
```

Figura 3.31 - Função para eliminar um registo preexistente da agenda telefónica.

A Figura 3.32 apresenta o algoritmo em linguagem natural da função LISTAR_FICHEIRO. Em primeiro lugar temos que verificar se existem de factos registos armazenados no ficheiro, o que é feito pela função LER_NUMREG. Depois é preciso fazer a impressão um a um do conteúdo dos registos. Para isso temos a função LER_REGISTO_FICHEIRO que lê o registo do ficheiro e a função ESCREVER_REGISTO_MONITOR que escreve o seu conteúdo no monitor.

A Figura 3.33 apresenta a implementação da função LISTAR_FICHEIRO. As funções auxiliares serão posteriormente apresentadas.

```
nome: Listar a agenda telefónica
begin
  Obtenção do número de registos armazenados no ficheiro;
  se o ficheiro está vazio, então sair da função;
  para todos os registos existentes no ficheiro fazer
  begin
    Ler o registo do ficheiro;
    Escrever o conteúdo do registo no monitor;
  end
end
```

Figura 3.32 - Algoritmo da listagem da agenda telefónica.

```
void LISTAR_FICHEIRO (FILE *FP)
{
   TREG REGISTO;   /* registo lido do ficheiro e escrito no monitor */
   unsigned int NREG,             /* número de registos armazenados */
                REG;                        /* contador do ciclo for */
                                                /* alusão a funções */
   void LER_NUMREG (FILE *, unsigned int *);
   void LER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG *);
   void ESCREVER_REGISTO_MONITOR (unsigned int, TREG);
          /* obtenção do número de registos actualmente armazenados */
   LER_NUMREG (FP, &NREG);
   if (NREG == 0)
   {
      fprintf (stdout, "\nO ficheiro está vazio!\n");  return;
   }
                      /* impressão um a um do conteúdo dos registos */
   fprintf (stdout, "\nImpressão do conteúdo dos registos\n");
   for (REG = 0; REG < NREG; REG++)
   {
                    /* leitura do conteúdo de um registo do ficheiro */
      LER_REGISTO_FICHEIRO (FP, REG, &REGISTO);
                     /* escrita no monitor do conteúdo de um registo */
      ESCREVER_REGISTO_MONITOR (REG, REGISTO);
   }
}
```

Figura 3.33 - Função para listar a agenda telefónica no monitor.

Vamos agora apresentar as funções auxiliares, que apareceram durante a descrição algorítmica das três operações do programa principal. A Figura 3.34 apresenta a função LER_REGISTO_TECLADO. Tem um parâmetro de saída que é o registo, cuja informação vai ser lida do teclado. A informação é validada, de maneira a assegurar que o utilizador não se esquece de preencher qualquer campo do registo. Um número de telefone é constituído apenas por caracteres numéricos, pelo que o formato de leitura impõe essa limitação aos caracteres aceites. Por uma questão de simplificação da função, a data não é validada.

A Figura 3.35 apresenta a função ESCREVER_REGISTO_MONITOR. Tem dois parâmetros de entrada que são o registo, cuja informação vai ser escrita no monitor, e o seu número de ordem no ficheiro.

A Figura 3.36 apresenta a função LER_TELEFONE. Tem um parâmetro de saída que é o número de telefone lido do teclado, que vai servir de chave de pesquisa para a eliminação do registo.

```
void LER_REGISTO_TECLADO (TREG *REG)
{
  int ST;                        /* sinalização do estado de leitura */
  fprintf (stdout, "\nLeitura dos dados de um registo\n");
  do
  {
    fprintf (stdout, "Nome? ");
    ST = fscanf (stdin, "%40[^\n]", REG->NOME);          /* NMX-1 */
    fscanf (stdin, "%*[^\n]");
    fscanf (stdin, "%*c");
  } while (ST != 1);
  do
  {
    fprintf (stdout, "Número de telefone? ");
    ST = fscanf (stdin, "%15[0123456789]", REG->NTELEF); /* TMX-1 */
    fscanf (stdin, "%*[^\n]");
    fscanf (stdin, "%*c");
  } while (ST != 1);
  do
  {
    fprintf (stdout, "Data de aniversário (DD-MM-AAAA)? ");
    ST = fscanf (stdin, "%2u-%2u-%4d", &(REG->ANIVERSARIO.DIA),\
                 &(REG->ANIVERSARIO.MES), &(REG->ANIVERSARIO.ANO));
    fscanf (stdin, "%*[^\n]");
    fscanf (stdin, "%*c");
  } while (ST != 3);
}
```

Figura 3.34 - Função para ler os dados de um registo do teclado.

```
void ESCREVER_REGISTO_MONITOR (unsigned int NREG, TREG REG)
{
  fprintf (stdout, "\nRegisto n°%u\n", NREG);
  fprintf (stdout, "Nome: %s\n", REG.NOME);
  fprintf (stdout, "Número de telefone: %s\n", REG.NTELEF);
  fprintf (stdout, "Data de aniversário: %02u-%02u-%04d\n",\
     REG.ANIVERSARIO.DIA, REG.ANIVERSARIO.MES, REG.ANIVERSARIO.ANO);
  fprintf (stdout, "\nCarregue em Enter para continuar");
  fscanf (stdin, "%*[^\n]");
  fscanf (stdin, "%*c");
}
```

Figura 3.35 - Função para escrever os dados de um registo no monitor.

```
void LER_TELEFONE (char NUMTELEF[])
{
  int ST;                        /* sinalização do estado de leitura */
  do
  {
    fprintf (stdout, "Número de telefone? ");
    ST = fscanf (stdin, "%15[0123456789]", NUMTELEF);    /* TMX-1 */
    fscanf (stdin, "%*[^\n]");
    fscanf (stdin, "%*c");
  } while (ST != 1);
}
```

Figura 3.36 - Função para ler um número de telefone do teclado.

A Figura 3.37 apresenta a função POS_INSERCAO. Tem como parâmetros de entrada o ficheiro, o número de registos armazenados e o registo. Tem como resultado de saída o índice do registo do ficheiro onde o novo registo vai ser colocado. Para ler o registo do ficheiro, usa a função LER_REGISTO_FICHEIRO e o processo de comparação usa a função **strcmp**. A pesquisa termina quando se encontra um registo cujo nome seja alfabeticamente posterior ao nome do novo registo, o que implica que no caso de já existir um registo com esse nome, o novo registo será colocado a seguir. Se o registo estiver alfabeticamente depois de todos os que já se encontram no ficheiro, então a função devolve o número de registos armazenados no ficheiro, ou seja, NR, como indicação que o novo registo deverá ser colocado no fim do ficheiro.

```
unsigned int POS_INSERCAO (FILE *FP, unsigned int NR, TREG REG)
{
   TREG REGAUX;                                    /* registo auxiliar */
   unsigned int R = 0;           /* posição de inserção determinada */

   void LER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG *);

   do
   {
      LER_REGISTO_FICHEIRO (FP, R, &REGAUX);
      if ( strcmp (REG.NOME, REGAUX.NOME) < 0 ) break;
      R++;
   } while (R < NR);

   return R;
}
```

Figura 3.37 - Função para determinar a posição de inserção do registo.

A Figura 3.38 apresenta a função POS_ELIMINACAO. Tem como parâmetros de entrada o ficheiro, o número de registos armazenados e a chave de pesquisa. Tem como resultado de saída o índice do registo do ficheiro onde se encontra o registo a ser eliminado. Mais uma vez o processo de comparação usa a função **strcmp**. A pesquisa termina quando se encontra o primeiro registo cujo número de telefone é igual ao número indicado. Se não existir, então a função devolve o número de registos armazenados no ficheiro, ou seja, NR, como indicação de registo inexistente no ficheiro.

```
unsigned int POS_ELIMINACAO (FILE *FP, unsigned int NR, char NTEL[])
{
   TREG REGAUX;                                    /* registo auxiliar */
   unsigned int R = 0;         /* posição de eliminação determinada */

   void LER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG *);

   do
   {
      LER_REGISTO_FICHEIRO (FP, R, &REGAUX);
      if ( !strcmp (NTEL, REGAUX.NTELEF) ) break;
      R++;
   } while (R < NR);

   return R;
}
```

Figura 3.38 - Função para determinar a posição de eliminação do registo.

A inserção do registo no ficheiro exige deslocar para baixo os registos do ficheiro a partir do índice que foi detectado como local de inserção do novo registo. A Figura 3.39 apresenta a função DESLOCAR_PARA_BAIXO. Tem como parâmetros de entrada o ficheiro, o número de registos armazenados e o índice de inserção do registo. Cada registo é lido do ficheiro e escrito de novo no ficheiro mas uma posição mais abaixo. O deslocamento dos registos é feito do fim do ficheiro até ao índice de inserção, como se exemplifica na parte esquerda da Figura 3.41.

```
void DESLOCAR_PARA_BAIXO (FILE *FP, unsigned int NR, unsigned int P)
{
  TREG REGAUX;                                    /* registo auxiliar */
  unsigned int R;                               /* contador do ciclo for */

  void LER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG *);
  void ESCREVER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG);

  for (R = NR; R > P; R--)
  {
    LER_REGISTO_FICHEIRO (FP, R-1, &REGAUX);
    ESCREVER_REGISTO_FICHEIRO (FP, R, REGAUX);
  }
}
```

Figura 3.39 - Função para deslocar para baixo os registos do ficheiro.

A eliminação do registo no ficheiro exige deslocar para cima os registos do ficheiro a partir do índice que foi detectado como local de eliminação do novo registo, a não ser que o registo seja o último registo do ficheiro. A Figura 3.40 apresenta a função DESLOCAR_PARA_ACIMA. Tem como parâmetros de entrada o ficheiro, o número de registos armazenados e o índice de eliminação do registo. Cada registo é lido do ficheiro e escrito de novo no ficheiro mas uma posição mais acima. O deslocamento dos registos é feito do índice de eliminação até ao fim do ficheiro, como se exemplifica na parte direita da Figura 3.41.

```
void DESLOCAR_PARA_CIMA (FILE *FP, unsigned int NR, unsigned int P)
{
  TREG REGAUX;                                    /* registo auxiliar */
  unsigned int R;                               /* contador do ciclo for */

  void LER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG *);
  void ESCREVER_REGISTO_FICHEIRO (FILE *, unsigned int, TREG);

  for (R = P; R < NR-1; R++)
  {
    LER_REGISTO_FICHEIRO (FP, R+1, &REGAUX);
    ESCREVER_REGISTO_FICHEIRO (FP, R, REGAUX);
  }
}
```

Figura 3.40 - Função para deslocar para cima os registos do ficheiro.

O número de registos do ficheiro está armazenado no seu cabeçalho. Para determinar o número de registos, a função LER_NUMREG, que se apresenta na Figura 3.42, limita-se a ler o cabeçalho. A função tem como parâmetro de entrada o ficheiro e como parâmetro de saída o número de registos lido do cabeçalho do ficheiro. Para actualizar o número de registos, a função ESCREVER_NUMREG, que se apresenta na Figura 3.43, escreve no cabeçalho o número actualizado de registos. A função tem como parâmetros de entrada o ficheiro e o número de registos a escrever no cabeçalho do ficheiro.

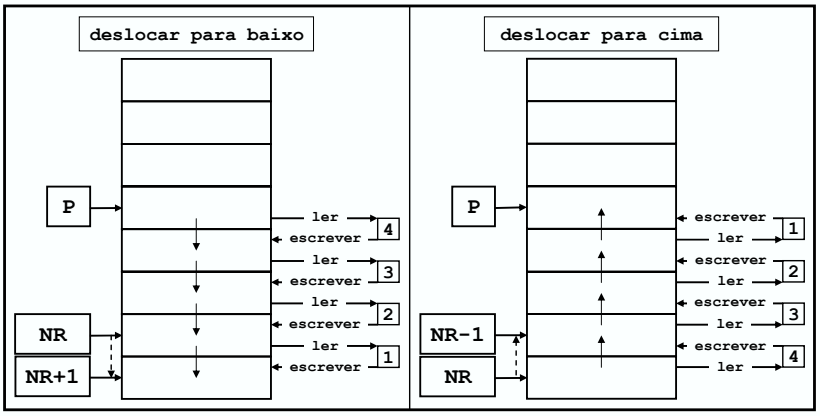

Figura 3.41 - Visualização gráfica das operações de deslocamento.

A função **fseek** permite colocar o ponteiro de posição de leitura/escrita no início do ficheiro. Após a escrita do cabeçalho e uma vez que a sua actualização é feita após a inserção ou eliminação de um registo é necessário usar a função **fflush**, para que o conteúdo do ficheiro seja imediatamente actualizado, ou seja, para assegurar que as alterações acabadas de fazer estão visíveis para a operação seguinte.

```
void LER_NUMREG (FILE *FP, unsigned int *NR)
{
  if ( fseek (FP, 0, SEEK_SET) != 0 )
  {
    perror ("erro no posicionamento do ficheiro - cabeçalho");
    exit (EXIT_FAILURE);
  }
  if ( fread (NR, sizeof (unsigned int), 1, FP) != 1 )
  {
    fprintf (stderr, "erro na leitura do cabeçalho do ficheiro\n");
    exit (EXIT_FAILURE);
  }
}
```

Figura 3.42 - Função para ler o número de registos armazenado no cabeçalho do ficheiro.

```
void ESCREVER_NUMREG (FILE *FP, unsigned int NR)
{
  if ( fseek (FP, 0, SEEK_SET) != 0 )
  {
    perror ("erro no posicionamento do ficheiro - cabeçalho");
    exit (EXIT_FAILURE);
  }
  if ( fwrite (&NR, sizeof (unsigned int), 1, FP) != 1 )
  {
    fprintf (stderr, "erro na escrita do cabeçalho do ficheiro\n");
    exit (EXIT_FAILURE);
  }
  if ( fflush (FP) != 0 )
  {
    perror ("erro na escrita efectiva no ficheiro");
    exit (EXIT_FAILURE);
  }
}
```

Figura 3.43 - Função para escrever o número de registos no cabeçalho do ficheiro.

Para ler ou escrever um registo no ficheiro primeiro é necessário colocar o ponteiro de posição de leitura/escrita no registo com o índice pretendido, usando a função **fseek**. A Figura 3.44 apresenta a função LER_REGISTO_FICHEIRO, que tem como parâmetros de entrada o ficheiro e o número de ordem do registo a ler e como parâmetro de saída o registo lido. A Figura 3.45 apresenta a função ESCREVER_REGISTO_FICHEIRO, que tem como parâmetros de entrada o ficheiro, o número de ordem do registo e o registo a escrever no ficheiro.

```
void LER_REGISTO_FICHEIRO (FILE *FP, unsigned int P, TREG *REG)
{
  if (fseek (FP, sizeof(unsigned int)+P*sizeof(TREG), SEEK_SET)!=0)
  {
    perror ("erro no posicionamento do ficheiro - registo");
    exit (EXIT_FAILURE);
  }
  if ( fread (REG, sizeof (TREG), 1, FP) != 1 )
  {
    fprintf (stderr, "erro na leitura do registo do ficheiro\n");
    exit (EXIT_FAILURE);
  }
}
```

Figura 3.44 - Função para ler um registo do ficheiro.

```
void ESCREVER_REGISTO_FICHEIRO (FILE *FP, unsigned int P, TREG REG)
{
  if (fseek (FP, sizeof(unsigned int)+P*sizeof(TREG), SEEK_SET)!=0)
  {
    perror ("erro no posicionamento do ficheiro - registo");
    exit (EXIT_FAILURE);
  }
  if (fwrite (&REG, sizeof (TREG), 1, FP) != 1)
  {
    fprintf (stderr, "erro na escrita do registo no ficheiro\n");
    exit (EXIT_FAILURE);
  }
}
```

Figura 3.45 - Função para escrever um registo no ficheiro.

3.12 Leituras recomendadas

- 12º capítulo do livro "C A Software Approach", 3ª edição, de Peter A. Darnell e Philip E. Margolis, da editora Springer-Verlag, 1996.

Capítulo 4

RECURSIVIDADE

Sumário

Neste capítulo apresentamos exemplos da implementação de funções recursivas. Uma função recursiva, ou recorrente, é uma função que se invoca a si própria. Tal como a decomposição hierarquizada, a recursividade permite ao programador decompor um problema em problemas mais pequenos, que têm a particularidade de serem exactamente do mesmo tipo do problema original.

O que não significa que os algoritmos recursivos sejam mais eficientes que os algoritmos iterativos equivalentes. Pelo contrário, alguns algoritmos recursivos desencadeiam um número arbitrariamente grande de invocações sucessivas da função, pelo que, nestes casos temos uma ineficiência acrescida associada à invocação de funções. Por outro lado, alguns algoritmos recursivos são computacionalmente ineficientes, devido ao facto de as múltiplas invocações recursivas repetirem o cálculo de valores.

No entanto, os algoritmos recursivos são apropriados para resolver problemas que são por natureza recursivos. Por vezes, existem problemas que têm soluções recursivas que são simples, concisas, elegantes e para os quais é difícil esboçar soluções iterativas com a mesma simplicidade e clareza. Mas, como alguns algoritmos recursivos são menos eficientes que os algoritmos iterativos equivalentes, a verdadeira importância da recursividade consiste em resolver esses problemas para os quais não existem soluções iterativas simples.

Vamos analisar alguns exemplos clássicos de algoritmos recursivos, com excepção de algoritmos de ordenação recursivos, que são algoritmos de ordenação muito eficientes, mas que serão tratados mais tarde no capítulo de Pesquisa e Ordenação.

4.1 Funções recursivas

Tipicamente, a solução de um problema repetitivo pode ser resolvido de forma iterativa utilizando explicitamente um ciclo repetitivo, ou de forma recursiva utilizando a invocação sucessiva da solução. A linguagem C, tal como outras linguagens de programação de alto nível, permite a definição de funções recursivas. Uma função recursiva, ou recorrente, é uma função que é definida em termos de si próprio, ou seja, a função invoca-se a si própria na parte executiva. Este tipo de recursividade, ou recorrência, designa-se por **recursividade directa**.

A grande vantagem da criação de uma função recursiva, utilizando para o efeito um algoritmo recursivo, é o de permitir desencadear um número arbitrariamente grande de repetições de instruções, sem contudo, usar explicitamente estruturas de controlo repetitivo. O que não significa que os algoritmos recursivos sejam mais eficientes que os algoritmos iterativos equivalentes.

Tal como a decomposição hierarquizada, a recursividade permite ao programador decompor um problema em problemas mais pequenos, que têm a particularidade de serem exactamente do mesmo tipo do problema original. Portanto, um algoritmo recursivo resolve o problema resolvendo uma instância menor do mesmo problema. Consequentemente, a complexidade do problema será diminuída até que a solução seja óbvia ou conhecida.

Os problemas que podem ser resolvidos recursivamente têm normalmente as seguintes características. Um ou mais casos de paragem, em que a solução é não recursiva e conhecida, e casos em que o problema pode ser diminuído recursivamente até se atingirem os casos de paragem. Portanto, quando se implementa uma função recursiva temos que assegurar que existe uma instrução que para a invocação recursiva e que calcula efectivamente um valor, e por outro lado, que o valor das sucessivas invocações é alterado de maneira a atingir esse valor de paragem. Usualmente um algoritmo recursivo usa uma instrução condicional **if** com a forma que se apresenta na Figura 4.1.

```
nome: Algoritmo recursivo
begin
  se o caso de paragem é atingido?
  então Resolver o problema
  senão Partir o problema em problemas mais pequenos, usando
        para esse efeito, uma ou mais invocações recursivas
end
```

Figura 4.1 - Solução genérica de um algoritmo recursivo.

Devemos ter sempre presente, que a invocação de uma função, produz um desperdício de tempo e de memória devido à criação de uma cópia local dos parâmetros de entrada que são passados por valor, bem como na salvaguarda do estado do programa na altura da invocação, na memória de tipo pilha (*stack*), para que, o programa possa retomar a execução na instrução seguinte à invocação da função, quando a execução da função terminar. Este problema é ainda mais relevante no caso de funções recursivas, principalmente quando existem múltiplas invocações recursivas. Geralmente, quando existem soluções recursivas e iterativas para o mesmo problema, a solução recursiva requer mais tempo de execução e mais espaço de memória devido às invocações extras da função.

4.2 Cálculo do factorial

Vamos considerar o problema do cálculo do factorial. A solução iterativa, cuja função se apresenta na Figura 4.2, decorre directamente da definição $n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1$, e calcula o produto acumulado de forma repetitiva para a variável auxiliar FACT, utilizando explicitamente um ciclo repetitivo **for**.

```
unsigned int FACTORIAL (int N)
{
   int I, FACT = 1;
   if (N < 0) return 0;    /* invocação anormal (valor devolvido 0) */
   for (I = 2; I <= N; I++) FACT *= I;     /* cálculo do produtório */
   return FACT;                        /* devolução do valor calculado */
}
```

Figura 4.2 - Função factorial implementada de forma iterativa.

Mas, se tivermos em conta que $n! = n \times (n-1)!$, sendo que $0! = 1$, então a função factorial pode ser implementada de forma recursiva, tal como se apresenta na Figura 4.3.

```
unsigned int FACTORIAL (int N)
{
   if (N < 0) return 0;    /* invocação anormal (valor devolvido 0) */
   if (N == 0) return 1;                       /* condição de paragem */
   return N * FACTORIAL (N-1);                 /* invocação recursiva */
}
```

Figura 4.3 - Função factorial implementada de forma recursiva.

A primeira condição de teste evita que a função entre num processo recursivo infinito, caso a função seja invocada para um valor de N negativo. Uma das instruções da função factorial recursiva é a **invocação recursiva**, quando a função se invoca a si própria. Para cada nova invocação, o valor do factorial a calcular é diminuído de uma unidade, até que acabará por atingir o valor zero. Nesse caso, a função não se invoca mais e retorna o valor 1. Assim temos que N ser igual a zero, funciona como **condição de paragem** da recursividade. Quando a recursividade termina, então é calculado o valor final da função, no retorno das sucessivas invocações da função.

A Figura 4.4 representa a análise gráfica da invocação da função factorial recursiva para o cálculo de 3!. Cada caixa representa o espaço de execução de uma nova invocação da função. Se considerarmos cada caixa como um nó estamos perante o que se designa por **árvore de recorrência** da função. Temos que $3! = 3 \times 2!$, por sua vez $2! = 2 \times 1!$, por sua vez $1! = 1 \times 0!$, e finalmente a invocação recursiva termina com a condição de paragem, sendo $0! = 1$. O valor de 3! é calculado no retorno das sucessivas invocações da função recursiva, pelo que, $3! = 1 \times 1 \times 2 \times 3 = 6$.

A implementação recursiva não tem qualquer vantagem sobre a implementação repetitiva. Mas, tem a desvantagem associada à invocação sucessiva de funções, que é o tempo gasto na invocação de uma função e o eventual esgotamento da memória de tipo pilha. Apesar da implementação recursiva não usar variáveis locais, no entanto, gasta mais memória porque a função tem de fazer a cópia do parâmetro de entrada que é passado por valor. Pelo que, a versão iterativa é mais eficiente e portanto, deve ser usada em detrimento da versão recursiva. O maior factorial que se consegue calcular em aritmética inteira de 32 *bits* é o 12! que é igual a 479001600.

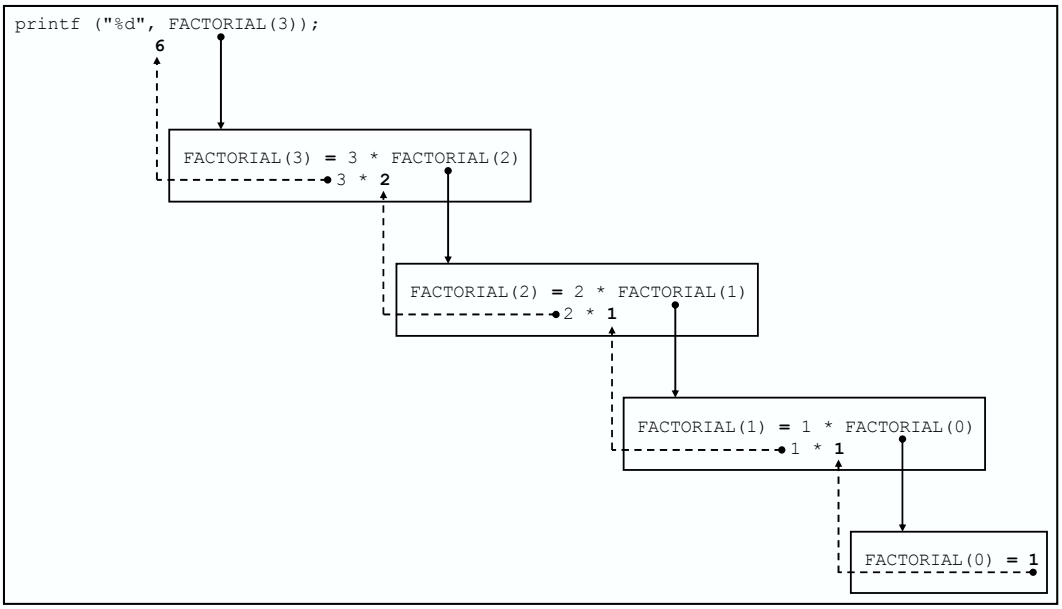

Figura 4.4 - Visualização gráfica da invocação da função factorial recursiva.

4.3 Expansão em série de Taylor

No computador as funções trigonométricas e logarítmicas são implementadas através da expansão em série de Taylor. O termo geral da expansão em série de Taylor da função seno é dado pela seguinte expressão à esquerda, sendo apresentada à direita a sua expansão para os primeiros cinco termos da série.

$$\text{seno}(x,N) = \sum_{n=0}^{N-1} (-1)^n \times \frac{x^{2n+1}}{(2n+1)!} \qquad \text{seno}(x,5) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots$$

Como é bem sabido, a função factorial explode rapidamente, assumindo valores de tal maneira elevados que só podem ser armazenados em variáveis de tipo real, com a consequente perda de precisão. Pelo que, uma forma de resolver o problema, passa por eliminar a necessidade do cálculo do factorial recorrendo a um processo recorrente. Uma vez que se trata de uma série geométrica, poderíamos calcular cada elemento da série a partir do elemento anterior e adicionar os elementos calculados. Mas, se aplicarmos de um modo sistemático a propriedade distributiva da multiplicação relativamente à adição, obtemos uma expressão, em que o cálculo do factorial desaparece.

$$\text{seno}(x,N) = x \times \left(1 - \frac{x^2}{2\times 3} \times \left(1 - \frac{x^2}{4\times 5} \times \left(1 - \frac{x^2}{6\times 7} \times \left(1 - \frac{x^2}{8\times 9} \times \ldots\right)\right)\right)\right)$$

Esta expressão pode ser calculada de duas maneiras. Podemos calculá-la de baixo para cima (*bottom-up*), ou seja, do termo P_4 para o termo P_1, através de um processo iterativo, ou em alternativa podemos calcular a expressão de cima para baixo (*top-down*), ou seja, do termo P_1 para o termo P_4, através de um processo recursivo.

Vamos começar por analisar a implementação iterativa. Neste caso vamos calcular de forma repetitiva a expressão entre parêntesis da direita para a esquerda, ou seja, do termo P_4 até ao termo P_1, utilizando o passo iterativo dado pela seguinte expressão.

$$P_{i-1} = 1 - \frac{x^2}{2 \times i \times (2 \times i + 1)} \times P_i$$

Como estamos perante o cálculo de um produto acumulado, P_i tem de ser inicializado a 1.0, que é o elemento neutro do produto. Depois do cálculo repetitivo, temos que o valor final do seno é $\text{seno}(x, N) = x \times P_1$. A Figura 4.5 apresenta a função iterativa.

```
double SENO_TAYLOR_ITERATIVO (double X, int N)
{
  int I;  double TAYLOR = 1.0;

  for (I = N; I > 0; I--)
    TAYLOR = 1 - X*X / (2*I * (2*I+1)) * TAYLOR;
  return X * TAYLOR;
}
```

Figura 4.5 - Função iterativa que calcula a expansão em série de Taylor da função seno.

Vamos agora analisar a implementação recursiva. Neste caso vamos calcular a expressão entre parêntesis da esquerda para a direita, ou seja, do termo P_1 até ao termo P_4, sendo que, cada termo é calculado em função do termo seguinte ainda por calcular. Daí a necessidade do processo ser recursivo. O passo recursivo é dado pela seguinte expressão.

$$\text{Seno}(x, N, i) = 1 - \frac{x^2}{2 \times i \times (2 \times i + 1)} \times \text{Seno}(x, N, i + 1)$$

O processo recursivo é terminado quando o termo a calcular é o último termo desejado, ou seja, quando i é igual a N, cujo valor é dado pela seguinte expressão.

$$\text{Seno}(x, N, N) = 1 - \frac{x^2}{2 \times N \times (2 \times N + 1)}$$

Caso o número de termos pretendidos na expansão seja apenas um, o valor final do seno é x e não há a necessidade de invocar o cálculo recursivo. Caso, o número de termos seja maior do que um, então o valor final do seno é igual a x vezes o valor obtido pelo cálculo recursivo do primeiro elemento, ou seja, $\text{seno}(x, N) = x \times \text{Seno}(x, N, 1)$.

Pelo que, para implementar o cálculo de forma recursiva precisamos de decompor a solução em duas funções, tal como se apresenta na Figura 4.6. Uma é a função recursiva propriamente dita que implementa a expansão em série de Taylor, e a outra é a função que calcula o valor final e que invoca, caso seja necessário, a função recursiva.

```
double TAYLOR (double X, int N, int I)
{                                              /* condição de paragem */
  if (I == N) return 1 - X*X / (2*N * (2*N+1));
  return 1 - X*X / (2*I * (2*I+1)) * TAYLOR (X, N, I+1);
}
double SENO_TAYLOR_RECURSIVO (double X, int N)
{
  if (N == 1) return X;
  return X * TAYLOR (X, N, 1);
}
```

Figura 4.6 - Função recursiva que calcula a expansão em série de Taylor da função seno.

Neste exemplo, a versão recursiva para além do desperdício de tempo gasto com a invocação sucessiva da função, tem uma implementação mais extensa em termos de código, uma vez que a solução tem de ser decomposta em duas funções. Pelo que, a versão iterativa é mais eficiente e simples de programar.

4.4 Números de Fibonacci

Um exemplo de cálculo recursivo mais complexo do que o cálculo do factorial é o cálculo dos números de Fibonacci, que são definidos pela seguinte relação de recorrência e cuja implementação recursiva se apresenta na Figura 4.7.

$$\text{Fibonacci(n)} = \begin{cases} 0, \text{se } n = 0 \\ 1, \text{se } n = 1 \\ \text{Fibonacci(n - 1)} + \text{Fibonacci(n - 2)}, \text{se } n \geq 2 \end{cases}$$

```
unsigned int FIBONACCI (int N)
{
  if (N <= 0) return 0;                          /* Fibonacci de 0 */
  if (N <= 2) return 1;                    /* Fibonacci de 1 e de 2 */
  return FIBONACCI (N-1) + FIBONACCI (N-2);  /* Fibonacci de n > 2 */
}
```

Figura 4.7 - Função que calcula os números de Fibonacci de forma recursiva.

A Figura 4.8 apresenta a árvore de recorrência do cálculo do Fibonacci de 5.

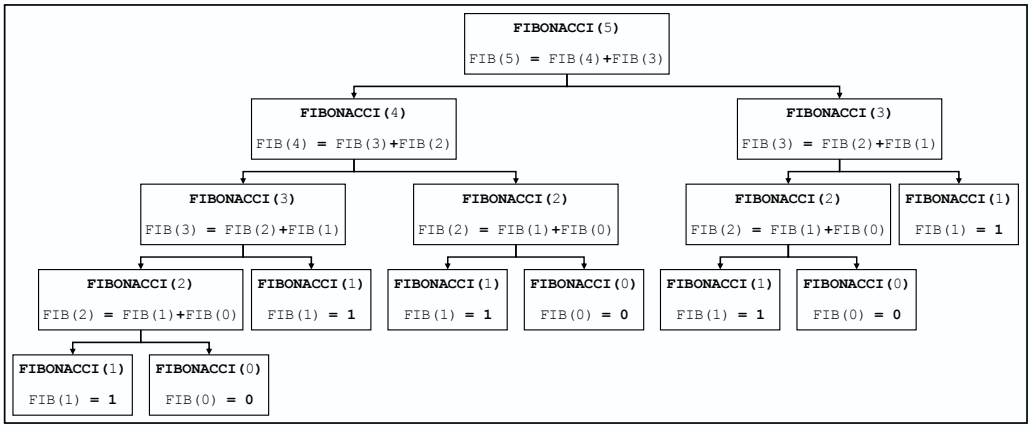

Figura 4.8 - Visualização gráfica da invocação da função Fibonacci recursiva.

Como se pode ver, esta solução é computacionalmente ineficiente, porque para calcular o Fibonacci de 5, calcula repetidamente alguns valores intermédios. Mais concretamente, duas vezes o Fibonacci de 3, três vezes o Fibonacci de 2, cinco vezes o Fibonacci de 1 e três vezes o Fibonacci de 0. Por outro lado, devido à dupla invocação recursiva o número de operações explode rapidamente. Como se pode ver na Figura 4.8, o cálculo do Fibonacci de 5 custa 7 adições. Para se calcular o Fibonacci de 6, temos que calcular o Fibonacci de 5 e o Fibonacci de 4, o que dá um total de 12 adições. O Fibonacci de 7 custa 20 adições, o Fibonacci de 8 custa 33 adições. Ou seja, o número de adições quase que duplica quando se incrementa o argumento da função de uma unidade. Pelo que, o tempo de execução desta função é praticamente exponencial.

Uma forma de resolver problemas recursivos de maneira a evitar o cálculo repetido de valores consiste em calcular os valores de baixo para cima e utilizar um agregado para manter os valores entretanto calculados. Este método designa-se por programação dinâmica e reduz o tempo de cálculo à custa da utilização de mais memória para armazenar valores internos. Esta solução é apresentada na Figura 4.9, sendo o cálculo efectuado do Fibonacci de 0 até ao Fibonacci de N. Os três primeiros valores, ou seja, o Fibonacci de 0, de 1 e de 2, são colocados na inicialização do agregado.

```
unsigned int FIBONACCI (int N)
{
  int I;  unsigned int FIB[50] = {0,1,1};

  if (N <= 0) return 0;                          /* Fibonacci de 0 */
  if (N <= 2) return 1;                    /* Fibonacci de 1 e de 2 */

  for (I = 3; I <= N; I++)   /* para calcular o Fibonacci de n > 2 */
    FIB[I] = FIB[I-1] + FIB[I-2];

  return FIB[N];
}
```

Figura 4.9 - Função que calcula os números de Fibonacci de forma dinâmica.

Mas, se repararmos bem na solução dinâmica verificamos que de facto a cálculo do Fibonacci de N só precisa dos valores do Fibonacci de N-1 e do Fibonacci de N-2, pelo que, podemos substituir a utilização do agregado por apenas três variáveis simples. Uma para armazenar o valor a calcular, que designamos por PROXIMO, outra para armazenar o valor acabado de calcular, que designamos por ACTUAL e outra para armazenar o valor calculado anteriormente, que designamos por ANTERIOR. A Figura 4.10 apresenta esta versão iterativa. O valor inicial de ANTERIOR é 0 e corresponde ao Fibonacci de 0, o valor inicial de ACTUAL é 1 e corresponde ao Fibonacci de 1 e o valor de PROXIMO corresponde ao Fibonacci que se pretende calcular de forma iterada, e é igual à soma do ANTERIOR com o ACTUAL.

```
unsigned int FIBONACCI (int N)
{
  int I;  unsigned int ANTERIOR = 0, ACTUAL = 1, PROXIMO;

  if (N <= 0)  return 0;                         /* Fibonacci de 0 */
  if (N == 1)  return 1;                         /* Fibonacci de 1 */

  for (I = 2; I <= N; I++)                  /* Fibonacci de n >= 2 */
  {
    PROXIMO = ACTUAL + ANTERIOR;
    ANTERIOR = ACTUAL;
    ACTUAL = PROXIMO;
  }

  return PROXIMO;
}
```

Figura 4.10 - Função que calcula os números de Fibonacci de forma iterativa.

Esta solução iterativa é a melhor das três soluções. É a mais rápida e a que gasta menos memória. O tempo de execução desta solução é linear, ou seja, o cálculo do Fibonacci de 2N custa aproximadamente o dobro do tempo do cálculo do Fibonacci de N. O maior número de Fibonacci que se consegue calcular em aritmética inteira de 32 *bits* é o Fibonacci de 47 que é igual a 2971215073.

4.5 Cálculo dos coeficientes binomiais

O coeficiente binomial C(n,k), que representa o número de combinações de n elementos a k elementos é dado pela seguinte expressão.

$$C(n, k) = \frac{n!}{(n - k)! \cdot k!}$$

Em muitas situações, mesmo para valores finais representáveis em variáveis inteiras, os valores parcelares podem ficar corrompidos, uma vez que o factorial rapidamente dá *overflow* em aritmética inteira. Por exemplo, duas situações limites são o C(n,0) e o C(n,n), em que o número de combinações é igual a 1 (n!/n!). Pelo que, a utilização da definição para calcular C(n,k) está normalmente fora de questão e neste caso nem sequer podemos recorrer a aritmética real, devido à consequente perda de precisão. Uma forma de resolver o problema, passa por eliminar a necessidade do cálculo do factorial recorrendo a um processo iterativo, descoberto por Blaise Pascal e que é conhecido pelo Triângulo de Pascal. Como se pode ver na Figura 4.11, cada elemento com excepção dos elementos terminais de cada linha, ou seja C(n,0) e C(n,n), é calculado através da soma dos valores da linha anterior. Ou seja, C(n,k) = C(n-1,k) + C(n-1,k-1), com n>k>0. A Figura 4.11 representa de forma gráfica o cálculo de C(5,3) que é igual a 10. Repare que existem valores que são usados duas vezes, o que significa que também vão ser calculados duas vezes.

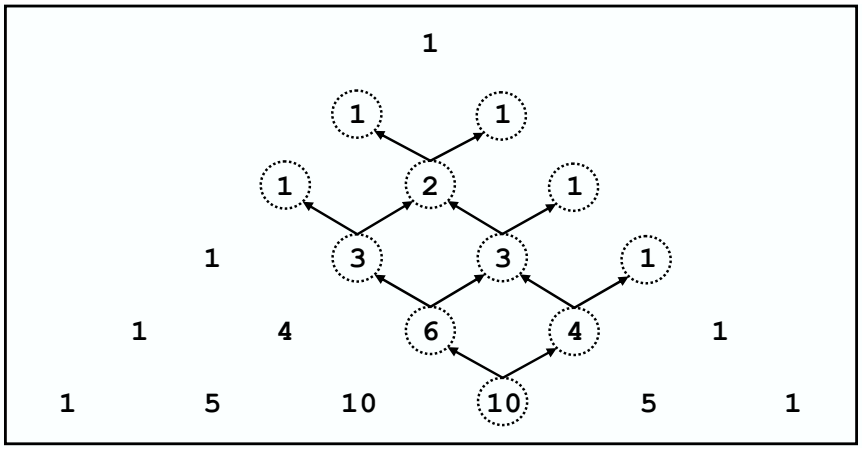

Figura 4.11 - Triângulo de Pascal.

A função recursiva para calcular o coeficiente binomial, que se apresenta na Figura 4.12, invoca-se recursivamente duas vezes e tem como condições de paragem os elementos terminais, cujo valor é 1. A primeira condição de teste evita que a função entre num processo recursivo infinito, caso a função seja invocada para k menor do que n.

```
unsigned int COMBINACOES (int N, int K)
{
   if (K > N) return 0;     /* invocação anormal (valor devolvido 0) */
   if ((K == 0) || (K == N)) return 1;      /* condições de paragem */
   return COMBINACOES (N-1, K) + COMBINACOES (N-1, K-1);
}
```

Figura 4.12 - Função recursiva para calcular os coeficientes binomiais.

Esta implementação é no entanto pouco prática para valores de n e k elevados, antes de mais porque, tal como no caso dos números de Fibonacci, repete o cálculo de certos valores e portanto, é computacionalmente ineficiente. Mas, o grande problema deste algoritmo é que devido à dupla invocação recursiva a invocação também explode rapidamente.

A solução dinâmica, que se apresenta na Figura 4.13, implica a utilização de um agregado bidimensional para armazenar os coeficientes binomiais, à medida que estes são calculados.

```
unsigned int COMBINACOES (int N, int K)
{
  int I, J, MIN;  unsigned int BICOEF[36][36];

  if (K > N) return 0;     /* invocação anormal (valor devolvido 0) */
  if ((K == 0) || (K == N)) return 1;       /* C(n,0) = C(n,n) = 1 */

  for (I = 0; I <= N; I++)
  {
    MIN = (I <= K) ? I : K;                      /* mínimo de I e K */
    for (J = 0; J <= MIN; J++)
      if ( (J == 0) || (J == I) )  BICOEF[I][J] = 1;
      else  BICOEF[I][J] = BICOEF[I-1][J-1] + BICOEF[I-1][J];
  }

  return BICOEF[N][K];
}
```

Figura 4.13 - Função dinâmica para calcular os coeficientes binomiais.

Enquanto que a solução recursiva calcula $2\times C(n,k)-1$ termos, a solução dinâmica calcula aproximadamente, por defeito, $N\times K$ termos. Os maiores coeficientes binomiais que se conseguem calcular em aritmética inteira de 32 *bits* são o C(35,16) e o C(35,19), que são iguais a 4059928950.

4.6 Cálculo das permutações

Consideremos agora que se pretende imprimir todas as permutações de um conjunto de N caracteres. Por exemplo, se o conjunto de caracteres for {a,b,c}, então o conjunto das permutações é { (a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a) }, ou seja, existem N! permutações. Podemos obter um algoritmo simples para gerar todas as permutações de um conjunto de caracteres, se construirmos um algoritmo recursivo.

Por exemplo, as permutações do conjunto {a,b,c,d} são os quatro seguintes grupos de permutações: **a** seguido das permutações do conjunto {**b**,**c**,**d**}; **b** seguido das permutações do conjunto {**a**,**c**,**d**}; **c** seguido das permutações do conjunto {**b**,**a**,**d**}; e **d** seguido das permutações do conjunto {**b**,**c**,**a**}.

É então possível resolver o problema para N caracteres, se tivermos um algoritmo recursivo que funcione para $N-1$ caracteres. Em cada passo do processo coloca-se um carácter à esquerda do conjunto e calculam-se recursivamente as permutações dos $N-1$ restantes caracteres à direita. Depois troca-se sucessivamente o carácter da esquerda com um dos caracteres da direita, de forma a ter experimentado todos os N grupos possíveis de permutações dos $N-1$ caracteres. Durante o processo recursivo, quando se tiver atingido o carácter mais à direita, imprime-se a sequência de permutações acabada de gerar.

A Figura 4.14 apresenta a função de cálculo das permutações. Assume-se que LISTA é uma cadeia de N caracteres terminada com o carácter nulo, de maneira a simplificar a sua escrita no monitor. Para trocar os caracteres utiliza-se a função TROCA.

```
void TROCA (char *CAR_I, char *CAR_J)
{
  char TEMP;

  TEMP = *CAR_I; *CAR_I = *CAR_J; *CAR_J = TEMP;
}

void PERMUTACOES (char LISTA[], int I, int N)
{
  int J;

  if (I == N)                                  /* condição de paragem */
    printf ("%s\n", LISTA[J]);     /* imprimir a permutação gerada */
  else for (J = I; J <= N; J++)          /* para todos os caracteres */
       {
         TROCA (&LISTA[I], &LISTA[J]); /* por o carácter à direita */
         PERMUTACOES (LISTA, I+1, N);/* permutar os n-1 caracteres */
         TROCA (&LISTA[J], &LISTA[I]);/* repor o carácter no sítio */
       }
}
```

Figura 4.14 - Função recursiva que gera as permutações de um conjunto de caracteres.

A Figura 4.15 mostra a utilização da função. A invocação inicial da função é **PERMUTACOES (LISTA, 0, N−1);**, para permutar todos os caracteres que se encontram na cadeia de caracteres LISTA, sendo que o primeiro carácter está na posição 0 e o último carácter está na posição N−1.

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

void PERMUTACOES (char [], unsigned int, unsigned int);

int main (void)
{
  char LISTA[11];                    /* cadeia de caracteres LISTA */
  unsigned int K;                 /* número de caracteres de LISTA */

  printf ("Caracteres a permutar -> ");
  scanf ("%10s", LISTA);  /* leitura da cadeia de caracteres LISTA */

  K = strlen (LISTA);      /* determinação do número de caracteres */

  printf ("Permutações\n\n");

  PERMUTACOES (LISTA, 0, K-1);

  return 0;
}

void TROCA (char *CAR_I, char *CAR_J)
...
void PERMUTACOES (char LISTA[], unsigned int I, unsigned int N)
...
```

Figura 4.15 - Programa que utiliza a função que gera as permutações.

4.7 Cálculo do determinante de uma matriz quadrada

O cálculo do determinante de uma matriz com dimensão elevada é normalmente complexo, a não ser que a matriz seja uma matriz diagonal. Pelo que, uma maneira de simplificar o cálculo consiste em transformar a matriz, numa matriz diagonal. Esta transformação da matriz pode ser feita recursivamente, através de operações de adição e subtracção de colunas da matriz, operações essas que não afectam o valor do seu determinante. A Figura 4.16 apresenta o algoritmo em pseudocódigo e linguagem natural do cálculo recursivo do determinante de uma matriz quadrada de dimensão N.

Quando o elemento da última linha e última coluna da matriz é nulo a coluna tem de ser trocada com outra coluna, cujo último elemento não seja nulo, de forma a colocar um valor não nulo na diagonal. Caso não haja nenhuma coluna nessa situação, então é sinal que todos os elementos da última linha são nulos, pelo que, o determinante é nulo. Sempre que se trocam duas colunas de uma matriz o determinante tem de ser multiplicado por −1. Ao dividir-se a última coluna pelo último elemento põe-se em evidência esse elemento e depois para anular a última linha da matriz, basta subtrair todas as colunas menos a última, pela última coluna multiplicada pelo último elemento da coluna a processar. Uma vez que, depois deste processamento a última linha da matriz é constituída apenas por zeros, com excepção do último elemento da linha, ou seja, o elemento que está na diagonal, agora o determinante desta matriz é igual a este valor multiplicado pelo determinante da matriz de ordem N−1. O processo é invocado recursivamente até N ser igual a 1. Nessa altura o determinante é o próprio valor. Em alternativa pode-se parar o processo recursivo quando N é igual a 2, uma vez que é fácil calcular o determinante dessa matriz.

```
nome: Calculo do determinante (MATRIZ, N)                          Função
begin
  if (N = 1)  then Calculo do determinante := MATRIZ[1,1]
  else begin
         if (Matriz[N,N] = 0)
         then if existe coluna com último elemento diferente de zero?
              then Trocar essa coluna com a última
                   e multiplicar o determinante por -1
              else Calculo do determinante := 0;
         if (Matriz[N,N] <> 0)
         then begin
                Dividir a última coluna pelo último elemento pondo-o
                em evidência;
                Subtrair todas as colunas menos a última,
                pela última coluna multiplicada pelo último elemento
                da coluna a processar;
                Calculo do determinante := Matriz[N,N]
                            * Calculo do determinante (MATRIZ, N-1);
         end
  end
end
```

Figura 4.16 - Algoritmo do cálculo recursivo do determinante.

A Figura 4.17 apresenta a função e a Figura 4.18 apresenta a aplicação da função sobre uma matriz quadrada de 4×4 e as sucessivas matrizes, equivalentes para efeito do cálculo do determinante, que vão sendo obtidas após cada invocação recursiva. Quando se atinge a matriz de um só elemento, a matriz original foi completamente transformada numa matriz diagonal. Agora o determinante é calculado no retorno das sucessivas invocações da função recursiva e é igual ao produto dos elementos na diagonal. Logo, o valor do determinante é 2.0 × -1.0 × -1.5 × 5.0, ou seja, é igual a 15.

```
#define NMAX 10
...
double CALC_DETERMINANTE (double MATRIZ[][NMAX], unsigned int N)
{
  int COL_AUX, NC, NL, UE = N-1; double ELEMENTO;
  if (N == 1)  return MATRIZ[0][0];         /* condição de paragem */
  else
  {
    COL_AUX = UE;       /* procurar coluna com último elemento ≠ 0 */
    while  ( (COL_AUX >= 0) && (MATRIZ[UE][COL_AUX] == 0) )
      COL_AUX--;
    if (COL_AUX >= 0)                     /* se existir tal coluna */
    {
      if (COL_AUX != UE)             /* se não for a última coluna */
        for (NL = 0; NL < N; NL++)           /* trocar as colunas */
        {
          ELEMENTO = MATRIZ[NL][UE];
          MATRIZ[NL][UE] = MATRIZ[NL][COL_AUX];
          MATRIZ[NL][COL_AUX] = -ELEMENTO;
        }
 /* dividir a coluna N-1 pelo último elemento pondo-o em evidência */
        for (NL = 0; NL < UE; NL++)
          MATRIZ[NL][UE] /= MATRIZ[UE][UE];
    /* subtrair todas as colunas menos a última pela última coluna */
       /* multiplicada pelo último elemento da coluna a processar */
        for (NC = 0; NC < UE; NC++)
          for (NL = 0; NL < UE; NL++)
            MATRIZ[NL][NC] -= MATRIZ[NL][UE] * MATRIZ[UE][NC];
            /* invocação recursiva para a matriz de dimensão N-1 */
        return  MATRIZ[UE][UE] * CALC_DETERMINANTE (MATRIZ, N-1);
    }
    else return 0;
  }
}
```

Figura 4.17 - Função recursiva que calcula o determinante de uma matriz quadrada.

$$\begin{bmatrix} 3.0 & 4.0 & 2.0 & 5.0 \\ 4.0 & 2.0 & 2.0 & 1.0 \\ 1.0 & 3.0 & 2.0 & 2.0 \\ 0.0 & 5.0 & 3.0 & 2.0 \end{bmatrix} \Rightarrow \begin{bmatrix} 3.0 & -8.5 & -5.5 & 0.0 \\ 4.0 & -0.5 & 0.5 & 0.0 \\ 1.0 & -2.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 2.0 \end{bmatrix}$$

$$\begin{bmatrix} -2.5 & 2.5 & 0.0 & 0.0 \\ 4.5 & -1.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 2.0 \end{bmatrix} \Rightarrow \begin{bmatrix} 5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 2.0 \end{bmatrix}$$

Figura 4.18 - Execução da função recursiva que calcula o determinante de uma matriz quadrada.

4.8 Torres de Hanói

Vamos agora apresentar um problema cuja única solução conhecida é a solução recursiva. As Torres de Hanói é o puzzle que se apresenta na primeira linha da Figura 4.19. Temos um conjunto de discos todos de tamanho diferente enfiados na Torre A. Pretende-se mudá-los para a Torre B, mas, só se pode mudar um disco de cada vez, só se pode mudar o disco de cima e nunca se pode colocar um disco sobre outro mais pequeno. Para efectuarmos a mudança podemos usar a Torre C como torre auxiliar.

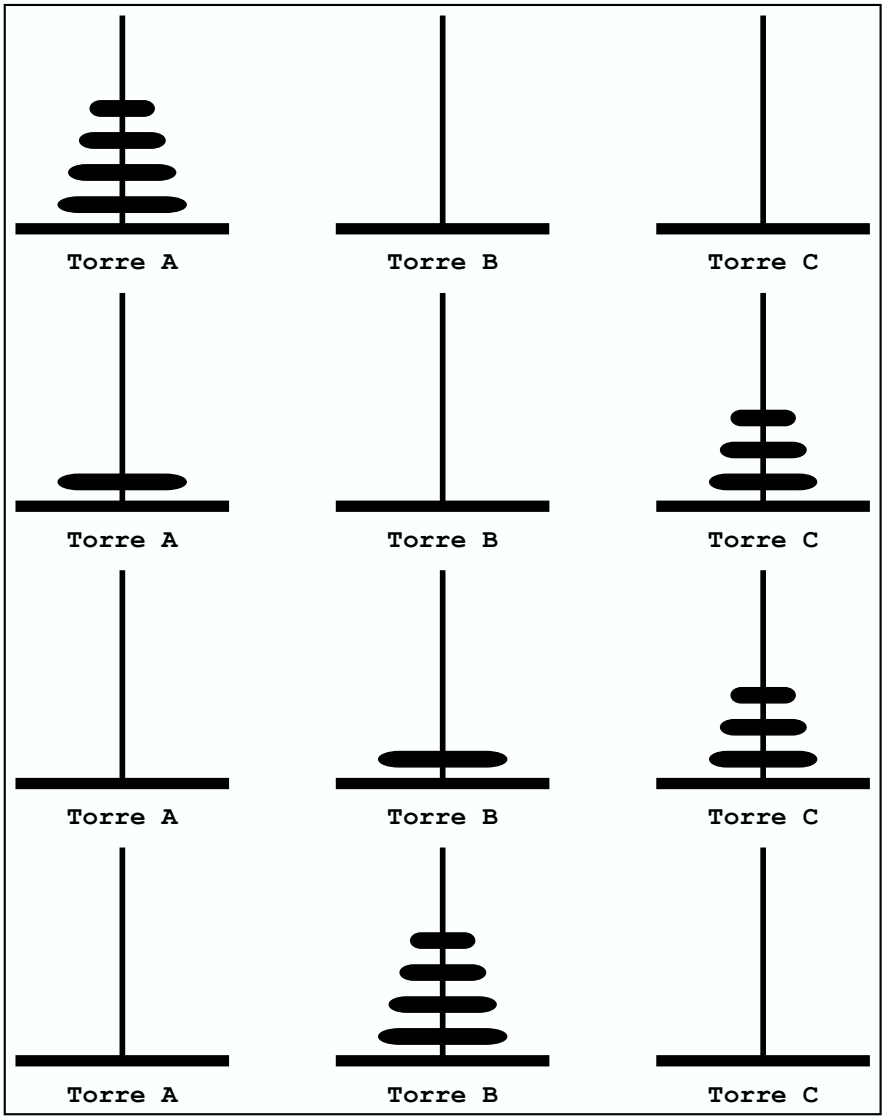

Figura 4.19 - Torres de Hanói.

A solução para este problema é trivial caso o número de discos seja um. Nesse caso basta mudá-lo da Torre A para a Torre B. No entanto se tivermos mais do que um disco a solução não é óbvia e até é bastante dispendiosa à medida que o número de discos aumenta. A solução que se apresenta na Figura 4.20 é a solução recursiva que resolve o puzzle para o caso de N discos colocados na Torre A. Consiste em diminuir a complexidade do problema até à situação em que temos apenas um disco e para o qual conhecemos a solução.

Assim para o caso apresentado na Figura 4.19 em que temos quatro discos na Torre A, a solução passa por movimentar os três discos de cima da Torre A para a Torre C, depois mudar o quarto disco da Torre A para a Torre B e finalmente mudar os três discos que se encontram na Torre C para a Torre B.

```
nome: Torres de Hanoi (N, TORRE_A, TORRE_B, TORRE_C)      Procedimento
begin
  if (N = 1)
  then Mover o disco da TORRE_A para a TORRE_B
  else begin
         Torres de Hanoi (N-1, TORRE_A, TORRE_C, TORRE_B);
         Torres de Hanoi (1, TORRE_A, TORRE_B, TORRE_C);
         Torres de Hanoi (N-1, TORRE_C, TORRE_B, TORRE_A);
  end
end
```

Figura 4.20 - Algoritmo das Torres de Hanói.

Obviamente que esta solução funciona se for aplicada recursivamente, porque, resolver o problema de mudar os três primeiros discos da Torre A para a Torre C, resolve-se da mesma forma, só que agora a torre de chegada é a Torre C, a torre auxiliar é a Torre B e o número de discos a mudar é menos um do que no problema inicial. A solução passa por mudar os dois discos de cima da Torre A para a Torre B, depois mudar o terceiro disco da Torre A para a Torre C e finalmente mudar os dois discos que se encontram na Torre B para a Torre C. Assim estamos perante o problema de mudar dois discos da Torre A para a Torre B, o que implica mudar o primeiro disco da Torre A para a Torre C, o segundo disco da Torre A para a Torre B e finalmente o primeiro disco da Torre C para a Torre B.

Após termos mudado os três primeiros discos da Torre A para a Torre C e termos atingido a situação apresentada na terceira linha da Figura 4.19, então para mudar os três discos que se encontram na Torre C para a Torre B aplica-se de novo a solução recursiva, mas agora a torre de partida é a Torre C, a torre de chegada é a Torre B e a torre auxiliar é a Torre A.

A Figura 4.21 e a Figura 4.22 apresentam o programa que simula as Torres de Hanói. Para implementar as três torres utilizam-se três agregados, sendo os discos representados por números inteiros de 1 a N. O programa principal lê do teclado o número de discos, valida o seu valor, constrói a situação inicial, imprime-a no monitor e depois invoca a função para simular a mudança dos discos.

A função INICIALIZAR coloca os N discos na Torre A e nenhum disco na Torre B e na Torre C. A função MUDAR_DISCOS implementa o algoritmo recursivo e após cada mudança de um disco invoca a função IMPRIMIR. A função IMPRIMIR vai imprimir no monitor o estado das torres, no início e após cada mudança de um disco. A função não tem parâmetros, porque utiliza variáveis globais. Uma vez que o algoritmo é recursivo, em cada nova invocação as torres vão mudando de posição, pelo que, a única forma de poder imprimir o estado das torres A, B e C, é usando variáveis globais para as torres e para os contadores do número de discos armazenados em cada torre. Para declarar variáveis globais, elas têm de ser declaradas antes da função **main** com o qualificativo **static**.

```
#include <stdio.h>

#define D_MAX 10
                                          /* agregados para as torres */
static int TORREA[D_MAX], TORREB[D_MAX], TORREC[D_MAX];
static int NDA, NDB, NDC;         /* número de discos de cada torre */

void INICIALIZAR (int, int [], int *, int [], int *, int [], int *);
void IMPRIMIR (void);
void MUDAR_DISCOS (int, int [], int *, int [], int *, int [], int *);

int main (void)
{
  int NDISCOS;            /* número de discos a colocar na Torre A */

  do
  {
    printf ("Numero de discos = "); scanf ("%d", &NDISCOS);
  } while ( (NDISCOS <= 0) || (NDISCOS > DISCOS_MAX) );

  INICIALIZAR (NDISCOS, TORREA, &NDA, TORREB, &NDB, TORREC, &NDC);
  printf ("---------------------------------\n");
  printf ("|        Torres de Hanoi        |\n");
  printf ("|     Numero de discos = %2d     |\n", NDISCOS);
  printf ("---------------------------------\n");
  printf ("|  TORRE A   TORRE B   TORRE C  |\n");
  printf ("---------------------------------\n");
  IMPRIMIR ();

  MUDAR_DISCOS (NDISCOS, TORREA, &NDA, TORREB, &NDB, TORREC, &NDC);
  return 0;
}

void INICIALIZAR (int ND, int TA[], int *NDA, int TB[], int *NDB,\
                  int TC[], int *NDC);
{
  int I;

  for (I = 0; I < D_MAX; I++)
  {
    TA[I] = 0;  TB[I] = 0;  TC[I] = 0;      /* limpar os agregados */
  }
  for (I = 0; I < ND; I++)  TA[I] = ND - I;   /* discos na Torre A */

  *NDA = ND;  *NDB = 0;  *NDC = 0;
}

void IMPRIMIR (void)
{
  int I, CMAX = NDA;
  if (NDB > CMAX) CMAX = NDB;
  if (NDC > CMAX) CMAX = NDC;
  for (I = CMAX; I > 0; I--)
  {
    if (NDA >= I) printf ("%10d", TORREA[I-1]);
    else printf ("%10c", ' ');
    if (NDB >= I) printf ("%10d", TORREB[I-1]);
    else printf ("%10c", ' ');
    if (NDC >= I) printf ("%10d", TORREC[I-1]);
    else printf ("%10c", ' ');
    printf ("\n");
  }
  printf ("---------------------------------\n"); scanf ("%*c");
}
```

Figura 4.21 - Programa das Torres de Hanói (1ª parte).

```
void MUDAR_DISCOS (int ND, int TA[], int *NDA, int TB[], int *NDB,\
                   int TC[], int *NDC);
{
  if (ND == 1)                                  /* condição de paragem */
  {                      /* mudar o disco da Torre A para a Torre B */
    (*NDB)++;  TB[*NDB-1] = TA[*NDA-1];  (*NDA)--;
    IMPRIMIR ();
  }
  else
  {
         /* mudar os N-1 discos de cima da Torre A para a Torre C */
    MUDAR_DISCOS (ND-1, TA, NDA, TC, NDC, TB, NDB);
                 /* mudar o último disco da Torre A para a Torre B */
    (*NDB)++;  TB[*NDB-1] = TA[*NDA-1];  (*NDA)--;
    IMPRIMIR ();
                  /* mudar os N-1 discos da Torre C para a Torre B */
    MUDAR_DISCOS (ND-1, TC, NDC, TB, NDB, TA, NDA);
  }
}
```

Figura 4.22 - Programa das Torres de Hanói (2ª parte).

O número de movimentos necessários para mudar N discos é igual a $2^N - 1$. Pelo que, o número de invocações da função recursiva aumenta exponencialmente com o aumento do número de discos. Por isso, não se deve invocar o programa para um número de discos maior do que 10. A Figura 4.23 apresenta a execução do programa para três discos.

```
---------------------------------
|        Torres de Hanoi        |
|     Numero de discos =  3     |
---------------------------------
|  TORRE A   TORRE B   TORRE C  |
---------------------------------
         1
         2
         3
---------------------------------
         2
         3         1
---------------------------------
         3         1         2
---------------------------------
                             1
         3                   2
---------------------------------
                             1
                   3         2
---------------------------------
         1         3         2
---------------------------------
                   2
         1         3
---------------------------------
                   1
                   2
                   3
---------------------------------
```

Figura 4.23 - Execução do programa Torres de Hanói para 3 discos.

4.9 Questões sobre a eficiência das soluções recursivas

Alguns algoritmos recursivos desencadeiam um número arbitrariamente grande de invocações sucessivas da função, pelo que, nestes casos temos uma ineficiência acrescida associada à invocação de funções. Por outro lado, existem algoritmos recursivos que são computacionalmente ineficientes, porque calculam certos valores repetidamente devido às múltiplas invocações recursivas. No caso dos algoritmos em que a invocação recursiva explode rapidamente, existe ainda o perigo de a memória de tipo pilha esgotar e do programa terminar a sua execução abruptamente. Portanto, o programador deve ter sempre em conta estas ineficiências e limitações, antes de optar por uma solução recursiva em detrimento de uma solução iterativa, caso ela exista.

Os algoritmos recursivos são apropriados para resolver problemas que são normalmente definidos de forma recursiva, ou seja, problemas que são por natureza recursivos. Por vezes, existem problemas que têm soluções recursivas que são simples, concisas, elegantes e para os quais é difícil esboçar soluções iterativas com a mesma simplicidade e clareza. Mas, tal como vimos nos exemplos apresentados, alguns algoritmos recursivos são menos eficientes que os algoritmos iterativos equivalentes. Pelo que, a verdadeira importância da recursividade consiste em resolver esses problemas para os quais não existem soluções iterativas simples.

4.10 Exercícios

1. Pretende-se escrever um programa que imprima no monitor, uma tabela em que se compara os valores da função co-seno calculada pela expansão em série de Taylor para 5, 10 e 15 termos e pela função matemática **cos**. Os valores inicial e final da tabela, bem como o número de elementos da tabela são lidos do teclado. O termo geral da expansão em série de Taylor da função co-seno é dado pela seguinte expressão à esquerda, sendo apresentada à direita a sua expansão para os primeiros cinco termos da série.

$$\text{coseno}(x,N) = \sum_{n=0}^{N-1} (-1)^n \times \frac{x^{2n}}{(2n)!} \qquad \text{coseno}(x,5) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \ldots$$

Desenhe a tabela com o formato que a seguir se apresenta.

```
-------------------------------------------------------------------------
|    x    |  coseno(x,5) | coseno(x,10) | coseno(x,15) |    cos(x)    |
-------------------------------------------------------------------------
| ##.#### |  #.######### |  #.######### |  #.######### | #.######### |
-------------------------------------------------------------------------
                                    ...
-------------------------------------------------------------------------
| ##.#### |  #.######### |  #.######### |  #.######### | #.######### |
-------------------------------------------------------------------------
```

Faça duas versões do programa. Na primeira versão, implemente a expansão da série de Taylor da função co-seno com uma função iterativa, e na segunda versão com uma função recursiva.

2. Pretende-se escrever um programa que escreva no monitor as permutações da cadeia de caracteres composta pelos caracteres numéricos de '0' a '9'. Altere o programa para escrever apenas as permutações que são compostas alternadamente por caracteres numéricos pares e ímpares.

3. Pretende-se escrever um programa que lê do teclado dois números inteiros positivos, que calcula e imprime no monitor o seu máximo divisor comum. O máximo divisor comum de dois números inteiros positivos pode ser calculado de forma repetitiva, utilizando para o efeito o método de Euclides, cujo algoritmo é dado pela seguinte expressão.

$$\text{mdc}(m,n) = \begin{cases} m, \text{se } n = 0 \\ \text{mdc}(n, \text{mod}(m,n)), \text{se } n \neq 0 \end{cases}$$

Faça duas versões do programa. Na primeira versão, implemente o cálculo do máximo divisor comum com uma função iterativa, e na segunda versão com uma função recursiva.

4. Pretende-se escrever um programa que lê do teclado dois números inteiros positivos, que calcula e imprime no monitor o valor da função de Ackermann, que se apresenta na seguinte expressão.

$$\text{Ackermann}(m,n) = \begin{cases} n+1, \text{se } m = 0 \\ \text{Ackermann}(m-1,1), \text{se } n = 0 \\ \text{Ackermann}(m-1, \text{Ackermann}(m,n-1)), \text{com } n > 0 \text{ e } m > 0 \end{cases}$$

Faça duas versões do programa. Na primeira versão, implemente o cálculo da função de Ackermann com uma função recursiva, e na segunda versão com uma função iterativa dinâmica, usando um agregado bidimensional.

4.11 Leituras recomendadas

- 3º capítulo do livro "Data Structures, Algorithms and Software Principles in C", de Thomas A. Standish, da editora Addison-Wesley Publishing Company, 1995.

Capítulo 5

MEMÓRIAS

Sumário

Neste capítulo começamos por introduzir o paradigma da programação modular e a construção de módulos na linguagem C, apresentando as suas características e a necessidade de criação de módulos genéricos. A título de exemplo, desenvolvemos um exemplo de um módulo abstracto com múltipla instanciação para operações sobre números complexos.

Seguidamente mostramos a organização da Memória de Acesso Aleatório (*RAM*), da Memória Fila (*Queue*/*FIFO*), da Memória Pilha (*Stack*/*LIFO*) e da Memória Associativa (*CAM*) e os seus ficheiros de interface.

Depois de descrevermos as particularidades e limitações dos tipos de implementação de memórias, fazemos uma abordagem às estruturas de dados que servem de suporte à implementação estática e semiestática da Memória de Acesso Aleatório, e à implementação estática, semiestática e dinâmica da Memória Fila, da Memória Pilha e da Memória Associativa.

Finalmente, apresentamos as funções da biblioteca de execução ANSI *stdlib* que permitem a atribuição e libertação de memória, durante a execução do programa.

5.1 Programação modular

O paradigma de programação procedimental é enunciado da seguinte forma "decide os procedimentos que precisas e usa os melhores algoritmos possíveis". Consiste na decomposição hierárquica da solução do problema, também designada de decomposição do topo para a base (*Top-Down Decomposition*), ou seja, implementa a estratégia do dividir para conquistar. A única maneira de lidar com um problema complexo consiste em decompô-lo num conjunto de problemas mais pequenos, cada um deles de resolução mais fácil que o problema original. Outra técnica que permite decompor a complexidade de um problema, em problemas mais pequenos que têm a particularidade de serem exactamente do mesmo tipo do problema original, é a construção de soluções recursivas.

A estratégia da decomposição hierárquica é implementada através da definição de novas operações no âmbito da linguagem, ou seja, através da criação de subprogramas. No caso da linguagem Pascal temos dois tipos de subprogramas, que são o procedimento e a função. No caso da linguagem C, temos apenas a função generalizada que é um tipo de subprograma que combina as características apresentadas pelo procedimento e pela função.

O subprograma pode ser visto como uma caixa preta, que recebe informação à entrada e que produz informação à saída, escondendo no entanto os detalhes da implementação da operação, ou seja, as acções que dentro do subprograma transformam a informação de entrada na informação de saída. Estas acções são invisíveis externamente e, portanto, não originam qualquer interacção com o exterior. Este conjunto de acções detalhadas, que representam a solução do problema, designa-se por algoritmo. Ao encapsulamento do algoritmo dentro de um subprograma, designa-se por abstracção procedimental.

A abstracção procedimental permite fazer a separação entre o objectivo de um subprograma da sua implementação. Após a definição do subprograma, a nova operação é identificada por um nome e por uma lista de parâmetros de comunicação com o exterior. Do ponto de vista operacional, tudo o que o programador precisa de saber para utilizar o subprograma é o seu nome e a sua interface com o exterior, que deve estar bem documentada. O programador não precisa de conhecer a sua implementação, ou seja, o algoritmo utilizado. Daí que, as linguagens de alto nível providenciam facilidades para a passagem de informação, de e para os subprogramas, de maneira a estabelecer o fluxo dos dados através das subprogramas e por conseguinte, ao longo do programa.

Uma questão essencial a ter em conta durante o desenvolvimento de um subprograma, é que ele deve ser implementado de forma a ser, se possível, completamente autónomo do ambiente externo onde vai ser utilizado, tornando-o versátil. Assim assegura-se que ele pode ser testado separadamente e, mais importante, que pode ser reutilizado noutro contexto. É portanto, uma nova operação que estende a operacionalidade da linguagem. Com a criação de subprogramas autónomos é possível criar bibliotecas de subprogramas que poderão ser reutilizados mais tarde noutros programas.

Neste paradigma de programação, os subprogramas podem ser vistos como blocos que podem ser interligados para construir programas mais complexos, fazendo uma montagem do programa da base para o topo (*Bottom-Up Assembly*). Permitindo assim, validar a solução do problema de uma maneira controlada e integrar de um modo progressivo os diferentes subprogramas, avaliando possíveis soluções alternativas através de diferentes arranjos de subprogramas.

Assim, se potenciarmos ao máximo a reutilização de subprogramas, o esforço que é necessário despender para criar novos programas é menor do que implementá-los de raiz. No entanto, a autonomia de um subprograma está limitada ao facto de a estrutura de dados que o seu algoritmo manipula ser exterior ao subprograma. Pelo que, a utilização do subprograma tem que ter sempre em conta não só a lista de parâmetros de comunicação com o exterior, mas também a própria estrutura de dados para o qual foi desenvolvido. O que implica, que se um programa necessitar de uma estrutura de dados que mesmo sendo semelhante é implementada de forma diferente, então o subprograma tem de ser modificado antes de ser reutilizado. Assim a reutilização do subprograma com um mínimo de esforço pode estar comprometida.

Por outro lado, ao longo dos anos, a ênfase na implementação do *software* dirigiu-se em direcção à organização de estruturas de dados cada vez mais complexas. Para estruturas de dados complexas não faz sentido implementar subprogramas de uma forma isolada, mas sim providenciar um conjunto de subprogramas que as manipulam e que permitem que as estruturas de dados sejam encaradas como um novo tipo de dados da linguagem. Um tipo de dados permite a declaração de variáveis desse tipo e tem associado um conjunto de operações permitidas sobre essas variáveis.

Este paradigma de programação, a que se dá o nome de programação modular, é enunciado da seguinte forma "decide os módulos que precisas e decompõe o programa para que as estruturas de dados sejam encapsuladas nos módulos". Nesta filosofia de programação já não escondemos apenas os algoritmos, mas também as estruturas de dados que são processadas pelos algoritmos. Ou seja, à abstracção procedimental acrescenta-se também a abstracção das estruturas de dados. Com a abstracção das estruturas de dados o programador concentra a sua atenção na operacionalidade das operações que processam as estruturas de dados, ou seja, na acção das operações sobre as estruturas de dados, em vez de se preocupar com os detalhes da implementação das operações.

Portanto, o paradigma de programação modular consiste na decomposição do programa em estruturas autónomas interactivas, onde existe uma separação clara entre a definição do módulo, ou seja, a sua interface com o exterior e a respectiva implementação do módulo. Do ponto de vista operacional, tudo o que o programador precisa de saber para utilizar o módulo é o seu nome e a sua interface com o exterior, que deve estar bem documentada. O programador não precisa de conhecer a sua implementação, ou seja, a organização da estrutura de dados interna do módulo e os algoritmos utilizados pelas operações.

Esta filosofia de programação permite a criação de estruturas de dados abstractas, uma vez que os detalhes da sua implementação estão escondidos, ou seja encapsulados no módulo. Permite também a sua protecção uma vez que não estão acessíveis a partir do exterior a não ser através das operações de processamento disponibilizadas pelo módulo. Permite ainda, a criação de operações virtuais, uma vez que se podem experimentar vários algoritmos alternativos de manipulação das estruturas de dados, através da criação de módulos de implementação alternativos, para a mesma interface.

Os módulos são assim entidades mais autónomas do que os subprogramas e por conseguinte mais reutilizáveis. São blocos construtivos mais poderosos, tanto mais poderosos quanto a sua estrutura de dados for reconfigurável em função das necessidades da aplicação, ou seja, quanto mais abstracta for a estrutura de dados interna do módulo. A abstracção tem a vantagem de esconder completamente a implementação da estrutura de dados, protegendo-a assim de operações que não são fornecidas pelo módulo.

Logo, uma questão essencial a ter em conta durante o desenvolvimento de um módulo, é que ele deve ser implementado da forma mais abstracta possível, tornando-o assim genérico. Um módulo genérico é um módulo cuja estrutura de dados pode ser do tipo de dados determinado pelo programador, em função da aplicação onde o módulo vai ser utilizado, sem que ele tenha que modificar a implementação do módulo. Um módulo genérico assegura uma maior reutilização.

Resumindo, os módulos apresentam quatro características muito importantes: permitem agrupar estruturas de dados e as operações de processamento associadas; apresentam interfaces bem definidas para os seus utilizadores, onde são reveladas as operações de processamento disponíveis; escondem os detalhes da implementação das mesmas focando a atenção do programador na sua funcionalidade; e podem ser compilados separadamente.

5.2 Módulos na linguagem C

Antes de analisarmos a composição de um módulo, devemos ter em consideração que devemos então criar, sempre que possível, um módulo genérico, de modo a que seja o mais reutilizável possível. É preciso também ter em atenção, que por vezes o programador não tem acesso à implementação do módulo, mas apenas à sua interface. Como é então possível criar um módulo genérico? Existem duas maneiras.

Uma consiste em criar um módulo de dados abstractos, ou seja, um módulo em que a estrutura de dados fica indefinida recorrendo à declaração de ponteiros de tipo ***void***. Aquando da utilização de um módulo abstracto é preciso concretizá-lo para o tipo de dados pretendido através da disponibilização de uma função de criação do módulo, que é responsável pela caracterização do tamanho em *bytes* do elemento básico da estrutura de dados. Uma vez que a biblioteca de execução ANSI ***string*** providencia funções de cópia de memória, sem ser atribuído qualquer interpretação ao conteúdo dos *bytes* copiados, é então possível manipular estruturas de dados sem que o código C se tenha que preocupar com o tipo de dados que está a processar. A implementação abstracta tem a vantagem de esconder completamente a representação da informação, protegendo a estrutura de dados interna do módulo. Se o programador não conhecer os pormenores da implementação da estrutura de dados, então não pode, nomeadamente, desenvolver funções que actuam sobre ela.

A alternativa à criação de um módulo abstracto, consiste na criação de um módulo concreto, mas, em que o tipo de dados dos elementos da estrutura de dados é definido à parte num ficheiro de interface, que vamos designar por elemento.h, e que depois é incluído no ficheiro de interface do módulo. Em certas aplicações, este ficheiro para além da definição do tipo de dados dos elementos, pode também definir as constantes que parametrizam a dimensão das estruturas de dados. Assim o utilizador do módulo, pode concretizar o módulo para uma estrutura de dados que corresponda às suas necessidades, quer em termos de dimensão quer em termos do tipo dos elementos, sem ter a necessidade de reprogramar o ficheiro de implementação do módulo. Em relação à criação de um módulo abstracto, esta solução exige a recompilação do módulo, sempre que este ficheiro de interface é modificado.

A Figura 5.1 apresenta um exemplo do ficheiro de interface do elemento constituinte de uma estrutura de dados, considerando que os elementos são do tipo TIPO_ELEMENTO, bem como da definição da constante N_ELEMENTOS que dimensiona a estrutura de dados do módulo.

```
 /*********** Interface da Estrutura de Dados do Módulo ***********/
                          /* Nome: elemento.h */

/* Definição da dimensão da estrutura de dados e do tipo de dados
dos seus elementos. Este ficheiro deve ser modificado para adequar a
definição a cada implementação específica.                          */

#ifndef _ELEMENTO
#define _ELEMENTO

 /****** Constantes de Parametrização das Estruturas de Dados ******/

#define  N_ELEMENTOS  100                    /* número de elementos */

 /************* Definição do Tipo de Dados do Elemento *************/

typedef  ...  TIPO_ELEMENTO;          /* tipo de dados dos elementos */

#endif
```

Figura 5.1 - Ficheiro de interface do elemento constituinte da estrutura de dados.

Os módulos na linguagem C são compostos por dois ficheiros. O ficheiro de interface, que se apresenta na Figura 5.2, tem a extensão **.h**. O ficheiro de implementação, que se apresenta na Figura 5.3, tem a extensão **.c**.

```
 /********************** Interface do Módulo ********************/
                          /* Nome: modulo.h */

#ifndef _MODULO
#define _MODULO

 /** Inclusão do Ficheiro de Interface do Tipo de Dados do Módulo **/

#include "elemento.h" /* caracterização do tipo elemento do módulo */

 /******************** Definição de Constantes *******************/

#define OK  0      /* operação realizada com sucesso */
...
 /************* Alusão às Funções Exportadas pelo Módulo *************/

void INICIALIZAR (TIPO_ELEMENTO [], int);
...
#endif
```

Figura 5.2 - Ficheiro de interface do módulo.

O ficheiro de interface declara as entidades do módulo que são visíveis no exterior e que são utilizáveis pelos utilizadores. Consiste na definição de constantes que representam identificadores de códigos de erro devolvidos pelas funções do módulo e que servem para assinalar o estado de execução das funções. Consiste também nas alusões, ou protótipos, das funções exportadas pelo módulo. Apenas estas funções podem ser utilizadas e é através delas que as aplicações manipulam as estruturas de dados internas do módulo.

Por vezes um módulo é incluído noutro módulo, que por sua vez é incluído noutro módulo e assim sucessivamente até que um ou mais destes módulos são incluídos nos ficheiros fonte da aplicação. De maneira a evitar uma possível múltipla inclusão de um módulo na aplicação, ele deve ser incluído condicionalmente. Para tal, existe a directiva do pré-processador #**ifndef** _MODULO #**define** _MODULO **… endif** que assegura que o módulo, cujo nome é MODULO, é incluído apenas uma vez.

```
 /******************** Implementação do Módulo ********************/
                        /* Nome: modulo.c */
 /************ Inclusão das Bibliotecas da Linguagem C ************/
#include <stdlib.h>
...
 /********** Inclusão do Ficheiro de Interface do Módulo **********/
#include "modulo.h"
 /***** Declaração das Estruturas de Dados Internas do Módulo *****/
static TIPO_ELEMENTO ESTRUTURA_DADOS_MODULO[N_ELEMENTOS];
...
 /******************** Implementação das Funções ********************/
static void TROCA (int *, int *)        /* função interna do módulo */
{ ... }
                                    /* função exportada pelo módulo */
void INICIALIZAR (TIPO_ELEMENTO [], int)
{ ... }
...
```

Figura 5.3 - Ficheiro de implementação do módulo.

O ficheiro de implementação implementa a funcionalidade do módulo. Começa por incluir as bibliotecas de execução ANSI da linguagem C, que são necessárias e o ficheiro de interface do módulo, de maneira a incluir as constantes que representam os códigos de erro devolvidos pelas funções e do tipo de dados dos elementos da estrutura de dados. Declara as estruturas de dados internas do módulo, que são o suporte ao armazenamento da informação interna do módulo. A declaração das estruturas de dados no ficheiro de implementação, com o qualificativo **static,** torna-as globais no ficheiro de implementação mas invisíveis no exterior. Pelo que, podem ser manipuladas pelas funções do módulo sem terem de ser passadas pela lista de parâmetros, mas, estão protegidas de serem manipuladas através de funções exteriores ao módulo. No caso das estruturas de dados do módulo serem estáticas ou semiestáticas elas são parametrizadas por constantes que definem a sua dimensão. Essas constantes encontram-se definidas no ficheiro de interface do tipo de dados dos elementos do módulo, ou seja, no ficheiro elemento.h.

Após a definição das estruturas de dados, o ficheiro de implementação faz a definição das funções exportadas pelo módulo, ou seja, das funções que foram declaradas no ficheiro de interface. Por vezes, para implementar uma função é necessário recorrer a funções auxiliares de maneira a estruturar melhor a sua funcionalidade. Se estas funções auxiliares forem partilhadas por várias funções, nesse caso devem ser aludidas logo a seguir à declaração das estruturas de dados. Estas funções auxiliares, sendo funções internas do módulo, devem ser declaradas com o qualificativo **static**. Desta forma são invisíveis para o exterior, pelo que, os seus nomes podem ser utilizados noutros módulos para implementar outras funções internas a esses módulos.

Os módulos são compilados separadamente, com a opção **–c**, para criar o ficheiro objecto, que tem a extensão **.o**, tal como se mostra na linha seguinte.

cc –c *nome_do_módulo.**c***

Para que uma aplicação utilize um módulo, o seu ficheiro de interface deve ser incluído nos ficheiros da aplicação, tal como se apresenta na Figura 5.4.

```
 /********************* Ficheiro da Aplicação *********************/
 /************ Inclusão das Bibliotecas da Linguagem C ************/
#include <stdio.h>
...
 /********** Inclusão do Ficheiro de Interface do Módulo **********/
#include "modulo.h"
 /******************** Definição de Constantes ********************/
#define  NP  10
...
/************ Alusão às Funções definidas neste Ficheiro ************/
...

int main (void)
{
  ...       /* invocação das funções do módulo e de outras funções */
  return 0;
}
 /******* Implementação das Funções definidas neste Ficheiro *******/
...
```

Figura 5.4 - Utilização de um módulo.

O ficheiro objecto é depois acrescentado no comando de compilação do programa que utiliza o módulo, tal como se mostra na linha seguinte.

cc *nome_do_ficheiro.**c*** *nome_do_módulo.**o*** **–o** *nome_do_ficheiro_executável*

5.3 Exemplo de um módulo

Vamos agora mostrar a implementação de um módulo que implementa a criação de números complexos e de operações sobre números complexos. A Figura 5.5 apresenta o seu ficheiro de interface. Como se pode ver, o programador apenas sabe que o módulo define um tipo de dados **PtComplexo** que é um ponteiro para uma estrutura, não sabendo no entanto, se a estrutura que armazena o número complexo utiliza uma representação em parte real e parte imaginária ou se utiliza a representação em coordenadas polares. Esta declaração prende-se com a necessidade do programador ter que declarar mais do que um número complexo na aplicação que vai desenvolver, pelo que, sem a exportação deste tipo de dados o programador não o poderia fazer. Portanto, estamos perante um módulo abstracto com possibilidade de múltipla instanciação.

O módulo providencia um conjunto de funções necessárias para criar e operar números complexos. Assim temos uma operação de inicialização que cria um número complexo com um valor especificado, uma operação de leitura que lê o número complexo introduzido pelo teclado na forma R+jI e uma operação de escrita no monitor de um complexo na mesma forma. A operação de leitura só lê informação do teclado para o número complexo, caso ele tenha sido criado previamente e a operação de escrita só escreve o número complexo se ele existir. Qualquer uma das quatros operações básicas, adição, subtracção, multiplicação e divisão, cria um novo número complexo para armazenar o resultado da operação. Quando uma função cria um número complexo, faz a atribuição de memória para o seu armazenamento e devolve um ponteiro para a sua localização na memória. A partir daí o número complexo pode ser manipulado no programa através deste ponteiro.

O módulo também fornece uma função que permite libertar a memória atribuída para um número complexo, quando ele não é mais preciso, ou seja, uma função para apagar o número complexo. Para evitar divisões por zero, é preciso uma função que verifica se um número complexo é o complexo nulo. Existem ainda funções para extrair a parte real e a parte imaginária do número complexo.

```
      /* Ficheiro de interface do módulo de números complexos */
                        /* Nome: complexo.h */

#ifndef _COMPLEXO
#define _COMPLEXO

typedef  struct  complexo  *PtComplexo;

PtComplexo Inicializar_Complexo (double R, double I);
/* Função que cria e inicializa um complexo na forma R+jI */

void Ler_Complexo (PtComplexo PC);
/* Função que cria e lê do teclado um complexo na forma R+jI */

void Escrever_Complexo (PtComplexo PC);
/* Função que escreve no monitor um complexo na forma R+jI */

PtComplexo Somar_Complexos (PtComplexo PC1, PtComplexo PC2);
/* Função que soma dois números complexos */

PtComplexo Subtrair_Complexos (PtComplexo PC1, PtComplexo PC2);
/* Função que subtrai dois números complexos */

PtComplexo Multiplicar_Complexos (PtComplexo PC1, PtComplexo PC2);
/* Função que multiplica dois números complexos */

PtComplexo Dividir_Complexos (PtComplexo PC1, PtComplexo PC2);
/* Função que divide dois números complexos */

void Apagar_Complexo (PtComplexo *PC);
/* Função que apaga o número complexo e devolve o ponteiro a NULL */

int Complexo_Nulo (PtComplexo PC);
/* Função que testa se o número complexo é nulo (0+j0). Devolve 0 em
caso afirmativo e 1 em caso contrário */

double Parte_Real (PtComplexo PC);
/* Função que devolve a parte real de um número complexo */

double Parte_Imaginaria (PtComplexo PC);
/* Função que devolve a parte imaginária de um número complexo */

#endif
```

Figura 5.5 - Ficheiro de interface do módulo de números complexos.

A Figura 5.6 e a Figura 5.7 apresentam o ficheiro de implementação do módulo. A Figura 5.8 apresenta um exemplo da utilização do módulo para simular uma máquina de calcular de números complexos. O programa começa por criar dois números complexos, Comp1 e Comp2, com o valor nulo. Depois tem um funcionamento repetitivo, que consiste em escrever o menu de operações no monitor e executar a operação escolhida pelo utilizador. A máquina de calcular permite ler do teclado valores para os dois operadores complexos Comp1 e Comp2 e executar as quatro operações básicas aritméticas. A operação de divisão só é feita se o divisor que é Comp2 não for nulo. Quando qualquer uma das quatro operações aritméticas é realizada, o resultado é armazenado no número complexo Resultado. Este complexo é depois impresso no monitor e apagado. Após o utilizador visualizar o resultado da operação e premir uma tecla, o monitor é limpo e o menu de operações é reescrito. Quando o utilizador escolher a opção de fim de execução do programa, os dois números complexos Comp1 e Comp2 são apagados.

```
    /* Ficheiro de implementação do módulo de números complexos */
                /* Nome: complexo.c - Primeira Parte */

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

#include "complexo.h"            /* Ficheiro de interface do módulo */

struct complexo
{
  double Real;  double Imag;
};

/* Função que cria e inicializa um complexo na forma R+jI */
PtComplexo Inicializar_Complexo (double R, double I)
{
  PtComplexo PC = (PtComplexo) malloc (sizeof (struct complexo));

  PC->Real = R;  PC->Imag = I;

  return PC;
}

/* Função que lê do teclado um complexo na forma R+jI */
void Ler_Complexo (PtComplexo PC)
{
  if (PC == NULL) return;

  printf ("Parte Real ");  scanf ("%lf", &PC->Real);
  printf ("Parte Imaginária ");  scanf ("%lf", &PC->Imag);
}

/* Função que escreve no monitor um complexo na forma R+jI */
void Escrever_Complexo (PtComplexo PC)
{
  if (PC != NULL) printf ("%f +j %f\n", PC->Real, PC->Imag);
}

/* Função que soma dois números complexos */
PtComplexo Somar_Complexos (PtComplexo PC1, PtComplexo PC2)
{
  PtComplexo PC = (PtComplexo) malloc (sizeof (struct complexo));

  PC->Real = PC1->Real + PC2->Real;
  PC->Imag = PC1->Imag + PC2->Imag;

  return PC;
}

/* Função que subtrai dois números complexos */
PtComplexo Subtrair_Complexos (PtComplexo PC1, PtComplexo PC2)
{
  PtComplexo PC = (PtComplexo) malloc (sizeof (struct complexo));

  PC->Real = PC1->Real - PC2->Real;
  PC->Imag = PC1->Imag - PC2->Imag;

  return PC;
}
```

Figura 5.6 - Ficheiro de implementação do módulo de números complexos (1º parte).

```
    /* Ficheiro de implementação do módulo de números complexos */
                /* Nome: complexo.c - Segunda Parte */

/* Função que multiplica dois números complexos */
PtComplexo Multiplicar_Complexos (PtComplexo PC1, PtComplexo PC2)
{
   PtComplexo PC = (PtComplexo) malloc (sizeof (struct complexo));

   PC->Real = PC1->Real * PC2->Real - PC1->Imag * PC2->Imag;
   PC->Imag = PC1->Real * PC2->Imag + PC1->Imag * PC2->Real;

   return PC;
}

/* Função que divide dois números complexos */
PtComplexo Dividir_Complexos (PtComplexo PC1, PtComplexo PC2)
{
   double QUO = PC2->Real * PC2->Real + PC2->Imag * PC2->Imag;

   PtComplexo PC = (PtComplexo) malloc (sizeof (struct complexo));

   PC->Real = (PC1->Real * PC2->Real + PC1->Imag * PC2->Imag) / QUO;
   PC->Imag = (PC1->Imag * PC2->Real - PC1->Real * PC2->Imag) / QUO;

   return PC;
}

/* Função que apaga o número complexo e devolve o ponteiro a NULL */
void Apagar_Complexo (PtComplexo *PC)
{
   PtComplexo TPC = *PC;

   if (TPC == NULL) return;

   *PC = NULL;

   free (TPC);
}

/* Função que devolve 0 se o número complexo é nulo (0+j0) ou 1 no
caso contrário */
int Complexo_Nulo (PtComplexo PC)
{
   if (PC->Real == 0 && PC->Imag == 0) return 0;
   else return 1;
}

/* Função que devolve a parte real de um número complexo */
double Parte_Real (PtComplexo PC)
{
   return PC->Real;
}

/* Função que devolve a parte imaginária de um número complexo */
double Parte_Imaginaria (PtComplexo PC)
{
   return PC->Imag;
}
```

Figura 5.7 - Ficheiro de implementação do módulo de números complexos (2º parte).

```
            /* Máquina de calcular de números complexos */
#include <stdio.h>
#include "complexo.h"

int main (void)
{
   PtComplexo Comp1, Comp2, Resultado;  int Opcao, Oper;

   Comp1 = Inicializar_Complexo (0.0, 0.0);
   Comp2 = Inicializar_Complexo (0.0, 0.0);
   do
   {
      system ("clear");
      printf ("\t1 - Ler o primeiro complexo\n");
      printf ("\t2 - Ler o segundo complexo\n");
      printf ("\t3 - Somar os complexos\n");
      printf ("\t4 - Subtrair os complexos\n");
      printf ("\t5 - Multiplicar os complexos\n");
      printf ("\t6 - Dividir os complexos\n");
      printf ("\t7 - Sair do programa\n");

      do
      {
        printf ("\n\tOpção -> "); scanf ("%d", &Opcao);
        scanf ("%*[^\n]"); scanf ("%*c");
      } while (Opcao<1 && Opcao>7);
      Oper = 0;
      switch (Opcao)
      {
        case 1 :  printf("\n\n"); Comp1 = Ler_Complexo(); break;
        case 2 :  printf("\n\n"); Comp2 = Ler_Complexo(); break;
        case 3 :  Resultado = Somar_Complexos(Comp1, Comp2);
                  printf ("Adição dos complexos -> ");
                  Escrever_Complexo (Resultado); Oper = 1; break;
        case 4 :  Resultado = Subtrair_Complexos(Comp1, Comp2);
                  printf ("Subtracção dos complexos -> ");
                  Escrever_Complexo (Resultado); Oper = 1; break;
        case 5 :  Resultado = Multiplicar_Complexos(Comp1, Comp2);
                  printf ("Multiplicação dos complexos -> ");
                  Escrever_Complexo (Resultado); Oper = 1; break;
        case 6 :  if ( Complexo_Nulo (Comp2) )
                  {
                    Resultado = Dividir_Complexos(Comp1, Comp2);
                    printf ("Divisão dos complexos -> ");
                    Escrever_Complexo (Resultado); Oper = 1;
                  }
                  else printf ("O divisor é o complexo nulo!!!\n");
                  break;
      }

      if (Opcao != 7)
      {
        printf ("\nPrima uma tecla para continuar\n"); scanf ("%*c");
      }
      if (Oper) Apagar_Complexo (&Resultado);

   } while (Opcao != 7);

   Apagar_Complexo (&Comp1);
   Apagar_Complexo (&Comp2);
   return 0;
}
```

Figura 5.8 - Máquina de calcular de números complexos.

5.4 Tipos de memórias e suas características

Vamos apresentar as características dos seguintes quatro tipos de memórias: a Memória de Acesso Aleatório (***RAM***); a Memória Fila (***Queue*/*FIFO***); a Memória Pilha (***Stack*/*LIFO***); e a Memória Associativa (***CAM***).

5.4.1 Memória de acesso aleatório (*RAM*)

Uma memória de acesso aleatório (***R**andom **A**ccess **M**emory*) é uma memória em que não existe nenhuma organização especial de acesso aos elementos armazenados, antes pelo contrário, é possível aceder em qualquer instante a qualquer posição da memória, indicando para o efeito o endereço, ou seja, o índice da posição de memória a que se pretende aceder. Estamos perante um acesso indexado, pelo que, uma memória de acesso aleatório só pode ser implementada por uma estrutura de dados que permita este tipo de acesso. A Figura 5.9 apresenta o ficheiro de interface de uma memória de acesso aleatório abstracta.

```
 /******************** Interface do Módulo RAM ********************/
#ifndef _RAM
#define _RAM

 /******************** Definição de Constantes ********************/

#define OK          0    /* operação realizada com sucesso */
#define NULL_PTR    1    /* ponteiro nulo */
#define NULL_SIZE   2    /* tamanho nulo */
#define NO_MEM      3    /* memória esgotada */
#define RAM_EMPTY   4    /* RAM vazia */
#define RAM_FULL    5    /* RAM cheia */
#define RAM_EXISTS  6    /* já foi instanciada uma RAM */
#define NO_RAM      7    /* ainda não foi instanciada qualquer RAM */

 /********************** Protótipos das Funções **********************/

int RAM_Create (unsigned int sz);
/* Concretiza a RAM para elementos de sz bytes. Valores de retorno:
OK, NULL_SIZE ou RAM_EXISTS. */

int RAM_Destroy (void);
/* Destrói a RAM. Valores de retorno: OK ou NO_RAM. */

int RAM_Write (void *pelemento, unsigned int pos);
/* Escreve o conteúdo do elemento apontado por pelemento na posição
pos da RAM. Valores de retorno: OK, NO_RAM, NULL_PTR, RAM_FULL ou
NO_MEM. */

int RAM_Read (void *pelemento, unsigned int pos);
/* Lê o conteúdo do elemento da posição pos da RAM para o elemento
apontado por pelemento. Valores de retorno: OK, NO_RAM, NULL_PTR ou
RAM_EMPTY. */

int RAM_Search (void *pelemento, int *pos);
/* Procura o primeiro elemento da RAM com conteúdo igual ao elemento
apontado por pelemento. Coloca em pos o índice do elemento
encontrado ou -1 caso não exista tal elemento. Valores de retorno:
OK, NO_RAM, NULL_PTR ou RAM_EMPTY. */

int RAM_Sort (void);
/* Ordena a RAM. Valores de retorno: OK, NO_RAM, ou RAM_EMPTY. */

#endif
```

Figura 5.9 - Ficheiro de interface da memória de acesso aleatório abstracta.

Sendo a característica principal da memória de acesso aleatório o acesso indexado, o posicionamento para a operação de leitura, que vamos designar por ***RAM_Read***, ou para a operação de escrita, que vamos designar por ***RAM_Write***, é feito através do índice do elemento de memória onde se pretende ler ou escrever. Para além das operações de leitura e de escrita, normalmente também é necessário procurar um elemento com um determinado valor, que vamos designar por ***RAM_Search*** e ordenar a memória, que vamos designar por ***RAM_Sort***. A ordenação da memória facilita a pesquisa de informação, permitindo por exemplo, utilizar a pesquisa binária em alternativa à pesquisa sequencial.

As operações de leitura e de escrita neste tipo de memória não afectam o seu tamanho. A memória de acesso aleatório tem sempre o tamanho que foi decidido na altura da sua definição, o que permite aceder a qualquer um dos seus elementos para ler a informação armazenada ou para escrever nova informação. No entanto, é conveniente não tentar ler informação de elementos onde ainda não foi feita qualquer operação de escrita. Logo, é aconselhável implementar uma política de escrita em elementos sucessivos da memória e manter um indicador de posição que indique qual o índice do último elemento da memória que contem informação útil.

As operações de criação ***RAM_Create*** e de destruição ***RAM_Destroy*** só existem para implementações abstractas e têm como função respectivamente, concretizar a memória para o tipo de elementos pretendidos e repor a situação inicial de memória ainda por concretizar de maneira a poder ser reutilizada, eventualmente para um novo tipo de dados.

5.4.2 Memória fila (*Queue/ FIFO*)

Uma memória fila, do inglês ***queue***, é uma memória em que só é possível processar a informação pela ordem de chegada. Daí que, também seja apelidada de memória do primeiro a chegar primeiro a sair, do inglês ***First In First Out***. A Figura 5.10 apresenta o ficheiro de interface de uma memória fila abstracta.

Numa fila, o posicionamento para a colocação de um novo elemento, que vamos designar por ***FIFO_In***, é a cauda da fila (***fifo tail***), e o posicionamento para a remoção de um elemento, que vamos designar por ***FIFO_Out***, é a cabeça da fila (***fifo head***). A colocação de um novo elemento consiste na adição de um novo elemento no fim da fila, ficando este novo elemento a ser a cauda da fila, e na escrita da informação nesse elemento. A remoção de um elemento consiste na leitura da informação armazenada no elemento que se encontra na cabeça da fila e da eliminação desse elemento da fila. Consequentemente, o elemento seguinte, caso o haja, passa a ser a cabeça da fila. Quando é retirado o último elemento, a fila fica vazia, sendo apenas possível efectuar a operação de colocação de elementos na fila. A situação inversa de fila cheia também pode acontecer para certo tipos de implementações, mais concretamente para implementações estáticas e semiestáticas.

Como o acesso à memória fila está limitado aos elementos posicionados nos extremos da memória, a fila mantém dois indicadores de posição para a cabeça e a cauda da fila. Ao contrário da memória de acesso aleatório, o tamanho da fila é dinâmico e depende do número de elementos colocados e do número de elementos retirados da fila.

As operações de criação ***FIFO_Create*** e de destruição ***FIFO_Destroy*** só existem para implementações abstractas e têm como função respectivamente, concretizar a fila para o tipo de elementos pretendidos e repor a situação inicial de fila ainda por concretizar de maneira a poder ser reutilizada, eventualmente para um novo tipo de dados.

```
/******************** Interface do Módulo FIFO ********************/
#ifndef _FIFO
#define _FIFO
 /******************** Definição de Constantes ********************/
#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NULL_SIZE   2   /* tamanho nulo */
#define NO_MEM      3   /* memória esgotada */
#define FIFO_EMPTY  4   /* fila vazia */
#define FIFO_FULL   5   /* fila cheia */
#define FIFO_EXISTS 6   /* já foi instanciada uma fila */
#define NO_FIFO     7   /* ainda não foi instanciada qualquer fila */
 /********************* Protótipos das Funções *********************/
int FIFO_Create (unsigned int sz);
/* Concretiza a fila para elementos de sz bytes. Valores de retorno:
OK, NULL_SIZE ou FIFO_EXISTS. */
int FIFO_Destroy (void);
/* Destrói a fila. Valores de retorno: OK ou NO_FIFO. */
int FIFO_In (void *pelemento);
/* Coloca o elemento apontado por pelemento na cauda da fila.
Valores de retorno: OK, NO_FIFO, NULL_PTR, FIFO_FULL ou NO_MEM. */
int FIFO_Out (void *pelemento);
/* Retira o elemento da cabeça da fila para o elemento apontado por
pelemento. Valores de retorno: OK, NO_FIFO, NULL_PTR ou FIFO_EMPTY.
*/
#endif
```

Figura 5.10 - Ficheiro de interface da memória fila abstracta.

5.4.3 Memória pilha (*Stack/ LIFO*)

Uma memória pilha, do inglês ***stack***, é uma memória em que só é possível processar a informação pela ordem inversa à ordem de chegada. Daí que, também seja apelidada de memória do último a chegar primeiro a sair, do inglês ***Last In First Out***. A Figura 5.11 apresenta o ficheiro de interface de uma memória pilha abstracta.

Numa pilha, o posicionamento para a colocação de um novo elemento, que se designa por ***STACK_Push***, e o posicionamento para a remoção de um elemento, que se designa por ***STACK_Pop***, é o topo da pilha (***top of the stack***). A colocação de um novo elemento consiste na adição de um novo elemento, em cima do topo da pilha e na escrita da informação nesse elemento, ficando este novo elemento a ser o topo da pilha. A remoção de um elemento consiste na leitura da informação armazenada no elemento que se encontra no topo da pilha e da eliminação desse elemento da pilha, ficando o elemento anterior, caso o haja, a ser o topo da pilha. Quando é retirado o último elemento, a pilha fica vazia, sendo apenas possível efectuar a operação de colocação de elementos na pilha. A situação inversa de pilha cheia também pode acontecer para certo tipos de implementações, mais concretamente para implementações estáticas e semiestáticas.

Como o acesso à memória pilha está limitado a este elemento posicionado no extremo da memória, a pilha mantém um indicador de posição para o topo da pilha. Tal como na fila, o tamanho da pilha também é dinâmico.

As operações de criação ***STACK_Create*** e de destruição ***STACK_Destroy*** só existem para implementações abstractas e têm como função respectivamente, concretizar a pilha para o tipo de elementos pretendidos e repor a situação inicial de pilha ainda por concretizar de maneira a poder ser reutilizada, eventualmente para um novo tipo de dados.

```
 /****************** Interface do Módulo STACK ******************/
#ifndef _STACK
#define _STACK
 /******************** Definição de Constantes ********************/
#define OK           0 /* operação realizada com sucesso */
#define NULL_PTR     1 /* ponteiro nulo */
#define NULL_SIZE    2 /* tamanho nulo */
#define NO_MEM       3 /* memória esgotada */
#define STACK_EMPTY  4 /* pilha vazia */
#define STACK_FULL   5 /* pilha cheia */
#define STACK_EXISTS 6 /* já foi instanciada uma pilha */
#define NO_STACK     7 /* ainda não foi instanciada qualquer pilha */
 /********************* Protótipos das Funções *********************/
int STACK_Create (unsigned int sz);
/* Concretiza a pilha para elementos de sz bytes. Valores de
retorno: OK, NULL_SIZE ou STACK_EXISTS. */
int STACK_Destroy (void);
/* Destrói a pilha. Valores de retorno: OK ou NO_STACK. */
int STACK_Push (void *pelemento);
/* Coloca o elemento apontado por pelemento no topo da pilha.
Valores de retorno: OK, NO_STACK, NULL_PTR, STACK_FULL ou NO_MEM. */
int STACK_Pop (void *pelemento);
/* Retira o elemento do topo da pilha para o elemento apontado por
pelemento. Valores de retorno: OK, NO_STACK, NULL_PTR ou
STACK_EMPTY. */
#endif
```

Figura 5.11 - Ficheiro de interface da memória pilha abstracta.

5.4.4 Memória associativa (*CAM*)

Uma memória associativa (***C****ontent* ***A****ccess* ***M****emory*) é uma memória em que o acesso aos seus elementos, é feita pelo conteúdo a que se pretende aceder, indicando para o efeito a chave do elemento a que se pretende aceder. Estamos perante um acesso por chave, pelo que, uma memória associativa só pode ser implementada por uma estrutura de dados em que a informação armazenada está sempre ordenada. A Figura 5.12 apresenta o ficheiro de interface de uma memória associativa abstracta.

Numa memória associativa, o posicionamento para a colocação de um novo elemento, que vamos designar por ***CAM_In***, e o posicionamento para a remoção de um elemento, que vamos designar por ***CAM_Out***, é feito através da chave do elemento que se pretende processar. A colocação de um novo elemento consiste na adição de um novo elemento na posição correspondente à chave de acesso do novo elemento, de maneira a manter a memória ordenada e na escrita da informação nesse elemento. A remoção de um elemento consiste na leitura da informação armazenada no primeiro elemento com a chave pretendida e da eliminação desse elemento da memória. Tal como nas memórias fila e pilha, o tamanho da memória associativa também é dinâmico.

Como a memória tem que estar sempre ordenada pela chave de pesquisa, as operações de colocação e de remoção de elementos podem implicar deslocamentos dos elementos da memória para permitir a colocação do elemento na memória ou para compensar a eliminação do elemento da memória. O facto da memória estar sempre ordenada, permite também implementar de forma eficiente operações de leitura de informação, que são normalmente muito importantes neste tipo de memória. Essas operações são a leitura do primeiro elemento da memória que tem uma determinada chave, que vamos designar por ***CAM_Read_First***, e a leitura de elementos sucessivos com a mesma chave, que vamos designar por ***CAM_Read_Next***.

```
 /******************** Interface do Módulo CAM ********************/
#ifndef _CAM
#define _CAM

 /******************** Definição de Constantes ********************/

#define OK          0 /* operação realizada com sucesso */
#define NULL_PTR    1 /* ponteiro nulo */
#define NULL_SIZE   2 /* tamanho nulo */
#define NO_MEM      3 /* memória esgotada */
#define CAM_EMPTY   4 /* CAM vazia */
#define CAM_FULL    5 /* CAM cheia */
#define CAM_EXISTS  6 /* já foi instanciada uma CAM */
#define NO_CAM      7 /* ainda não foi instanciada qualquer CAM */
#define NO_KEY      8 /* não existe elemento com a chave indicada */
#define NO_FUNC     9 /* não foi comunicada a função de comparação */

 /******************** Protótipos das Funções ********************/

int CAM_Create (unsigned int sz, CompFunc cmpf);
/* Concretiza a CAM para elementos de sz bytes e indica a função de
comparação cmpf, que permite manter a memória ordenada. Valores de
retorno: OK, NULL_SIZE, CAM_EXISTS ou NULL_PTR. */

int CAM_Destroy (void);
/* Destrói a CAM. Valores de retorno: OK ou NO_CAM. */

int CAM_In (void *pelemento);
/* Coloca o elemento apontado por pelemento na posição da CAM com a
chave indicada. Permite-se a existência na CAM de elementos
distintos com a mesma chave. Elementos com a mesma chave são
armazenados por ordem cronológica da colocação na CAM. Valores de
retorno: OK, NO_CAM, NULL_PTR, CAM_FULL, NO_MEM ou NO_FUNC. */

int CAM_Out (void *pelemento);
/* Retira o elemento da CAM com a chave indicada para o elemento
apontado por pelemento. Havendo mais do que um elemento com a chave
indicada será retirado o primeiro que foi introduzido. Valores de
retorno: OK, NO_CAM, NULL_PTR, CAM_EMPTY, NO_KEY ou NO_FUNC. */

int CAM_Read_First (void *pelemento);
/* Lê o conteúdo do primeiro elemento da CAM que contém a chave
indicada para o elemento apontado por pelemento. Valores de retorno:
OK, NO_CAM, NULL_PTR, CAM_EMPTY, NO_KEY ou NO_FUNC. */

int CAM_Read_Next (void *pelemento);
/* Lê o conteúdo do elemento seguinte da CAM que contém a chave
indicada para o elemento apontado por pelemento. Os elementos são
lidos sucessivamente a partir do primeiro. Valores de retorno: OK,
NO_CAM, NULL_PTR, CAM_EMPTY, NO_KEY ou NO_FUNC. */

#endif
```

Figura 5.12 - Ficheiro de interface da memória associativa abstracta.

As operações de criação ***CAM_Create*** e de destruição ***CAM_Destroy*** só existem para implementações abstractas e têm como função respectivamente, concretizar a memória para o tipo de elementos pretendidos e indicar a função de comparação dos elementos, e repor a situação inicial de memória ainda por concretizar de maneira a poder ser reutilizada, eventualmente para um novo tipo de dados.

5.5 Tipos de implementação

Existem três tipos de implementação de memórias:

- Numa implementação estática a memória é definida à partida e é fixa. A sua dimensão, ou seja o número de elementos que pode armazenar, é previamente conhecida e não pode ser alterada. As implementações estáticas são baseadas em agregados de elementos. Os elementos são de tipos de dados simples ou estruturas, e são definidos à partida, pelo que, só podem armazenar dados desse tipo. Independentemente da utilização da memória, ou seja, do número de elementos que contêm de facto informação, a ocupação de memória do computador é sempre a mesma. A implementação estática é a mais simples, mas, a menos versátil, uma vez que a memória tem uma dimensão inalterável e não pode ser reconfigurada em termos do tipo dos elementos que armazena, durante a execução do programa.

- Numa implementação semiestática a dimensão da memória é definida à partida e é fixa. Tal como na implementação estática, a sua dimensão é previamente conhecida e não pode ser alterada. As implementações semiestáticas são baseadas em agregados de ponteiros para elementos, que são atribuídos dinamicamente. Os elementos são criados por atribuição de memória dinâmica quando a informação é colocada na memória, e são eliminados quando a informação é retirada da memória, por libertação da memória dinâmica. A ocupação de memória do computador tem uma componente constante, que é o agregado de ponteiros que serve de estrutura básica de suporte e, uma componente variável que depende do número de elementos que contêm de facto informação. A memória tem uma organização mais complexa que a implementação estática. Mas é mais versátil, porque com a utilização de ponteiros para **void** é possível criar memórias de tipos de dados indefinidos, ou seja, tipos de dados que podem ser concretizados na altura da criação da memória. Este tipo de realização designa-se por memória de dados abstractos. Mas, a memória continua a ter uma dimensão inalterável, devido à estrutura de dados de suporte ser estática.

- Numa implementação dinâmica, a dimensão da memória é inicialmente nula e depois vai crescendo à medida das necessidades. As implementações dinâmicas são baseadas em estruturas ligadas, de tipo lista ligada, lista biligada ou árvore binária. Estas estruturas são constituídas por nós ou elos interligados entre si, que são criados ou eliminados na memória dinâmica à medida que a memória vai respectivamente crescendo ou decrescendo. Os elementos que contêm a informação a armazenar na memória, estão dependurados nos nós, através de um ponteiro para o elemento. Os elementos também são criados ou eliminados dinamicamente quando respectivamente se coloca ou retira a informação da memória. A ocupação de memória do computador depende do número de elementos efectivamente armazenados na memória. A memória tem uma organização mais complexa que a implementação semiestática, mas, é a implementação mais versátil, porque permite a realização de uma memória de dados abstractos e mais importante ainda, porque a sua dimensão não está limitada.

5.6 Memória de acesso aleatório (*RAM*)

Como a memória de acesso aleatório é uma memória de acesso indexado, então é implementada através de um estrutura de dados de tipo agregado. A implementação estática é baseada num agregado de elementos do tipo que se pretende armazenar, como se apresenta na Figura 5.13.

As funções de leitura e de escrita têm um parâmetro de entrada que é o índice do elemento de memória que vai ser processado. Uma vez que a memória permite o acesso a qualquer um dos seus elementos, para procurar um elemento específico, a função de pesquisa tanto pode usar a pesquisa sequencial como a pesquisa binária, caso a memória esteja ordenada. Para ordenar a memória pode ser usado qualquer algoritmo de ordenação.

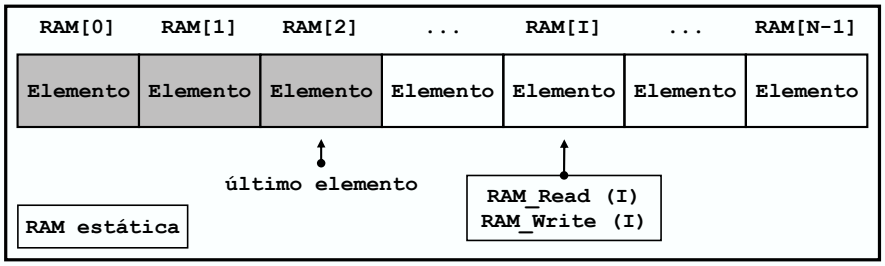

Figura 5.13 - Implementação estática da memória de acesso aleatório.

A implementação semiestática é baseada num agregado de ponteiros, como se apresenta na Figura 5.14. A operação de escrita de informação é responsável pela atribuição de memória para o elemento. Portanto, as operações de leitura, de pesquisa e de ordenação só podem aceder aos elementos existentes de facto na memória, pelo que, é muito importante disponibilizar um contador que indica o número de elementos úteis existentes na memória. Este contador comporta-se assim como um ponteiro indicador do último elemento da memória onde foi escrita informação. A actualização deste contador fica à responsabilidade da operação de escrita, que deve utilizar uma política de escrita em elementos sucessivos do agregado para que não haja elementos sem informação na parte utilizada do agregado.

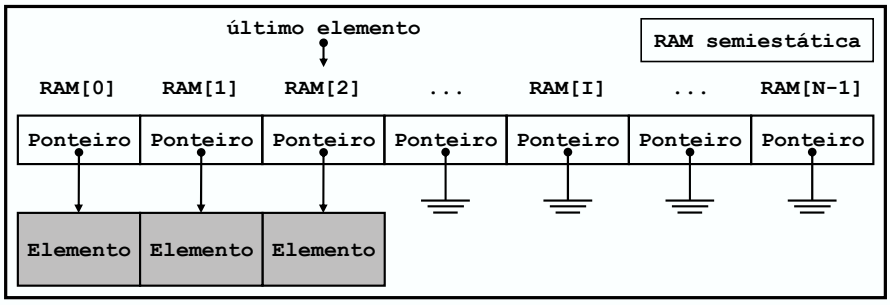

Figura 5.14 - Implementação semiestática da memória de acesso aleatório.

No caso da implementação semiestática é possível criar uma memória de acesso aleatório abstracta. Para isso, é necessário providenciar a função de criação da memória **RAM_Create** que concretiza o tipo de elementos, através da especificação do seu tamanho em *bytes*, e a função de destruição da memória **RAM_Destroy** que além de libertar a memória dinâmica atribuída para os elementos, coloca o indicador de tamanho dos elementos de novo a zero e o contador de elementos úteis a zero. Ou seja, coloca o indicador do último elemento útil da memória no início do agregado.

Para implementar uma memória de acesso aleatório dinâmica seria preciso recorrer a uma lista ligada de elementos. Só que depois a memória só poderia ser acedida sequencialmente a partir do elemento inicial e perder-se-ia o acesso indexado, pelo que, a memória deixaria então de ter as características de acesso aleatório. Portanto, não existe implementação dinâmica da memória de acesso aleatório.

5.7 Memória fila (*Queue*/*FIFO*)

A Figura 5.15 apresenta a implementação estática da memória fila, que é baseada num agregado de elementos do tipo que se pretende armazenar, usado de forma circular. A fila tem dois indicadores numéricos que indicam os índices dos elementos que são, em cada instante, a cabeça da fila e a cauda da fila. Por uma questão de implementação, a cauda da fila indica a primeira posição livre para a próxima operação de colocação de um elemento. Sempre que se coloca um elemento na fila o indicador de cauda da fila é deslocado para o elemento seguinte do agregado. Sempre que se retira um elemento da fila o indicador de cabeça da fila é deslocado para o elemento seguinte do agregado.

Esta implementação não é a forma habitual de funcionamento de um fila, onde sempre que o elemento da cabeça da fila sai da fila, toda a fila é deslocada para a frente. Mas, para evitar deslocamentos de elementos no agregado, é a cabeça da fila que se desloca para o elemento seguinte. Quando ao colocar um novo elemento na fila, a cauda da fila se sobrepor à cabeça da fila, então é sinal que a fila está cheia. Quando ao retirar um elemento da fila, a cabeça da fila se sobrepor à cauda da fila, então é sinal que a fila ficou vazia.

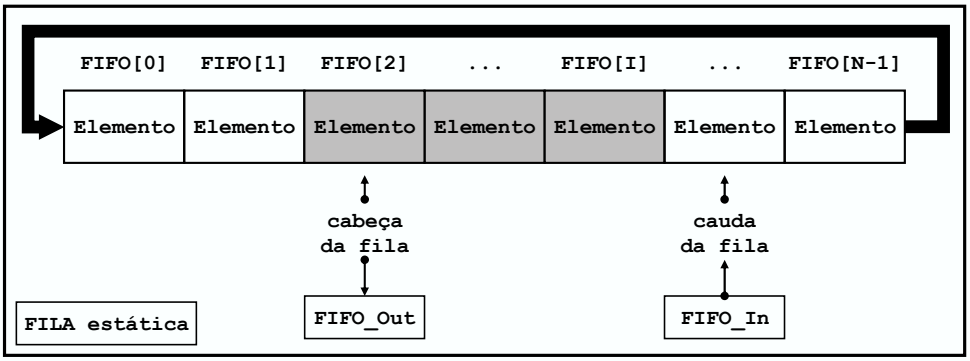

Figura 5.15 - Implementação estática da memória fila.

A implementação semiestática, que se apresenta na Figura 5.16, é baseada num agregado de ponteiros usado de forma circular e comporta-se da mesma forma que a implementação estática, mas, com as seguintes diferenças. A operação de colocação de um elemento na fila é responsável pela atribuição de memória para o elemento e a operação de remoção de um elemento da fila é responsável pela libertação da memória ocupada pelo elemento.

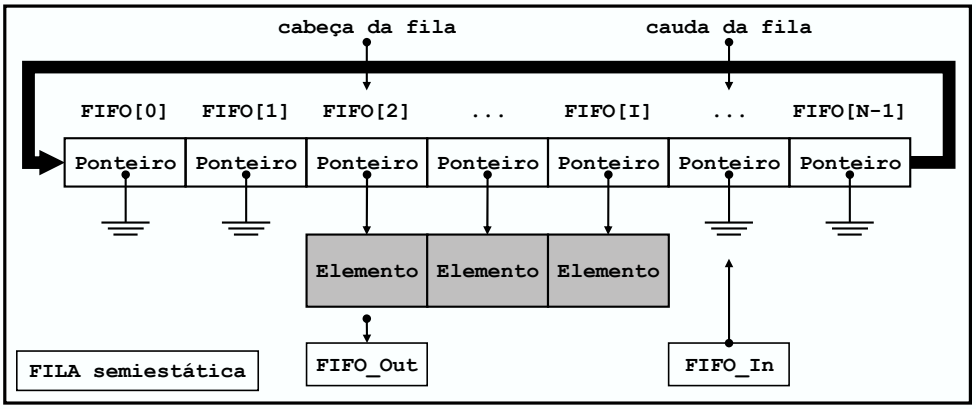

Figura 5.16 - Implementação semiestática da memória fila.

A implementação dinâmica, que se apresenta na Figura 5.17, é baseada numa lista ligada de elementos. Uma lista ligada é uma estrutura constituída por elementos, a que vamos chamar nós, ligados através de ponteiros. Cada nó da lista ligada é constituído por dois ponteiros. Uma para o elemento que armazena a informação e outro para o nó seguinte da lista. Repare que o último nó da lista aponta para NULL, para servir de indicador de finalização da fila. A memória para os nós e para os elementos da lista é atribuída, quando um elemento é colocado na fila e é libertada quando um elemento é retirado da fila. Os indicadores de cabeça e cauda da fila são ponteiros. A cabeça da fila aponta sempre para o elemento mais antigo que se encontra na fila e que é o primeiro a ser retirado. A cauda da fila aponta sempre para o elemento mais recente que se encontra na fila e à frente do qual um novo elemento é colocado. Quando são ambos ponteiros nulos, é sinal que a fila está vazia. Uma fila dinâmica nunca está cheia. Quando muito, pode não existir memória para continuar a acrescentar-lhe mais elementos.

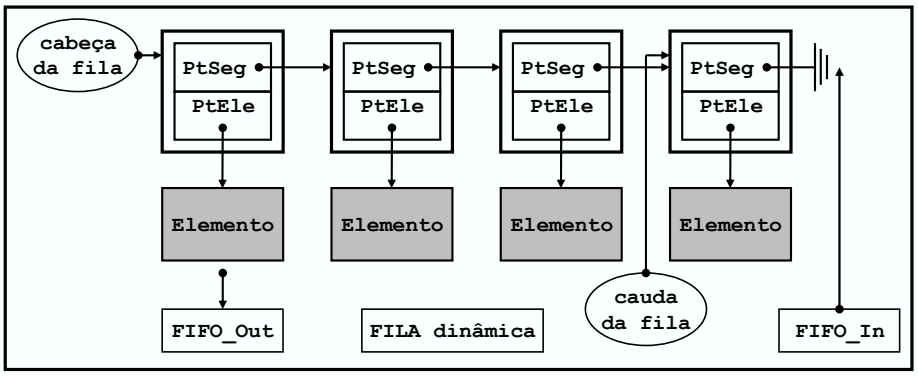

Figura 5.17 - Implementação dinâmica da memória fila.

No caso das implementações semiestática e dinâmica é possível criar uma fila abstracta. Para isso, é necessário providenciar a função de criação da fila **FIFO_Create** que concretiza o tipo de elementos, através da especificação do seu tamanho em *bytes*, e a função de destruição da fila **FIFO_Destroy** que além de libertar toda a memória dinâmica atribuída, coloca o indicador de tamanho dos elementos de novo a zero e recoloca os indicadores de cabeça e cauda da fila nas condições iniciais.

5.8 Memória pilha (*Stack*/*LIFO*)

Tal como a memória fila, a memória pilha tem as implementações estática e semiestática, que se apresentam na Figura 5.18, baseadas em agregados. A pilha tem um indicador numérico que indica o índice do elemento que é o topo da pilha. Por uma questão de implementação, o topo da pilha indica a primeira posição livre para a próxima operação de colocação de um elemento. Sempre que se coloca um elemento na pilha o indicador de topo da pilha é deslocado para o elemento seguinte do agregado. Quando o topo da pilha atinge o elemento a seguir ao fim do agregado, ou seja, o índice N, então é sinal que a pilha está cheia. Sempre que se retira um elemento da pilha, primeiro o indicador de topo da pilha é deslocado para o elemento anterior do agregado e depois o elemento é retirado da pilha. Quando o indicador de topo da pilha atinge o elemento inicial do agregado, ou seja, o índice 0, então é sinal que a pilha ficou vazia.

A implementação semiestática comporta-se da mesma forma que a implementação estática, mas, com as seguintes diferenças. A operação de colocação de um elemento na pilha é responsável pela atribuição de memória para o elemento e a operação de remoção de um elemento da pilha é responsável pela libertação da memória ocupada pelo elemento.

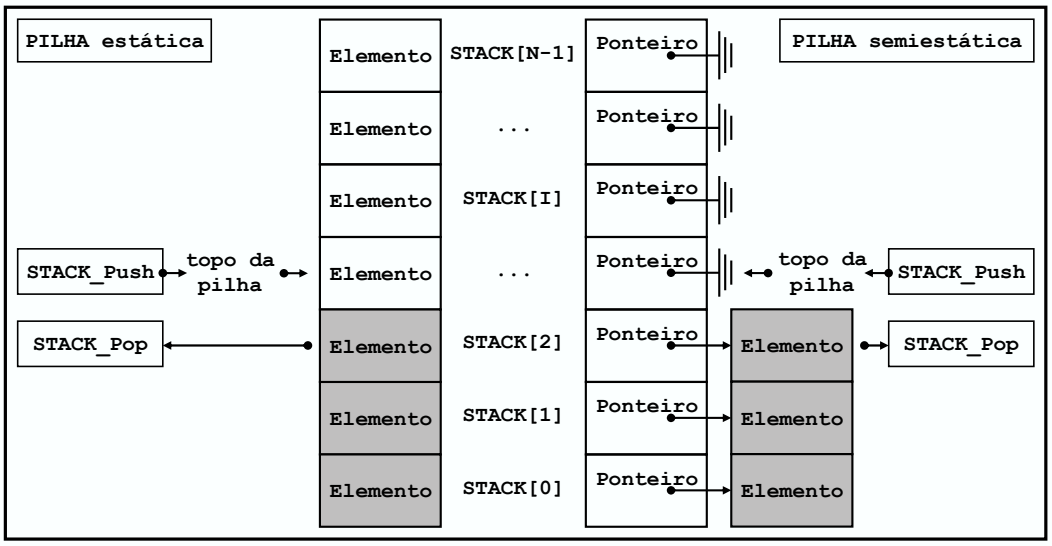

Figura 5.18 - Implementações estática e semiestática da memória pilha.

A implementação dinâmica de uma pilha, que se apresenta na Figura 5.19, é baseada numa lista ligada de elementos. Mas, enquanto que na fila cada nó da lista ligada aponta para o nó seguinte, na pilha cada nó aponta para o nó anterior, sendo que o primeiro nó da lista aponta para NULL, para servir de indicador de finalização da pilha. A memória para os nós e para os elementos da lista é atribuída, quando um elemento é colocado na pilha e é libertada quando um elemento é retirado da pilha. O indicador de topo da pilha é um ponteiro. O topo da pilha aponta sempre para o elemento mais recente que se encontra na pilha, que é o primeiro elemento a ser retirado e à frente do qual um novo elemento é colocado. Quando o topo da pilha é um ponteiro nulo, é sinal que a pilha está vazia. Uma pilha dinâmica nunca está cheia. Quando muito, pode não existir memória para continuar a acrescentar-lhe mais elementos.

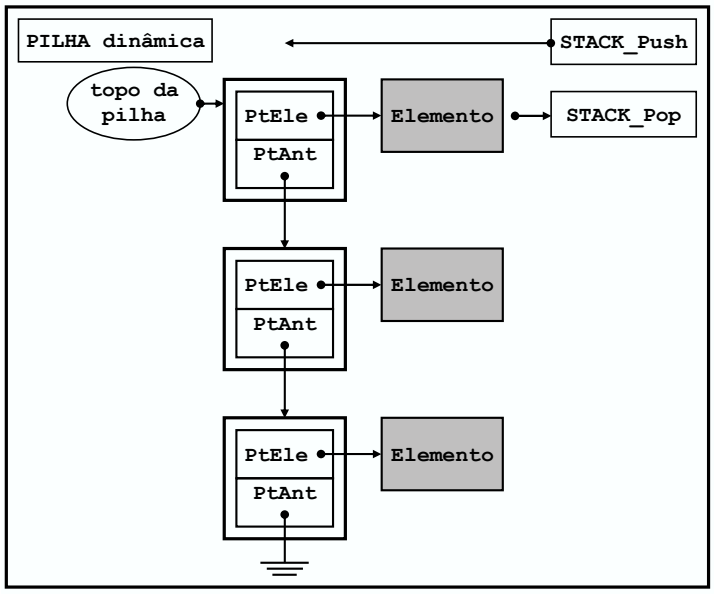

Figura 5.19- Implementação dinâmica da memória pilha.

No caso das implementações semiestática e dinâmica é possível criar uma pilha abstracta. Para isso, é necessário providenciar a função de criação da pilha **STACK_Create** que concretiza o tipo de elementos, através da especificação do seu tamanho em *bytes*, e a função de destruição da pilha **STACK_Destroy** que além de libertar a memória dinâmica atribuída, coloca o indicador de tamanho dos elementos de novo a zero e recoloca o indicador de topo da pilha na condição inicial.

5.9 Memória associativa (*CAM*)

As implementações estática e semiestática da memória associativa, que se apresentam na Figura 5.20, são baseadas em agregados. Quando se pretende colocar um elemento na memória é necessário pesquisar a memória para encontrar a posição de colocação do elemento, que depende da sua chave e dos elementos já existentes na memória. Quando se pretende retirar um elemento da memória também é necessário pesquisar a memória para encontrar a posição onde o elemento se encontra. Para implementar a pesquisa da memória é necessário que a memória mantenha um contador que indica o número de elementos úteis armazenados, ou seja, um indicador do último elemento da memória onde foi escrita informação. Este contador é actualizado pelas operações de colocação e remoção de elementos na memória. Por questões de eficiência, também é conveniente utilizar a pesquisa binária. Para manter a memória sempre ordenada, colocar ou retirar um elemento da memória pode implicar deslocar os elementos no agregado.

No caso da implementação semiestática, estes deslocamentos são deslocamentos de ponteiros, ou seja, de entidades de 4 *bytes*, pelo que, são operações menos custosas que na implementação estática. Na implementação semiestática, as operações de colocação e remoção de elementos são ainda responsáveis respectivamente, pela atribuição de memória para o elemento e pela libertação da memória ocupada pelo elemento.

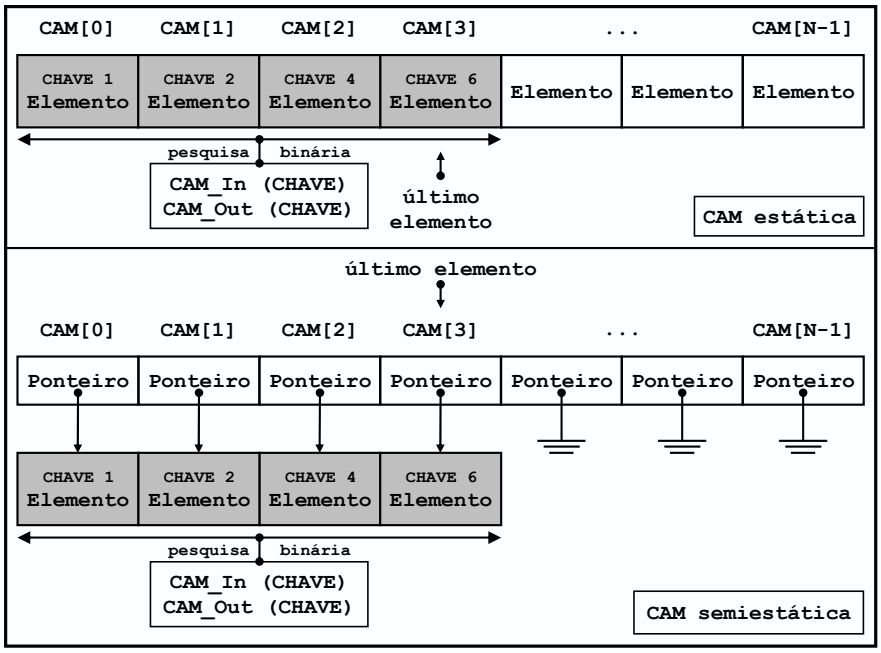

Figura 5.20 - Implementações estática e semiestática da memória associativa.

No caso da implementação semiestática é possível criar uma memória associativa abstracta. Para isso, é necessário providenciar a função de criação da memória associativa **CAM_Create** que concretiza o tipo de elementos, através da especificação do seu tamanho em *bytes*, e a função de destruição da memória associativa **CAM_Destroy** que além de libertar a memória dinâmica atribuída para os elementos, coloca o indicador de tamanho dos elementos de novo a zero e o contador de elementos úteis a zero. Ou seja, coloca o indicador do último elemento útil da memória associativa no início do agregado.

Para obter uma implementação mais eficiente da memória associativa, é essencial utilizar uma estrutura de dados que evite a necessidade de fazer deslocamentos dos elementos, quando se pretende colocar ou retirar um elemento da memória. Para isso precisamos de uma estrutura de dados dinâmica em que os elementos podem ser colocados e retirados por ajustamento de ligações entre os elementos. Mas, não pode ser uma lista ligada, porque para pesquisar a memória à procura de um elemento com uma determinada chave, a memória tem de ser pesquisada em ambos os sentidos. Pelo que, a lista tem de ser biligada.

Uma lista biligada é uma lista em que cada nó tem um ponteiro para o nó seguinte e um ponteiro para o nó anterior. Sendo que, o nó inicial aponta para trás para NULL e o nó final aponta para a frente para NULL, para servirem de indicadores de finalização da lista. Colocar ou retirar um elemento numa lista biligada, implica fazer ou desfazer mais ligações. Mas, em contrapartida todas as operações de ligação ou desligação ao nó de atrás e ao nó da frente são possíveis de fazer, tendo apenas um ponteiro a indicar, o nó atrás ou à frente do qual se vai colocar o novo elemento, ou o nó do elemento que vai ser eliminado.

A Figura 5.21 apresenta esta implementação dinâmica linear, que mantém um ponteiro para o primeiro nó da lista, que se designa por cabeça da memória. O inconveniente desta implementação tem a ver com a eficiência da pesquisa de informação. Numa lista, seja ela ligada ou biligada, só se pode implementar a pesquisa sequencial, a partir da cabeça da lista.

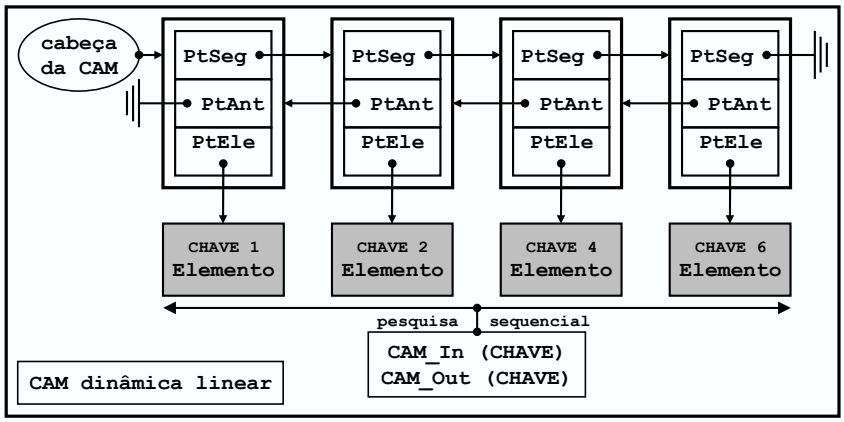

Figura 5.21 - Implementação dinâmica linear da memória associativa.

Para optimizarmos a pesquisa pode-se optar por outra estrutura de dados que implemente uma pesquisa ainda mais eficiente que a pesquisa binária, mas, que seja ainda uma estrutura de dados em que os elementos possam ser colocados e retirados sem custos de deslocamentos de elementos. Tal estrutura de dados, que se apresenta na Figura 5.22, é uma tabela de dispersão (*hashing table*).

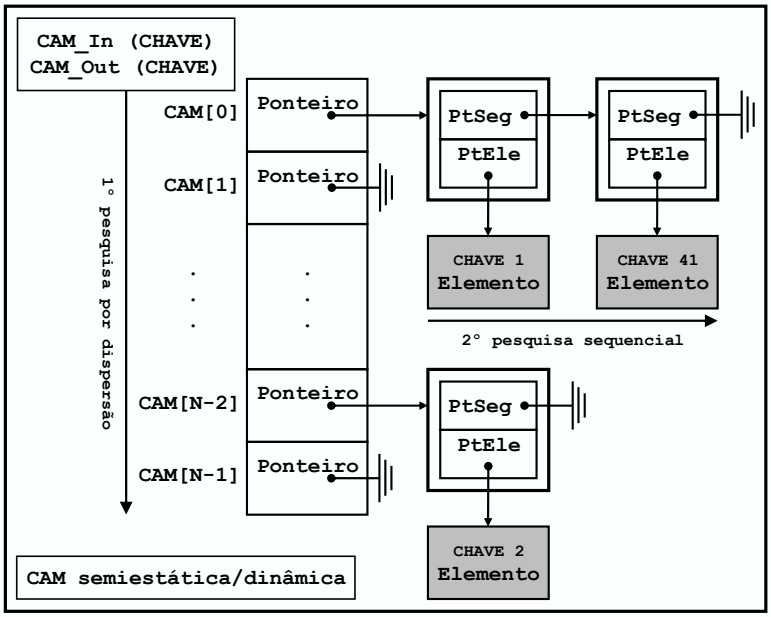

Figura 5.22 - Implementação da memória associativa com tabela de dispersão.

Uma tabela de dispersão é um agregado, onde os elementos são colocados em posições determinadas por uma função de dispersão (*hashing function*). Uma função de dispersão é uma função aritmética que determina a posição de colocação do elemento, usando a chave do elemento. Portanto, os elementos não são colocados em posições seguidas do agregado, mas sim em posições que dependem da chave do elemento.

Uma vez que o número de posições de armazenamento, ou seja, a dimensão do agregado é tipicamente várias ordens de grandeza menor do que o número total de chaves possíveis, podem ocorrer as situações de *overflow* e colisão. A situação de *overflow* acontece quando a tabela fica completamente preenchida e como o agregado é uma estrutura estática não tem solução, a não ser com um bom dimensionamento do agregado.

A situação de colisão acontece quando dadas duas chaves diferentes, a função de dispersão calcula a mesma a posição de colocação. O que compromete a implementação da memória, uma vez que, não se pode colocar dois elementos na mesma posição da tabela. Portanto, tem que se resolver o problema das colisões. Uma maneira passa por reduzir o seu número, utilizando uma função de dispersão que assegure uma boa dispersão.

Mas, independentemente da função usada vão existir sempre colisões. Portanto, tem de ser criada uma estrutura de dados que permita colocar mais do que um elemento na mesma posição da tabela. Uma solução possível consiste em criar uma estrutura semiestática em que cada elemento do agregado é um ponteiro que aponta para uma lista ligada de elementos. Temos assim uma tabela de listas ligadas.

Esta implementação é semiestática, sob o ponto de vista da estrutura de suporte, mas dinâmica sob o ponto de vista da colocação e remoção dos elementos. Permite resolver o problema das colisões e permite ainda a existência na memória associativa de elementos distintos com a mesma chave. Elementos com a mesma chave vão ficar na mesma posição da tabela e podem ser colocados na lista ligada por ordem cronológica da sua colocação na memória associativa, se por exemplo, a lista for implementada como uma fila.

Para pesquisar a existência de um elemento nesta implementação da memória associativa, seja para colocar um novo elemento, seja para retirar um elemento já existente, aplica-se a função de dispersão para a chave do elemento e obtém-se a posição da tabela onde o elemento deve estar colocado. Depois utiliza-se a pesquisa sequencial para analisar a lista ligada de elementos até detectar a posição de colocação ou remoção do elemento com a chave pretendida. Como o número de elementos existente na lista ligada é pequena, a pesquisa sequencial é aceitável. Por outro lado, nesta implementação a leitura sucessiva de elementos com a mesma chave é aplicada facilmente.

No entanto, a grande limitação da tabela de dispersão tem a ver com o facto da estrutura de suporte ser um agregado, que é uma estrutura de dados estática. Se a memória associativa necessitar de crescer esta solução não é a mais adequada. Portanto, é necessária uma estrutura de dados dinâmica que permita colocar e retirar elementos de forma eficiente, como na lista biligada, mas que suporte também uma pesquisa eficiente, como é o caso da pesquisa binária. Tal estrutura de dados é a árvore binária de pesquisa. A Figura 5.23 apresenta esta implementação dinâmica hierárquica.

Uma árvore é uma estrutura de dados constituída por uma colecção de nós. Esta colecção pode ser nula, ou constituída por um nó inicial, que se designa por raiz da árvore, e zero ou mais subárvores. Portanto, é uma estrutura de dados com uma organização recursiva, em que cada nó é também uma árvore. Uma árvore diz-se binária se cada nó não tiver mais do que duas subárvores, a subárvore da esquerda e a subárvore da direita. Numa árvore binária de pesquisa, um elemento é colocado na árvore de maneira que, os elementos da sua subárvore da esquerda têm uma chave menor do que a sua chave e os elementos da sua subárvore da direita têm uma chave maior do que a sua chave.

Para implementar uma árvore binária, cada nó da árvore é constituído por três ponteiros. Um para o elemento que armazena a informação e os outros dois para os nós seguintes da árvore, ou seja, para as subárvores da esquerda e da direita. Quando o nó seguinte não existe, então o respectivo ponteiro aponta para NULL, para servir de indicador de inexistência da subárvore.

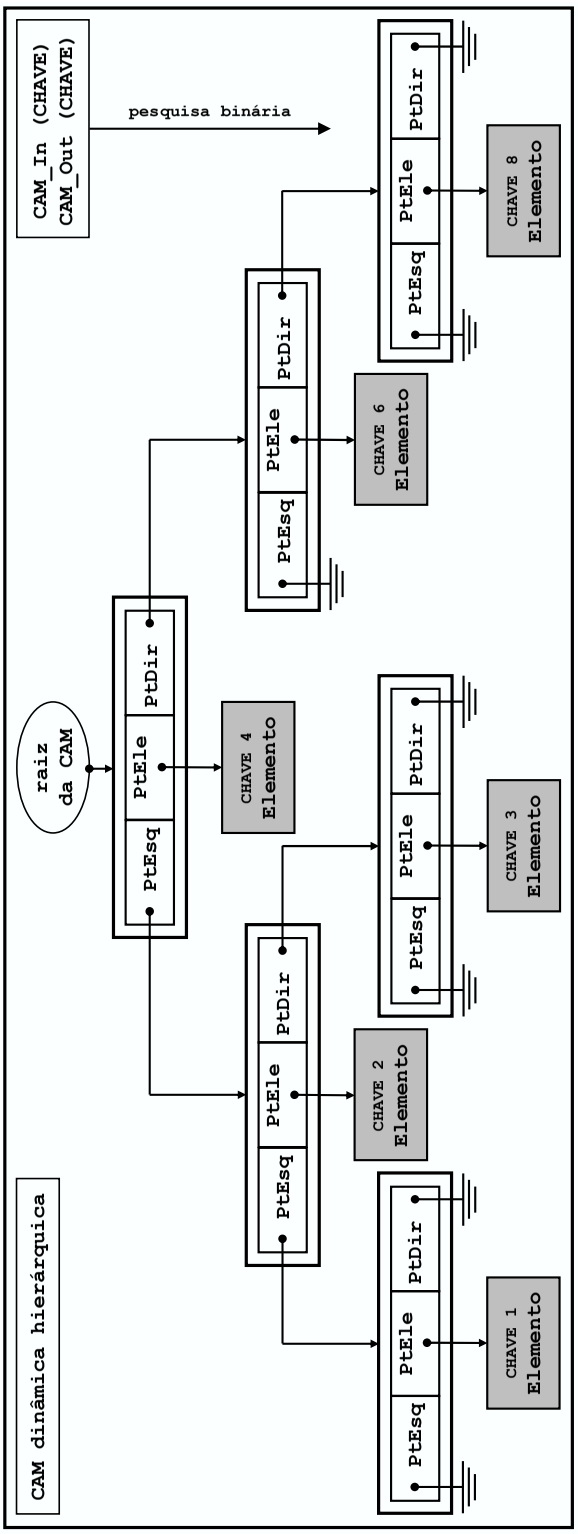

Figura 5.23 - Implementação dinâmica hierárquica da memória associativa.

Uma árvore binária diz-se completa, ou completamente balanceada, quando, para um número de níveis previamente fixado, contém o número máximo de nós, pelo que, se verifica que $N = 2^K - 1$, em que N é o número de nós e K o número de níveis da árvore. Se uma árvore estiver balanceada, então também as suas subárvores da esquerda e da direita o estarão. Por definição, considera-se que uma árvore vazia, ou seja, uma árvore sem qualquer nó, está balanceada.

Numa árvore binária de pesquisa pode ser aplicada a pesquisa binária de forma recursiva, mas, só em árvores completamente balanceadas é que se consegue obter uma pesquisa proporcional a $\log_2 N$. Portanto, é fundamental garantir-se em cada instante que a árvore se encontra organizada numa estrutura tão próxima quanto possível do balanceamento completo. Daqui resulta que, para se garantir a operacionalidade máxima de uma organização hierárquica em árvore binária, se deve conceber as operações para colocar e retirar elementos, como operações invariantes em termos de balanceamento.

5.10 Atribuição dinâmica de memória

Para permitir a construção de estruturas de dados semiestáticas e dinâmicas, a linguagem C permite a atribuição dinâmica de memória durante a execução do programa. A biblioteca de execução ANSI ***stdlib*** providencia as quatro funções que se apresentam na Figura 5.24.

Nome da função	Significado
**void ***malloc (size_t sz);	atribui **sz** *bytes* na memória. Devolve um ponteiro para o início do bloco atribuído ou NULL no caso contrário
**void ***calloc (size_t nelem, size_t sz);	atribui um espaço de memória contíguo de **nelem** objectos de **sz** *bytes* cada. Todos os *bits* do espaço de memória atribuído são inicializados a zero. Devolve um ponteiro para o início do bloco atribuído ou NULL no caso contrário
void ***realloc (void ***ptr, size_t nsz);	altera o tamanho do espaço de memória atribuído anteriormente e apontado por **ptr** para **nsz** *bytes*. Devolve um ponteiro para o início do bloco atribuído ou NULL no caso contrário
void free (**void ***ptr);	liberta o espaço de memória previamente atribuído pelas funções anteriores e apontado por **ptr**

Figura 5.24 - Funções para gestão da memória dinâmica.

A Figura 5.25 apresenta um exemplo de atribuição dinâmica de memória. Pretende-se escrever um programa que leia um ficheiro de texto constituído por números inteiros, um por linha, para os ordenar e imprimir no monitor. Vamos admitir, que o número de números inteiros armazenados no ficheiro depende de ficheiro para ficheiro e se encontra armazenado na primeira linha do ficheiro, que se comporta como cabeçalho do ficheiro. Uma vez que, a dimensão do agregado é dependente do tamanho do ficheiro, não podemos declarar o agregado de forma estática, sob pena de o dimensionarmos com tamanho insuficiente. A não ser que se dimensione o agregado por excesso, prevendo o pior caso. Normalmente, não é possível implementar esta estratégia, uma vez que, na maioria das aplicações é praticamente impossível prever o pior caso. Nem é tão pouco desejável, porque gera um desperdício de memória na maioria dos casos.

A solução correcta passa por criar o agregado dinamicamente, recorrendo à função **calloc**, assim que se saiba o número de números contido no ficheiro. Para tal declara-se o ponteiro PARINT que vai receber o endereço devolvido pela função, que é o endereço inicial do bloco de memória atribuído para o agregado. Devido à dualidade ponteiro agregado, o ponteiro PARINT é utilizado para aceder aos elementos do agregado, tal como se fosse um agregado. O programa antes de terminar, deve libertar a memória atribuída dinamicamente, usando para o efeito a função **free**.

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

int main (int argc, char *argv[])
{
  FILE *FP;  int *PARINT, N, I;

  if ( argc < 2 )          /* o número de argumentos é suficiente? */
  {
    fprintf (stderr, "Uso: %s nome do ficheiro\n", argv[0]);
    exit (EXIT_FAILURE);
  }

            /* abertura do ficheiro de entrada cujo nome é argv[1] */
  if ( (FP = fopen (argv[1], "r")) == NULL )
  {
    fprintf (stderr, "Não foi possível abrir o ficheiro %s\n"\
                  , argv[1]);
    exit (EXIT_FAILURE);
  }

  /* leitura do número de números inteiros armazenado no ficheiro */
  fscanf (FP, "%d", &N);

           /* atribuição de memória para um agregado de N inteiros */
  if ( (PARINT = (int *) calloc (N, sizeof (int))) == NULL )
  {
    fprintf (stderr, "Não foi possível atribuir o agregado\n");
    fclose (FP);                  /* fecho do ficheiro de entrada */
    exit (EXIT_FAILURE);
  }

  for (I = 0; I < N; I++)   /* leitura do ficheiro para o agregado */
    fscanf (FP, "%d", &PARINT[I]);

  fclose (FP);                    /* fecho do ficheiro de entrada */

  ...                                /* processamento do agregado */

  free (PARINT);/* libertação da memória atribuída para o agregado */

  return EXIT_SUCCESS;
}
```

Figura 5.25 - Exemplo de aplicação da função calloc.

5.11 Leituras recomendadas

- 4º capítulo do livro "Data Structures, Algorithms and Software Principles in C", de Thomas A. Standish, da editora Addison-Wesley Publishing Company, 1995.

- 8º capítulo do livro "C A Software Approach", 3ª edição, de Peter A. Darnell e Philip E. Margolis, da editora Springer-Verlag, 1996.

Capítulo 6

PESQUISA E ORDENAÇÃO

Sumário

Uma das tarefas mais habituais na programação é a pesquisa de informação, o que exige o desenvolvimento de algoritmos eficientes de pesquisa. Neste capítulo introduzimos os métodos de pesquisa sequencial, de pesquisa binária e de pesquisa por tabela. Apresentamos vários algoritmos que utilizam o método de pesquisa sequencial e o algoritmo de pesquisa binária, nas versões iterativa e recursiva.

Por outro lado a pesquisa é fortemente condicionada pela organização da informação, e por conseguinte, a ordenação é uma tarefa ainda mais frequente. Apresentamos as classes mais simples dos algoritmos de ordenação, que são, a ordenação por selecção, a ordenação por troca e a ordenação por inserção. Para cada uma delas expomos os algoritmos mais simples, que apesar de não serem os mais eficientes, são contudo suficientemente rápidos para pequenos agregados e apropriados para mostrar as características dos princípios de ordenação. Explicamos os algoritmos de ordenação Sequencial (*Sequential*), Selecção (*Selection*), Bolha (*Bubble*), Concha (*Shell*), Crivo (*Shaker*) e Inserção (*Insertion*), mostramos a sua execução com um pequeno agregado e fazemos a comparação do desempenho de cada um deles. Apresentamos também o algoritmo de ordenação Fusão de Listas (*Merge List*) que é muito versátil, porque pode ser aplicado, quer na ordenação de agregados, quer na ordenação de ficheiros. Apresentamos ainda os algoritmos de ordenação recursivos Fusão (*Merge*) e Rápido (*Quick*), que são muito eficientes.

Finalmente, explicamos como generalizar os algoritmos de ordenação e como contabilizar as instruções de comparação e de atribuições de elementos, de modo a poder fazer testes de desempenho dos algoritmos.

6.1 Pesquisa

Uma das tarefas mais comuns no dia a dia é a pesquisa de informação. Mas, a pesquisa depende muito da forma como a informação está organizada. Se a informação estiver completamente desordenada não temos outra alternativa que não seja analisar toda a informação por ordem, seja ela do princípio para o fim ou vice-versa, até encontrar o que pretendemos. Este processo de pesquisa é normalmente lento. Imagine, por exemplo, procurar a nota de um teste numa pauta não ordenada, no caso de uma disciplina com algumas centenas de alunos. Mas, se a informação estiver ordenada por uma ordem, seja ela crescente ou decrescente no caso de informação numérica, ascendente ou descendente no caso de informação textual, como por exemplo a mesma pauta listada por ordem do número mecanográfico, então é possível fazer uma procura mais ou menos selectiva, partindo a informação a procurar em intervalos sucessivos cada vez menores, evitando assim analisar informação irrelevante e acelerar o processo de pesquisa. E, se a informação além de estar ordenada por ordem, estiver ainda composta com entradas específicas, como por exemplo, uma pauta de uma disciplina com alunos de vários cursos, que se apresenta dividida em pautas por cursos, cada uma delas ordenada por número mecanográfico, então é possível fazer uma procura dirigida à pauta do respectivo curso e depois fazer uma procura em função do número mecanográfico, optimizando ainda mais o processo de pesquisa. Portanto, o método de pesquisa é inevitavelmente dependente da forma como a informação está organizada e apresentada. Quanto mais ordenada estiver a informação, mais eficiente poderá ser o método de pesquisa.

Tal como no dia a dia, a pesquisa de informação é também uma tarefa trivial em programação. Pesquisar um agregado à procura da localização de um determinado valor, ou de uma determinada característica acerca dos seus valores, é uma tarefa muito frequente e simples. Mas, é computacionalmente dispendiosa porque, o agregado pode ser constituído por centenas ou mesmo milhares de elementos. Pelo que, exige o desenvolvimento de algoritmos eficientes.

A maneira mais simples de pesquisar um agregado é a **pesquisa sequencial** (*sequencial search*), também chamada de **pesquisa linear**. Consiste em analisar todos os elementos do agregado de maneira metódica. A pesquisa começa no elemento inicial do agregado e avança elemento a elemento até encontrar o valor procurado, ou até atingir o elemento final do agregado. Este método de pesquisa é normalmente demorado, é dependente do tamanho do agregado, mas, não depende do arranjo interno dos elementos no agregado. Independentemente da forma como a informação está armazenada no agregado, o valor procurado será encontrado, caso exista no agregado.

No entanto, se tivermos informação à priori sobre os elementos do agregado é possível acelerar a pesquisa. Se por exemplo, os elementos estiverem ordenados por ordem crescente ou decrescente, é então possível fazer uma **pesquisa binária** (*binary search*), também chamada de **pesquisa logarítmica**. Este tipo de pesquisa começa por seleccionar o elemento central do agregado e compara-o com o valor procurado. Se o elemento escolhido for menor então podemos excluir a primeira metade do agregado e analisamos apenas a segunda metade. Caso contrário, então podemos excluir a segunda metade do agregado e analisamos apenas a primeira metade. Em cada passo da pesquisa, o número de elementos do agregado que têm de ser analisados é reduzido a metade, pelo que, este método de pesquisa é mais eficiente. O processo é repetido até que o elemento seleccionado seja o valor procurado, ou até que o número de elementos que têm de ser

analisados seja reduzido a zero, o que significa, que o valor procurado não existe no agregado.

Se os elementos do agregado, em vez de estarem colocados em posições sucessivas do agregado, estiverem colocados em posições predeterminadas do agregado, posições essas que são determinadas por uma função de dispersão (*hashing function*), então é possível fazer uma **pesquisa por tabela** (*table search*), também chamada de **pesquisa por dispersão** (*hashing*). Uma função de dispersão é uma função aritmética que determina a posição de colocação do elemento no agregado, tendo em consideração o valor do elemento, ou no caso de um elemento estruturado, usando um campo do elemento como chave de colocação no agregado. No entanto, normalmente uma função de dispersão produz situações de colisão. A situação de colisão acontece quando dados dois valores diferentes, a função de dispersão calcula a mesma a posição de colocação. Portanto, o agregado tem de permitir colocar mais do que um elemento numa mesma posição. Uma solução possível consiste em criar uma estrutura semiestática em que cada elemento do agregado é um ponteiro que aponta para uma lista ligada de elementos. Temos assim um agregado de listas ligadas. Este modelo de estrutura de dados, que se designa por uma tabela de dispersão com encadeamento de elementos, é pesquisado em dois tempos. Para procurar um elemento, primeiro é usada a função de dispersão para, dado o valor do elemento desejado, calcular a sua posição de colocação no agregado e aceder directamente a essa posição. Depois utiliza-se a pesquisa sequencial para analisar a lista ligada de elementos até detectar o elemento com o valor pretendido. Este método de pesquisa será abordado mais tarde a propósito da implementação de memórias associativas.

Para os algoritmos apresentados neste capítulo vamos considerar que a estrutura de dados a pesquisar ou ordenar, é um agregado de elementos inteiros de nome seq, com capacidade para armazenar NMAX elementos, tendo no entanto apenas nelem elementos úteis.

Tendo em consideração este agregado de elementos inteiros, vamos analisar vários algoritmos que utilizam a pesquisa sequencial e o algoritmo de pesquisa binária nas suas versões iterativa e recursiva. Vamos considerar que as funções só são invocadas para agregados que contêm de facto elementos com informação, ou seja, nelem é maior do que zero, pelo que, os algoritmos podem ser simplificados, uma vez que não têm que se preocupar com esta situação anómala.

6.1.1 Pesquisa Sequencial

Com o objectivo de tornar as funções, o mais versáteis possíveis, elas têm uma variável de entrada que indica o índice do agregado onde deve começar a pesquisa. As funções calculam e devolvem o índice do elemento onde está armazenado o valor procurado.

6.1.1.1 Procurar o maior valor de um agregado

A Figura 6.1 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do elemento com o maior valor armazenado na parte restante do agregado. No caso de existir mais do que um elemento com o mesmo valor a função devolve o índice do primeiro elemento encontrado. No início assume-se que o maior valor é o elemento de índice inicial e depois o agregado é analisado até ao último elemento, à procura de um valor ainda maior. Se pretendermos procurar o maior valor armazenado no agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
unsigned int ProcurarMaior (int seq[], unsigned int nelem,\
                            unsigned int inicio)
{
   unsigned int indmaior = inicio, indactual;

   for (indactual = inicio+1; indactual < nelem; indactual++)
     if (seq[indactual] > seq[indmaior]) indmaior = indactual;
   return indmaior;
}
```

Figura 6.1 - Função que determina o elemento de maior valor do agregado.

6.1.1.2 Procurar o menor valor de um agregado

A Figura 6.2 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do elemento com o menor valor armazenado na parte restante do agregado. No caso de existir mais do que um elemento com o mesmo valor a função devolve o índice do primeiro elemento encontrado. No início assume-se que o menor valor é o elemento de índice inicial e depois o agregado é analisado até ao último elemento, à procura de um valor ainda menor. Se pretendermos procurar o menor valor armazenado no agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
unsigned int ProcurarMenor (int seq[], unsigned int nelem,\
                            unsigned int inicio)
{
   unsigned int indmenor = inicio, indactual;

   for (indactual = inicio+1; indactual < nelem; indactual++)
     if (seq[indactual] < seq[indmenor]) indmenor = indactual;
   return indmenor;
}
```

Figura 6.2 - Função que determina o elemento de menor valor do agregado.

6.1.1.3 Procurar um valor no agregado

A Figura 6.3 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do elemento que armazena o valor que se pretende procurar na parte restante do agregado. Como não temos a certeza de que o valor existe de facto no agregado, se a pesquisa ultrapassar o último elemento do agregado sem o ter encontrado, então a função devolve o índice −1 como sinal de pesquisa falhada. A pesquisa deve acabar assim que a primeira ocorrência do valor pretendido seja encontrado, pelo que, se usa a instrução de **return** dentro do ciclo **for**, para terminar a pesquisa assim que se encontrar um elemento com o valor pretendido. Se pretendermos pesquisar todo o agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
int ProcurarValor (int seq[], unsigned int nelem,\
                   unsigned int inicio, int valor)
{
   unsigned int indactual;

   for (indactual = inicio; indactual < nelem; indactual++)
     if (seq[indactual] == valor) return indactual;
   return -1;
}
```

Figura 6.3 - Função de pesquisa sequencial que procura um valor no agregado.

6.1.1.4 Procurar o primeiro que serve

Existem alguns problemas concretos do dia a dia, cuja solução optimizada requer que se procure um determinado valor num agregado, mas, se este valor não existir podemos em alternativa utilizar um valor próximo do valor pretendido. Nesta situação podemos aplicar uma de três estratégias. A primeira e mais simples consiste em procurar no agregado, o primeiro valor que não excede o valor procurado. Este algoritmo designa-se por o primeiro que serve (*first fit*). A Figura 6.4 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do primeiro elemento do agregado com um valor que não excede o valor procurado. Como nesta pesquisa, não temos a certeza de que existe de facto tal valor, se a pesquisa ultrapassar o último elemento do agregado sem ter encontrado um valor, então a função devolve o índice −1 como sinal de pesquisa falhada. Como o nome indica, a pesquisa deve acabar assim que seja encontrado o primeiro valor que sirva, pelo que, se usa a instrução de **return** dentro do ciclo **for**, para terminar a pesquisa assim que se encontrar um elemento com o valor pretendido. Se pretendermos pesquisar todo o agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
int ProcurarPrimeiro (int seq[], unsigned int nelem,\
                      unsigned int inicio, int valor)
{
   unsigned int indactual;

   for (indactual = inicio; indactual < nelem; indactual++)
      if (seq[indactual] <= valor) return indactual;
   return -1;
}
```

Figura 6.4 - Função que determina o elemento do agregado com o primeiro valor que serve.

6.1.1.5 Procurar o melhor que serve

A segunda estratégia de optimização consiste em procurar no agregado, o valor mais próximo do valor pretendido, ou seja, o melhor valor. Este algoritmo designa-se por o melhor que serve (*best fit*). A Figura 6.5 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do elemento do agregado com o melhor valor que não excede o valor pretendido. Começa-se por procurar o primeiro valor que serve. Caso ele não exista, a função devolve o índice −1 como sinal de pesquisa falhada. Caso contrário procura-se até ao último elemento do agregado um elemento com um valor melhor, ou seja, um elemento com um valor que seja ainda maior do que o já encontrado e que não exceda o valor procurado. Se pretendermos pesquisar todo o agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
int ProcurarMelhor (int seq[], unsigned int nelem,\
                    unsigned int inicio, int valor)
{
   int indmelhor; unsigned int indactual;

   indmelhor = ProcurarPrimeiro (seq, nelem, inicio, valor);
   if (indmelhor == -1 ) return -1;
   for (indactual = indmelhor+1; indactual < nelem; indactual++)
      if (seq[indactual] > seq[indmelhor] && seq[indactual] <= valor)
         indmelhor = indactual;
   return indmelhor;
}
```

Figura 6.5 - Função que determina o elemento do agregado com o melhor valor que serve.

6.1.1.6 Procurar o pior que serve

A terceira estratégia de optimização consiste em procurar no agregado, o valor menos próximo do valor pretendido, ou seja, o pior valor. Este algoritmo designa-se por o pior que serve (*worst fit*). Por vezes, uma estratégia pela negativa é melhor do que pela positiva. Imagine-se por exemplo, que se pretende aproveitar uma sobra de 100 cm de uma calha de alumínio e que temos a necessidade de cortar secções de 80 cm, 50 cm e 40 cm. Uma estratégia o melhor que serve escolherá o valor 80 cm, desperdiçando 20 cm, enquanto que uma estratégia o pior que serve escolherá sucessivamente os valores 40 cm e 50 cm, desperdiçando apenas 10 cm.

A Figura 6.6 apresenta a função que determina, a partir de um índice inicial de pesquisa, o índice do elemento do agregado com o pior valor que não excede o valor pretendido. Começa-se por procurar o primeiro valor que serve. Caso ele não exista, a função devolve o índice −1 como sinal de pesquisa falhada. Caso contrário procura-se até ao último elemento do agregado um elemento com um valor pior, ou seja, um elemento com um valor que seja ainda menor do que o já encontrado. Se pretendermos pesquisar todo o agregado, basta invocar a função a partir do primeiro elemento do agregado.

```
int ProcurarPior (int seq[], unsigned int nelem,\
                  unsigned int inicio, int valor)
{
   int indpior; unsigned int indactual;

   indpior = ProcurarPrimeiro (seq, nelem, inicio, valor);
   if (indpior == -1 ) return -1;
   for (indactual = indpior+1; indactual < nelem; indactual++)
      if (seq[indactual] < seq[indpior]) indpior = indactual;
   return indpior;
}
```

Figura 6.6 - Função que determina o elemento do agregado com o pior valor que serve.

6.1.1.7 Exemplificação dos algoritmos de pesquisa sequencial

A Figura 6.7 apresenta a aplicação destes algoritmos num agregado com vinte elementos úteis. Se pesquisarmos o agregado a começar no elemento de índice 7, temos que, o maior valor é o elemento com índice 19, cujo valor é 250, e o menor valor é o elemento com índice 11, cujo valor é 1. Se procurarmos um elemento no agregado com valor 55, a começar no elemento de índice 7, encontramos o elemento com índice 16. Mas, se procurarmos um elemento com valor 40, não conseguimos encontrar nenhum valor, pelo que, a função devolve o índice −1.

O primeiro elemento com valor que não excede 40, a começar no elemento de índice 7, é o elemento com índice 7, cujo valor é 32. Nas mesmas circunstâncias, o elemento com o melhor valor é o elemento com índice 15, cujo valor é 39, e o elemento com o pior valor é o elemento com índice 11, cujo valor é 1. No entanto, se começarmos a pesquisa no elemento de índice 17 do agregado, não encontraremos nenhum valor que não exceda 40, pelo que, as funções, o primeiro que serve, o melhor que serve e o pior que serve devolvem o índice −1.

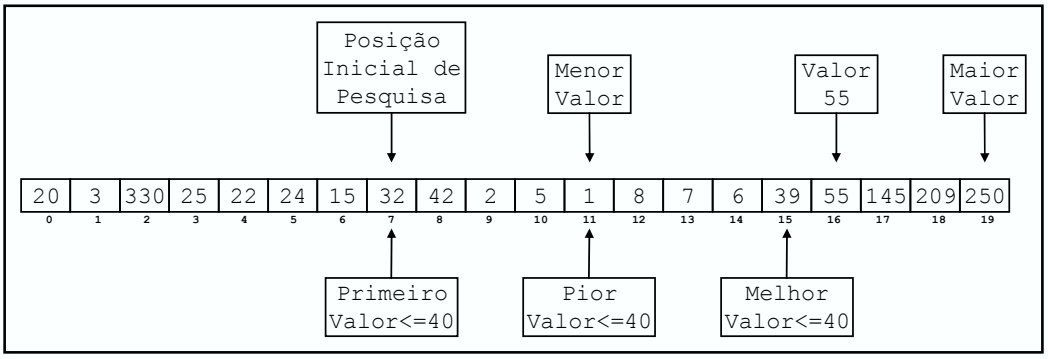

Figura 6.7 - Resultados da utilização dos algoritmos de pesquisa sequencial.

6.1.2 Pesquisa Binária

Uma forma de acelerar a pesquisa consiste em utilizar uma estratégia de partição sucessiva do agregado ao meio para diminuir o número de elementos a analisar. Mas, este método de pesquisa só funciona se os elementos estiverem ordenados. A Figura 6.8 apresenta a função de pesquisa binária, na sua versão iterativa, que calcula o índice do elemento com o valor procurado. Como não temos a certeza que o valor procurado existe de facto no agregado, a função devolve o índice −1 como sinal de pesquisa falhada. Com o objectivo de tornar a função o mais versátil possível, a função tem duas variáveis de entrada que indicam os índices dos elementos onde deve começar e acabar a pesquisa.

```
int ProcuraBinariaIterativa (int seq[], unsigned int inicio,\
                             unsigned int fim, int valor)
{
   unsigned int minimo = inicio, maximo = fim, medio;

   while (minimo <= maximo)
   {
      medio = (minimo + maximo) / 2;     /* cálculo da posição média */

      if (seq[medio] == valor) return medio; /* pesquisa com sucesso */

                /* actualização dos limites do intervalo de pesquisa */
      if (seq[medio] > valor) maximo = medio - 1;
      else minimo = medio + 1;

   return -1;                                        /* pesquisa falhada */
}
```

Figura 6.8 - Função de pesquisa binária que procura um valor no agregado (versão iterativa).

Vamos mostrar na Figura 6.9, o funcionamento deste algoritmo, para o caso de um agregado com vinte elementos ordenado por ordem crescente. Pretendemos procurar o valor 34. Vamos invocar a função para todos os elementos úteis do agregado, ou seja, do elemento de índice 0 até ao elemento nelem−1, que neste caso é o elemento de índice 19. Calcula-se o elemento médio do agregado, através da divisão inteira da soma das posições mínima e máxima, e que neste caso é o elemento de índice 9. Como o valor procurado, que é 34, é menor do que valor armazenado no elemento médio, neste caso 44, então isso significa que ele se encontra na primeira metade do agregado, pelo que, a posição máxima passa para o elemento à esquerda da posição média, ou seja, a nova posição máxima é agora o elemento de índice 8. Se pelo contrário, o valor procurado fosse maior do que 44 então isso significava que ele se encontrava na segunda metade do agregado, pelo que, a nova posição mínima passaria para o elemento à direita da posição média, ou seja, a nova

posição mínima seria o elemento de índice 10. Agora que estamos a analisar a primeira metade do agregado, calcula-se a nova posição média, que é o elemento de índice 4. Como o valor procurado é maior do que valor armazenado no elemento médio, que é 26, então isso significa que ele se encontra na segunda metade do intervalo em análise, pelo que, a posição mínima passa para o elemento à direita da posição média, ou seja, para o elemento de índice 5. Agora, a nova posição média passa a ser o elemento de índice 6. Como o valor procurado ainda é maior do que valor armazenado no elemento médio, que é 30, então isso significa que ele se encontra na segunda metade do intervalo em análise, pelo que, a posição mínima passa para o elemento à direita da posição média, ou seja, para o elemento de índice 7. A nova posição média passa agora a ser o elemento de índice 7, cujo valor é o valor procurado, pelo que, a pesquisa termina com sucesso e devolve o índice 7. Para obtermos este resultado, foi preciso analisar quatro elementos do agregado enquanto que a pesquisa sequencial necessitaria de analisar oito elementos.

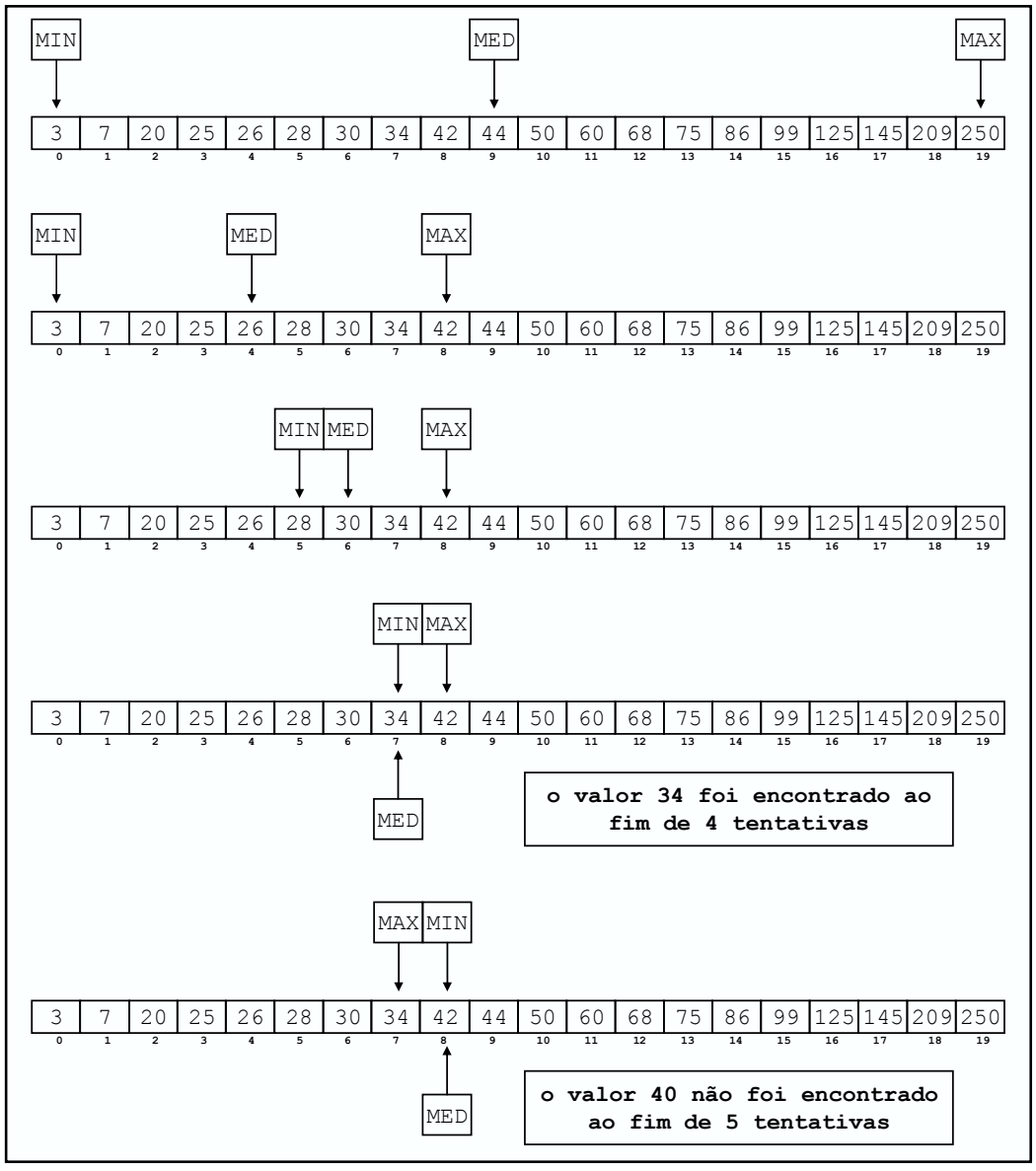

Figura 6.9 - Utilização do algoritmo de pesquisa binária num agregado ordenado.

Se por exemplo, o valor procurado fosse 40, que é maior do que 34, então a nova posição mínima passaria para o elemento à direita da posição média, ou seja, para o elemento de índice 8, exactamente igual ao valor da posição máxima. Consequentemente, a nova posição média passaria a ser o elemento de índice 8, cujo valor é 42, pelo que, a posição máxima passaria para o elemento à esquerda da posição média, ou seja, para o elemento de índice 7 e portanto, as posições mínima e máxima trocavam de posição parando o ciclo de pesquisa. Como o valor armazenado no elemento cujo índice é a posição média final, que é o elemento de índice 8, é diferente do valor procurado, então a pesquisa terminaria sem sucesso e devolveria o índice −1 como sinal de pesquisa falhada. Neste caso a função analisava cinco elementos do agregado enquanto que a pesquisa sequencial necessitaria de analisar os vinte elementos do agregado para chegar ao mesmo resultado.

Esta estratégia de pesquisa também pode ser implementada de forma recursiva, sendo que cada nova pesquisa analisa uma das metades do agregado anteriormente analisado até que o valor seja encontrado ou até não existirem mais elementos para analisar. A versão recursiva é apresentada na Figura 6.10. Compare-a com a versão iterativa que foi apresentada na Figura 6.8. A função começa por testar a situação de paragem no caso da pesquisa sem sucesso, o que acontece quando as posições inicial e final trocam de posição. Nesse caso, a função devolve o índice −1 como sinal de pesquisa falhada. Caso tal não se verifique, calcula o índice do elemento central do agregado em análise. Se o elemento for o valor procurado, estamos perante a situação de paragem com sucesso e a função devolve o índice do elemento central do agregado. Caso contrário, em função da comparação do valor procurado com o valor do elemento central, a função invoca-se recursivamente de forma alternativa para a primeira metade ou para a segunda metade do agregado.

```
int ProcuraBinariaRecursiva (int seq[], unsigned int inicio,\
                             unsigned int fim, int valor)
{
   unsigned int medio;

            /* condição de paragem no caso de pesquisa sem sucesso */
   if (inicio > fim) return -1;

   medio = (inicio + fim) / 2;          /* cálculo da posição média */

            /* condição de paragem no caso de pesquisa com sucesso */
   if (seq[medio] == valor) return medio;

   if (seq[medio] > valor)
        ProcuraBinariaRecursiva (seq, inicio, medio-1, valor);
          /* invocação recursiva para a primeira metade do agregado */
   else ProcuraBinariaRecursiva (seq, medio+1, fim, valor);
           /* invocação recursiva para a segunda metade do agregado */
}
```

Figura 6.10 - Função de pesquisa binária que procura um valor no agregado (versão recursiva).

A implementação recursiva não tem qualquer vantagem sobre a implementação repetitiva, quando aplicada a agregados. No entanto é normalmente aplicada para pesquisar estruturas de dados dinâmicas organizadas de forma binária, como é o caso das árvores binárias de pesquisa, permitindo combinar a eficiência da pesquisa binária com a flexibilidade destas estruturas de dados dinâmicas.

6.1.3 Comparação entre as pesquisas sequencial e binária

Dada a dificuldade em calcular com exactidão a eficiência de um algoritmo, para exprimir a sua eficiência em função do número de elementos processados, utiliza-se a notação matemática conhecida por ordem de magnitude ou notação O maiúsculo (*Big O Notation*). A ordem de magnitude de uma função é igual à ordem do seu termo que cresce mais rapidamente. Por exemplo, a função $f(n) = n^2+n$ é de ordem de magnitude $O(n^2)$, uma vez que para grandes valores de n, o termo n^2 cresce mais rapidamente que o termo n e portanto, domina a função.

Para um agregado com N elementos, o pior caso da pesquisa sequencial é quando o valor procurado está no último elemento do agregado ou não existe no agregado, o que exige a análise dos N elementos do agregado. Se considerarmos ainda, que a probabilidade do valor procurado estar em qualquer um dos elementos do agregado é igual, então a pesquisa sequencial analisa em média (N+1)/2 elementos. Se considerarmos que a probabilidade do valor não existir no agregado é igual a ser qualquer um dos elementos do agregado, então a pesquisa sequencial analisa em média (N+2)/2 elementos. Portanto, a eficiência do algoritmo de pesquisa sequencial é de ordem O(N).

Para o mesmo agregado, o pior caso da pesquisa binária é quando se reduz o intervalo em análise a apenas um elemento do agregado, o que exige a análise de $\log_2(N+1)$ elementos do agregado. Se considerarmos ainda, que a probabilidade do valor estar em qualquer um dos elementos do agregado é igual, então a pesquisa binária analisa em média $\log_2(N+1) - 1$ elementos. Se considerarmos que a probabilidade do valor não existir no agregado é igual a ser qualquer um dos elementos do agregado, então a pesquisa binária analisa em média $\log_2(N+1) - ½$ elementos. Portanto, a eficiência do algoritmo de pesquisa binária é de ordem $O(\log_2 N)$.

6.2 Ordenação

A ordenação é o processo de organizar um conjunto de objectos segundo uma determinada ordem. Como já foi dito anteriormente, se a informação estiver ordenada é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo a pesquisa binária ou a pesquisa por tabela. Pelo que, a ordenação é uma tarefa muito importante no processamento de dados e é feita para facilitar a pesquisa.

Os algoritmos de ordenação são classificados em dois tipos. A ordenação de informação armazenada em agregados designa-se por **ordenação interna**. Enquanto que, a ordenação de informação armazenada em ficheiros, designa-se por **ordenação externa**. Neste capítulo vamos apresentar alguns algoritmos de ordenação interna, se bem que alguns também possam ser utilizados para ordenação externa.

Existem algoritmos de ordenação muito eficientes para ordenar agregados de grandes dimensões, mas são normalmente complexos. Pelo contrário, os algoritmos de ordenação que vamos apresentar são simples, mas apropriados para mostrar as características dos princípios de ordenação. São normalmente pouco eficientes, mas são suficientemente rápidos para agregados de pequenas dimensões. Os algoritmos de ordenação que ordenam os elementos do agregado, no próprio agregado, fazendo para o efeito um rearranjo interno dos seus elementos, enquadram-se numa das três seguintes categorias: **ordenação por selecção**; **ordenação por troca**; e **ordenação por inserção**.

Para implementar os algoritmos de ordenação baseados em trocas de elementos vamos utilizar a função Swap que se apresenta na Figura 6.11. A troca de dois elementos do agregado exige uma variável temporária do mesmo tipo dos elementos do agregado e custa três instruções de atribuição. Os elementos a trocar são parâmetros de entrada-saída, pelo que, são passados por referência.

```
void Swap (int *x, int *y)
{
   int temp;
   temp = *x; *x = *y; *y = temp;
}
```

Figura 6.11 - Função para trocar dois elementos de um agregado de inteiros.

6.2.1 Ordenação por selecção

Os algoritmos de ordenação por selecção utilizam uma pesquisa sequencial. Fazem passagens sucessivas sobre todos os elementos de uma parte do agregado e em cada passagem seleccionam o elemento de menor valor de todos os elementos analisados, no caso de se pretender uma ordenação crescente, ou em alternativa, o elemento de maior valor de todos os elementos analisados, no caso de se pretender uma ordenação decrescente, colocando esse valor nos sucessivos elementos iniciais do agregado. Em cada passagem, um elemento, de entre todos os restantes elementos do agregado ainda não ordenados, é colocado no sítio certo. Mais concretamente, no caso de uma ordenação crescente, na primeira passagem, o menor valor de todos fica colocado no primeiro elemento do agregado, na segunda passagem, o segundo menor valor fica colocado no segundo elemento do agregado, e assim sucessivamente até que na última passagem o penúltimo menor valor, ou seja, o segundo maior valor, fica colocado no penúltimo elemento do agregado, e, consequentemente, o maior valor fica automaticamente colocado no último elemento do agregado. Vamos agora apresentar dois algoritmos que aplicam este método de ordenação.

A Figura 6.12 apresenta o algoritmo de **Ordenação Sequencial** (*Sequencial Sort*) para a ordenação crescente do agregado. Cada elemento do agregado é comparado com os restantes elementos do agregado abaixo dele. Se o elemento de cima, cujo índice é indi for menor que o elemento de baixo, cujo índice é indj, então trocam-se os elementos. Para efectuar a troca dos elementos do agregado é utilizada a função Swap. Só são analisados os primeiros nelem−1 elementos, porque, quando se ordena o penúltimo elemento do agregado, o último elemento fica automaticamente ordenado.

```
void Sequential_Sort (int seq[], unsigned int nelem)
{
   unsigned int indi, indj;

   for (indi = 0; indi < nelem-1; indi++)
      for (indj = indi+1; indj < nelem; indj++)
         if (seq[indi] > seq[indj])
            Swap (&seq[indi], &seq[indj]);     /* trocar os elementos */
}
```

Figura 6.12 - Algoritmo de ordenação Sequencial.

A Figura 6.13 apresenta a execução do algoritmo para um agregado com 10 elementos. Em cada passagem o elemento ordenado fica a sombreado para melhor se observar a parte ordenada do agregado. Na nona passagem para além do nono elemento, também o décimo

elemento fica ordenado. O algoritmo executa nove passagens e faz um total de 45 comparações e de 25 trocas para ordenar o agregado.

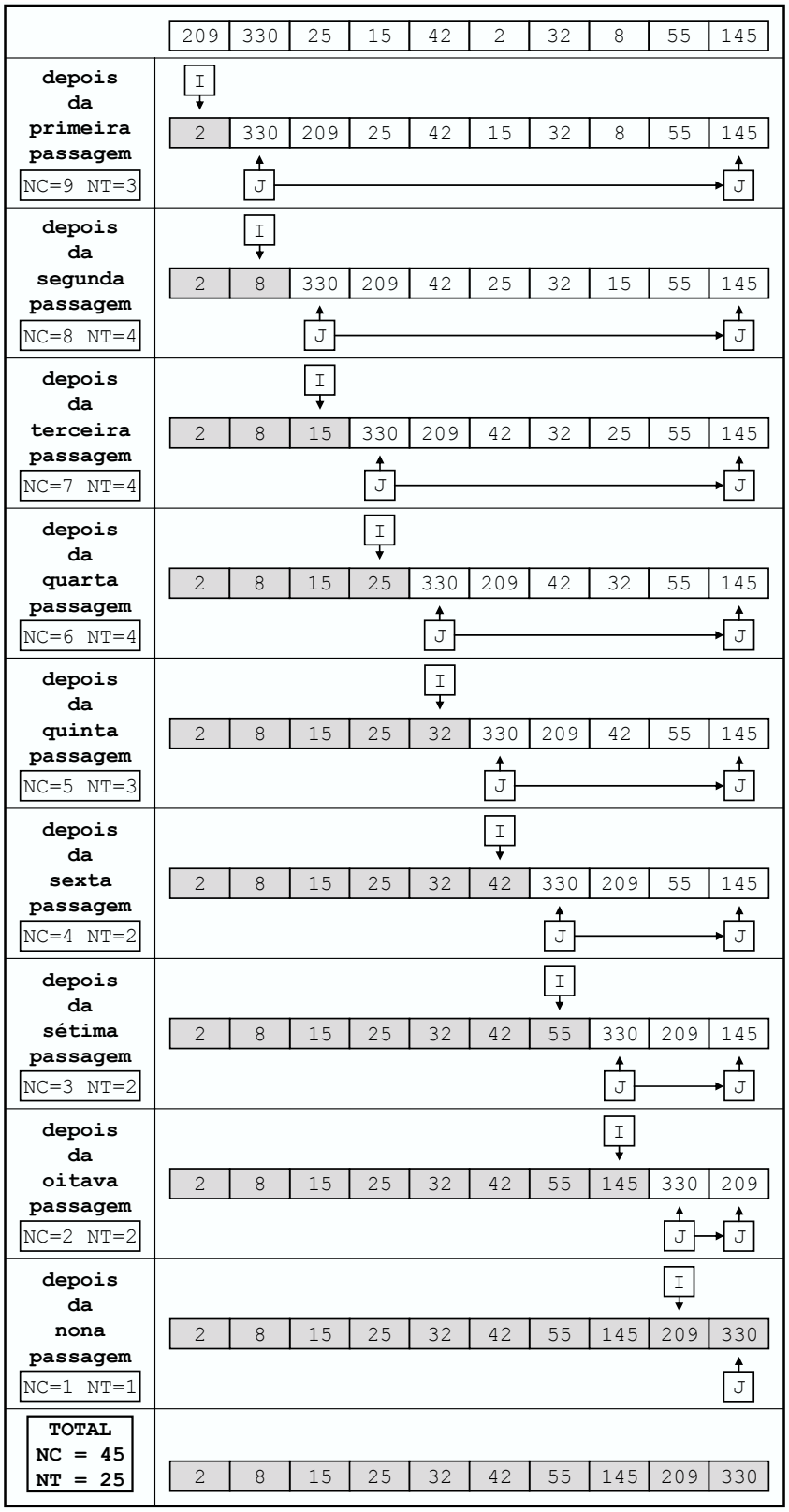

Figura 6.13 - Execução do algoritmo de ordenação Sequencial.

Para um agregado com N elementos, todos os elementos do agregado são comparados com todos os outros elementos, pelo que, este algoritmo faz $(N^2–N)/2$ comparações e o número de trocas se bem que dependente do grau de desordenação dos elementos, no pior caso pode atingir também as $(N^2–N)/2$ trocas. Portanto, este algoritmo tem uma eficiência de comparação de ordem $O(N^2)$ e uma eficiência de trocas de ordem $O(N^2)$, pelo que, pertence à classe $O(N^2)$.

Este algoritmo tem a desvantagem de eventualmente fazer trocas desnecessárias de elementos. Pelo que, para reduzir o número de trocas, em cada passagem deve-se detectar o elemento de menor valor, memorizando-se para o efeito o índice onde ele se encontra armazenado e só depois de comparar o elemento com os restantes elementos do agregado, é que se faz a troca do elemento analisado com o elemento de menor valor.

A Figura 6.14 apresenta o algoritmo de **Ordenação Selecção** (*Selection Sort*), que implementa esta optimização das trocas, para a ordenação crescente do agregado. Existe a variável auxiliar indmin que no início de cada passagem é inicializada a indi e que representa o índice do elemento de menor valor detectado durante a passagem. Cada elemento do agregado é comparado com os restantes elementos do agregado abaixo dele. Se o elemento de menor valor, cujo índice é indmin for menor que o elemento, cujo índice é indj, então este elemento com índice indj é o novo menor valor até então detectado, pelo que, o índice indmin é actualizado a indj. No final da passagem pelo agregado, se indmin for diferente de indi, então é sinal que o elemento de menor valor não está na posição inicial indi e, portanto, é preciso trocar os dois elementos.

```
void Selection_Sort (int seq[], unsigned int nelem)
{
   unsigned int indi, indj, indmin;

   for (indi = 0; indi < nelem-1; indi++)
   {
      indmin = indi;              /* o menor valor está na posição i */

      for (indj = indi+1; indj < nelem; indj++)
         if (seq[indmin] > seq[indj])
            indmin = indj;        /* o menor valor está na posição j */

      if (indmin != indi)
         Swap (&seq[indi], &seq[indmin]);     /* trocar os elementos */
   }
}
```

Figura 6.14 - Algoritmo de ordenação Selecção.

A Figura 6.15 apresenta a execução do algoritmo para um agregado com 10 elementos e o algoritmo executa nove passagens para ordenar o agregado, sendo o número total de comparações igual ao do algoritmo anterior. Na quarta e na nona passagens não é efectuada qualquer troca e nas restantes passagens é efectuada apenas uma troca, pelo que, o número de trocas é de apenas 7 contra as 25 do algoritmo anterior.

Para um agregado com N elementos, este algoritmo faz tantas comparações como o algoritmo Sequencial. No entanto, como faz no máximo uma troca por passagem, o número de trocas no pior caso pode atingir as $N-1$ trocas. Este algoritmo tem uma eficiência de comparação de ordem $O(N^2)$ e uma eficiência de trocas de ordem $O(N)$. Apesar de ser mais eficiente no número de trocas efectuadas, é no entanto, um algoritmo que pertence à classe $O(N^2)$.

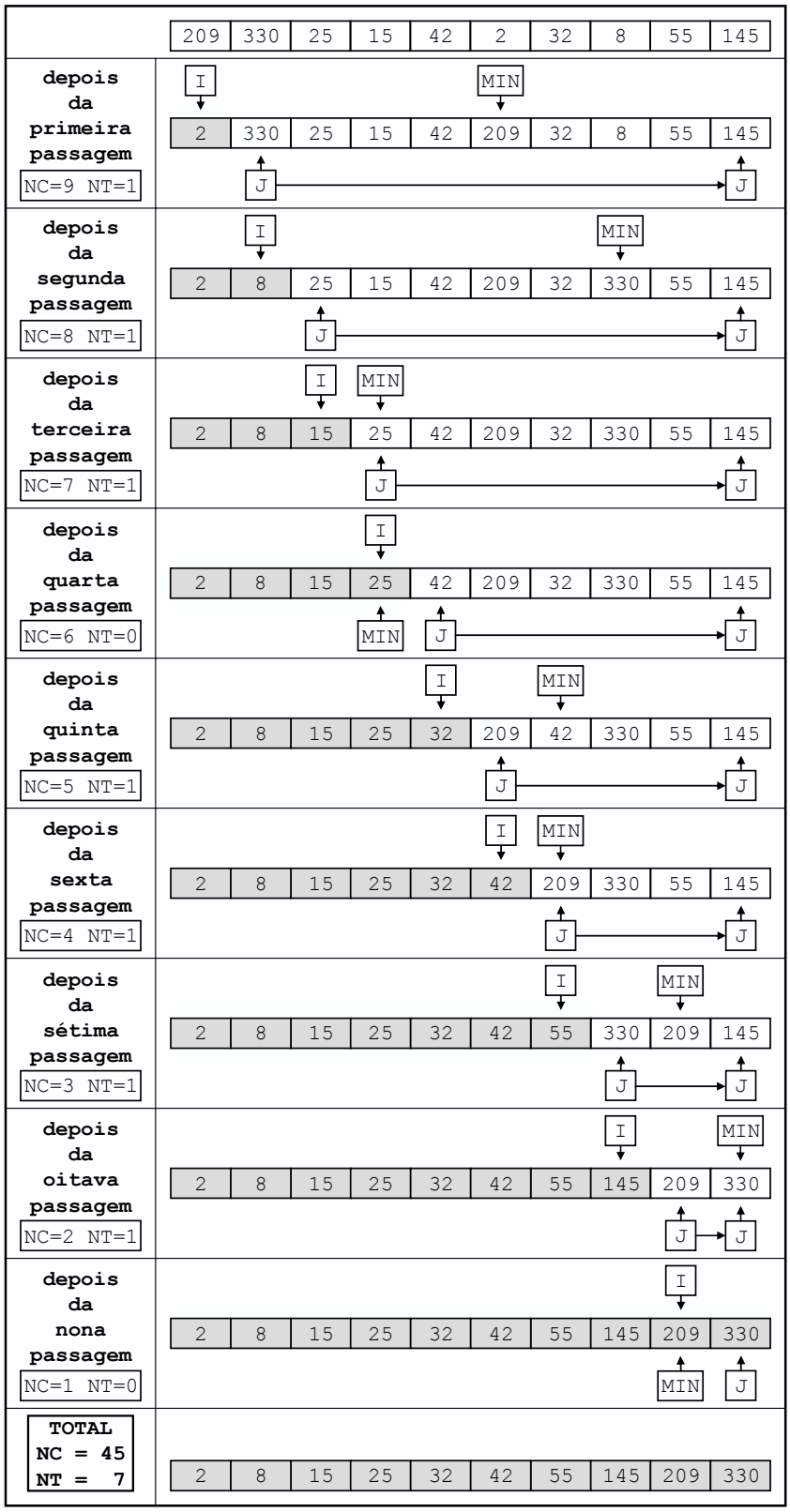

Figura 6.15 - Execução do algoritmo de ordenação Selecção.

6.2.2 Ordenação por troca

Os algoritmos de ordenação por troca, fazem passagens sucessivas sobre o agregado, comparando elementos que se encontram a uma distância fixa, que caso estejam fora de ordem são trocados. Quando durante uma passagem não se tiver efectuado qualquer troca, então é sinal de que os elementos que se encontram à distância de comparação já estão ordenados, pelo que, a ordenação termina. Vamos agora apresentar três algoritmos que aplicam este método de ordenação.

A Figura 6.16 apresenta o algoritmo de **Ordenação Bolha** (*Bubble Sort*) para a ordenação crescente do agregado. Existe a variável auxiliar indinicial que é inicializada com o índice do segundo elemento do agregado e que representa o índice do elemento da parte inicial do agregado onde termina a comparação de elementos em cada passagem. Para todos os elementos finais do agregado, desde o último elemento até ao elemento de índice indinicial, cada elemento é comparado com o elemento atrás dele, ou seja, compara-se o elemento de índice indi com o elemento de índice indi–1, e caso o valor seja menor trocam-se os elementos. Deste modo os elementos de menor valor vão sendo deslocados em direcção à parte inicial do agregado. Em cada passagem, contabiliza-se o número de trocas efectuadas, e quando uma passagem não tiver efectuado qualquer troca, isso é sinal de que o agregado já está ordenado. Em cada passagem, pelo menos um novo elemento fica ordenado, pelo que, a variável indinicial é incrementada de uma unidade, de maneira a evitar fazer-se comparações desnecessárias com os elementos que já estão ordenados na parte inicial do agregado. Portanto, a ordenação também acaba quando a variável indinicial excede o fim do agregado, ou seja, ao fim de nelem–1 passagens.

```
void Bubble_Sort (int seq[], unsigned int nelem)
{
   unsigned int indi, indinicial, ntrocas;
   indinicial = 1;    /* inicializar o limite superior de ordenação */
   do
   {
      ntrocas = 0;              /* inicializar o contador de trocas */
      for (indi = nelem-1; indi >= indinicial; indi--)
         if (seq[indi-1] > seq[indi])
         {
            Swap (&seq[indi], &seq[indi-1]);   /* trocar os elementos */
            ntrocas++;    /* actualizar o número de trocas efectuadas */
         }
      indinicial++;     /* actualizar o limite superior de ordenação */
   } while (ntrocas && indinicial < nelem);
}
```

Figura 6.16 - Algoritmo de ordenação Bolha.

A Figura 6.17 apresenta a execução do algoritmo para um agregado com 10 elementos. Uma vez que o agregado está bastante desordenado, o algoritmo executa nove passagens, se bem que na última não efectua qualquer troca. No entanto mesmo que efectuasse trocas na nona passagem, o algoritmo terminaria na mesma, uma vez que, o indicador de posição inicial excedeu o fim do agregado. O algoritmo faz um total de 45 comparações e de 25 trocas para ordenar o agregado.

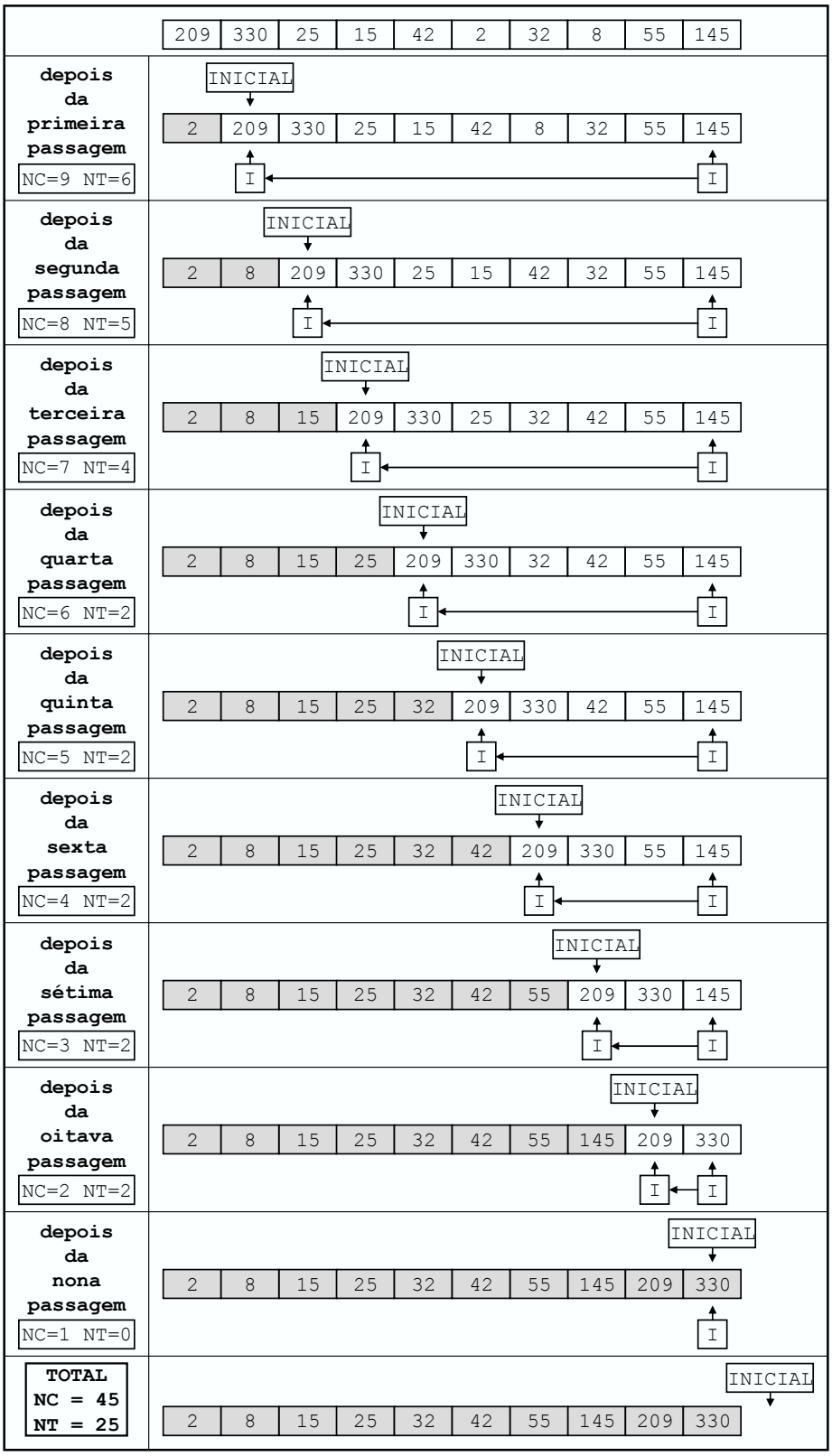

Figura 6.17 - Execução do algoritmo de ordenação Bolha.

Para um agregado com N elementos, este algoritmo faz no melhor caso apenas uma passagem, fazendo N–1 comparações e faz no pior caso N–1 passagens, fazendo $(N^2–N)/2$ comparações. Em média faz aproximadamente $N^2/3$ comparações. O número de trocas se bem que dependente do grau de desordenação dos elementos, no pior caso pode atingir as $(N^2–N)/2$ trocas. Portanto, este algoritmo tem uma eficiência de comparação de ordem $O(N^2)$, uma eficiência de trocas de ordem $O(N^2)$, pelo que, pertence à classe $O(N^2)$.

Como este algoritmo tem a capacidade de após cada passagem determinar se o agregado está ou não ordenado, então é indicado para ordenar agregados que estejam parcialmente desordenados. No entanto, caso o agregado esteja muito desordenado, ele é o pior dos algoritmos de ordenação, uma vez que faz muitas trocas. De maneira a diminuir o número de trocas, Donald L. Shell criou uma variante deste algoritmo, que em vez de comparar elementos adjacentes, compara elementos distanciados de um incremento que vai sendo progressivamente diminuído, até que nas últimas passagens compara elementos adjacentes.

A Figura 6.18 apresenta o algoritmo de **Ordenação Concha** (*Shell Sort*) para a ordenação crescente do agregado. Existe a variável auxiliar incremento que representa a distância de comparação e cujo valor inicial é metade do comprimento do agregado. Para os últimos elementos do agregado, mais concretamente para os nelem–incremento últimos elementos, compara-se o elemento de índice indi, com o elemento distanciado incremento elementos, ou seja com o elemento de índice indi–incremento, e caso o valor seja menor trocam-se os elementos. Deste modo os elementos de menor valor vão sendo deslocados em direcção à parte inicial do agregado. Em cada passagem, contabiliza-se o número de trocas efectuadas, e quando uma passagem não tiver efectuado qualquer troca, isso é sinal de que os elementos que estão separados da distância de comparação, já estão ordenados e a distância de comparação é reduzida, sendo dividida ao meio. O algoritmo é repetido até que a última distância de comparação usada seja igual a um. Quando numa passagem com distância de comparação unitária, não se efectuar qualquer troca de elementos, então é sinal que o agregado está ordenado. Ao contrário dos outros algoritmos de ordenação, até esta passagem final não há garantia que algum elemento do agregado já esteja ordenado. A série de incrementos utilizada é a que foi proposta pelo Shell.

```
void Shell_Sort (int seq[], unsigned int nelem)       /* 1ª versão */
{
   unsigned int indi, ntrocas, incremento;

   for (incremento = nelem/2; incremento > 0; incremento /= 2)
      do
      {
         ntrocas = 0;            /* inicializar o contador de trocas */

         for (indi = incremento; indi < nelem; indi++)
            if (seq[indi-incremento] > seq[indi])
            {                                  /* trocar os elementos */
               Swap (&seq[indi], &seq[indi-incremento]);

               ntrocas++;  /* actualizar o número de trocas efectuadas */
            }

      } while (ntrocas);
}
```

Figura 6.18 - Algoritmo de ordenação Concha (1ª versão).

A Figura 6.19 apresenta a execução do algoritmo para um agregado com 10 elementos. O algoritmo executa 7 passagens. Nas primeiras duas passagens, o incremento é de 5 elementos, nas três passagens seguintes o incremento é de 2 elementos e nas duas

passagens finais o incremento é de 1 elemento. Faz um total de 52 comparações e de 11 trocas para ordenar o agregado. Comparando com a ordenação Bolha, temos menos 2 passagens, mais algumas comparações, mas, menos de metade das trocas.

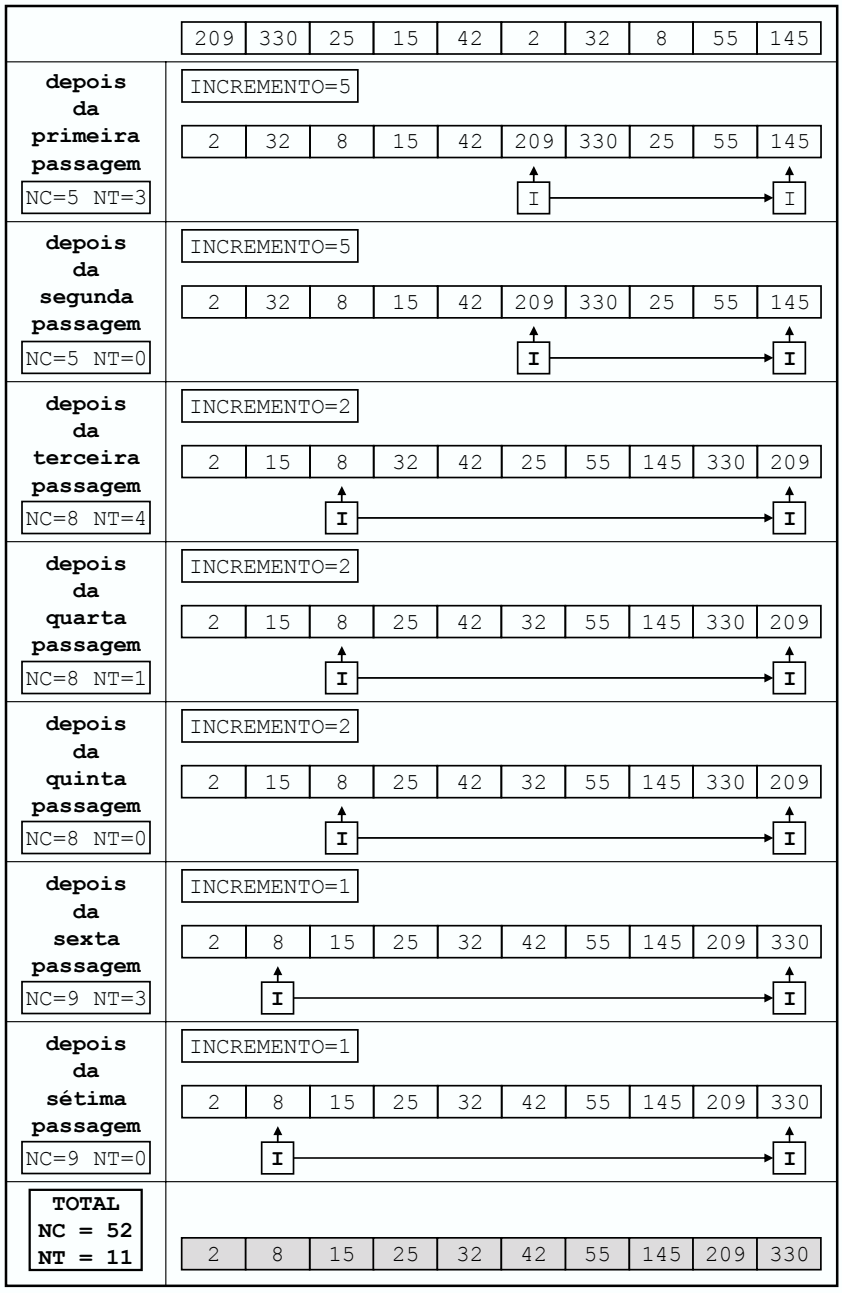

Figura 6.19 - Execução do algoritmo de ordenação Concha.

Normalmente, o algoritmo Concha é mais eficiente do que o algoritmo Bolha, uma vez que, as primeiras passagens analisam apenas parte dos elementos do agregado e portanto, fazem poucas trocas, mas trocam elementos que estão muito fora de sítio. Quando as últimas passagens, que analisam os elementos adjacentes, são efectuadas, então o agregado já se encontra parcialmente ordenado, pelo que, são necessárias poucas passagens e poucas trocas para acabar a ordenação. No entanto, para passar de incremento em incremento esta versão do algoritmo exige uma passagem sem trocas.

Existe uma implementação alternativa deste algoritmo, que se apresenta na Figura 6.20, que não necessita de fazer esta passagem extra, para determinar a passagem ao incremento seguinte. Utiliza uma técnica de inserção de elementos em vez de troca de elementos. Pega no agregado constituído pelos elementos que estão à distância de comparação e para cada um desses elementos faz a sua inserção na posição correcta, de maneira que este agregado fique ordenado. Depois passa à distância de comparação seguinte. Quando a distância de comparação é unitária, então estamos perante o algoritmo de Inserção, que se apresenta na Figura 6.23. Para ordenar o mesmo agregado, esta versão faz um total de 29 comparações e de 55 instruções de atribuição, ou seja, cópias de elementos, que correspondem a aproximadamente 18 trocas. Se por um lado esta versão é mais eficiente nas comparações, menos 23 o que dá uma eficiência na ordem dos 50%. Por outro lado, ela é menos eficiente nas trocas, com aproximadamente mais 7 trocas para trocar os 11 elementos do agregado que estão fora do sítio, ou seja, existe uma ineficiência na ordem dos 70%. Esta ineficiência é característica da técnica de inserção.

```
void Shell_Sort (int seq[], unsigned int nelem)        /* 2ª versão */
{
   unsigned int indi, indj, incremento;  int temp;

   for (incremento = nelem/2; incremento > 0; incremento /= 2)
     for (indi = incremento; indi < nelem; indi++)
     {
        temp = seq[indi];             /* copiar o elemento a ordenar */

        for (indj = indi; indj >= incremento; indj -= incremento)
          if (temp < seq[indj-incremento])
             seq[indj] = seq[indj-incremento]; /* deslocar elementos */
          else break;

        seq[indj] = temp;/* inserir o elemento a ordenar na posição */
     }
}
```

Figura 6.20 - Algoritmo de ordenação Concha (2ª versão).

O desempenho deste algoritmo depende da série de incrementos. Nesta versão e para um agregado com N elementos, em relação ao número de comparações, este algoritmo no pior caso e usando a série de incrementos propostos por Donald Shell, pertence à classe $O(N^2)$. Com a série de incrementos propostos por Hibbard pertence à classe $O(N^{3/2})$ e com a série de incrementos propostos por Sedgewick pertence à classe $O(N^{4/3})$.

Voltando ao algoritmo de ordenação Bolha, foi dito que em cada passagem pelo agregado, pelo menos um novo elemento fica ordenado. De facto todos os elementos atrás do elemento onde foi feita a última troca, já estão ordenados, pelo que, podemos colocar a variável indinicial, que representa o índice do elemento da parte inicial do agregado onde termina a comparação de elementos em cada passagem ascendente, na posição seguinte à posição onde foi feita a última troca. Desta forma evitam-se fazer ainda mais comparações desnecessárias com os elementos que já estão ordenados na parte inicial do agregado. Se aplicarmos esta técnica fazendo também passagens descendentes no agregado, alternadas com as passagens ascendentes, usando uma variável auxiliar indfinal que representa o índice do elemento da parte final do agregado onde termina a comparação de elementos em cada passagem descendente, podemos assim ordenar o agregado com menos comparações de elementos. Estas variáveis auxiliares podem ainda ser usadas para determinar quando o agregado está ordenado, pelo que, não é necessário contar o número de trocas efectuadas.
A Figura 6.21 apresenta o algoritmo de **Ordenação Crivo** (*Shaker Sort*) para a ordenação crescente do agregado. Existe a variável auxiliar indinicial que é inicializada com o índice do

segundo elemento do agregado e que representa o índice do elemento da parte inicial do agregado onde termina a comparação de elementos em cada passagem ascendente. Existe também a variável auxiliar indfinal que é inicializada com o índice do último elemento do agregado e que representa o índice do elemento da parte final do agregado onde termina a comparação de elementos em cada passagem descendente. Na passagem ascendente, cada elemento final do agregado, desde o elemento de índice indfinal até ao elemento de índice indinicial, é comparado com o elemento atrás dele, ou seja, compara-se o elemento de índice indi com o elemento de índice indi–1, e caso o valor seja menor trocam-se os elementos. O índice do elemento onde foi feita a última troca é armazenado e após ter sido completada a passagem ascendente, o índice indinicial é colocado na posição seguinte à posição da última troca. Deste modo os elementos de menor valor vão sendo deslocados em direcção à parte inicial do agregado. Depois na passagem descendente, cada elemento inicial do agregado, desde o elemento de índice indinicial até ao elemento de índice indfinal, é comparado com o elemento atrás dele, ou seja, compara-se o elemento de índice indi com o elemento de índice indi–1, e caso o valor seja menor trocam-se os elementos. O índice do elemento onde foi feita a última troca é armazenado e após ter sido completada a passagem descendente, o índice indfinal é colocado na posição anterior à posição da última troca. Deste modo os elementos de maior valor vão sendo deslocados em direcção à parte final do agregado.

Em cada passagem ascendente e descendente, ordena-se pelo menos dois elementos, um na parte inicial do agregado e outro na parte final do agregado. Quando não é feita qualquer troca durante uma passagem ascendente, o índice indinicial será colocado depois do índice indfinal. Por outro lado, quando não é feita qualquer troca durante uma passagem descendente, o índice indfinal será colocado antes do índice indinicial. Qualquer destas situações é sinal de que os elementos entre o índice indinicial e o índice indfinal também estão ordenados e a ordenação termina, porque o agregado está todo ordenado.

```
void Shaker_Sort (int seq[], unsigned int nelem)
{
   unsigned int indi, indinicial = 1, indfinal = nelem-1, utroca;

   do
   {
      /* passagem ascendente */
      for (indi = indfinal; indi >= indinicial; indi--)
         if (seq[indi-1] > seq[indi])
         {
            Swap (&seq[indi], &seq[indi-1]);   /* trocar os elementos */
            utroca = indi;    /* actualizar a posição da última troca */
         }
      indinicial = utroca+1;         /* actualizar o limite superior */

      /* passagem descendente */
      for (indi = indinicial; indi <= indfinal; indi++)
         if (seq[indi-1] > seq[indi])
         {
            Swap (&seq[indi], &seq[indi-1]);   /* trocar os elementos */
            utroca = indi;    /* actualizar a posição da última troca */
         }
      indfinal = utroca-1;           /* actualizar o limite inferior */
   } while (indinicial < indfinal);
}
```

Figura 6.21 - Algoritmo de ordenação Crivo.

A Figura 6.22 apresenta a execução do algoritmo para um agregado com 10 elementos. O algoritmo executa 6 passagens, 3 em cada sentido. Na terceira passagem descendente não é efectuada qualquer troca, pelo que, o índice de posição final passa para trás do índice de posição inicial parando o processo de ordenação. O algoritmo faz um total de 39 comparações e de 25 trocas para ordenar o agregado. Comparando com a ordenação Bolha, temos menos 3 passagens e menos 6 comparações, e o mesmo número de trocas. Os elementos do agregado que vão ficando ordenados, estão a sombreado para melhor se observar as duas extremidades ordenadas do agregado.

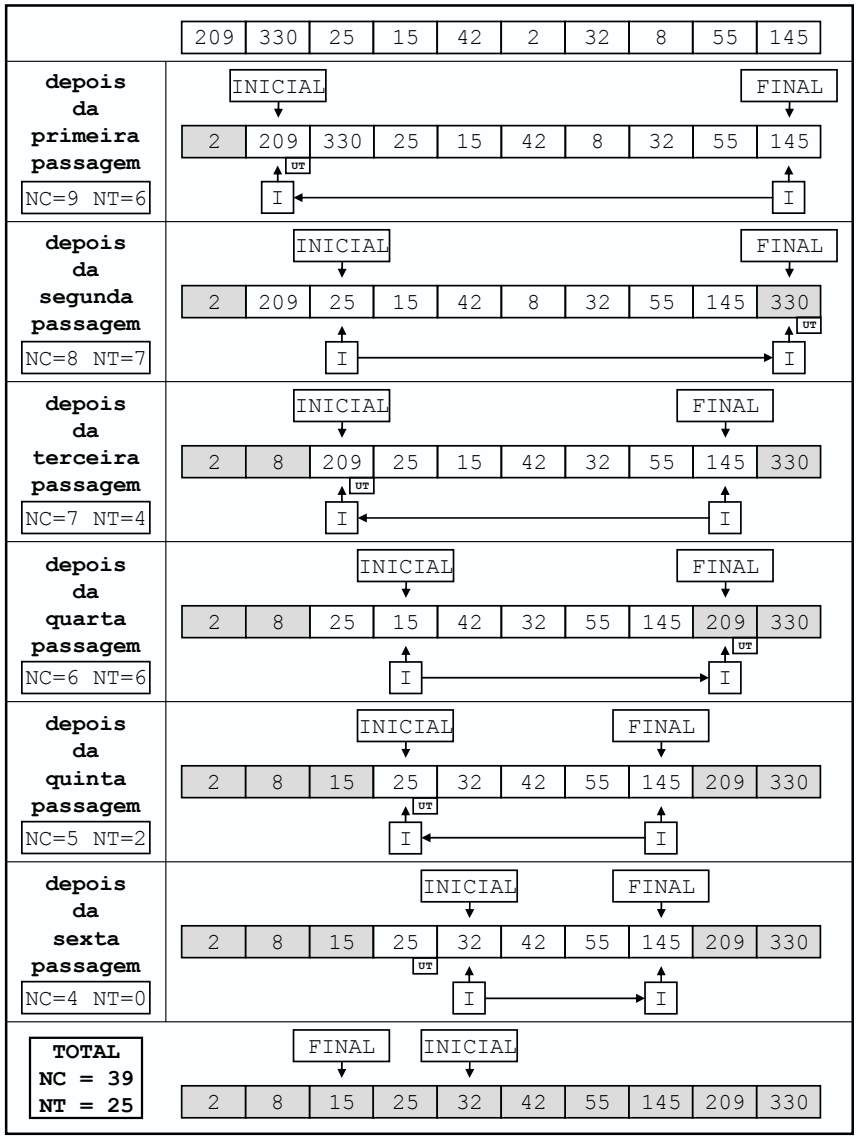

Figura 6.22 - Execução do algoritmo de ordenação Crivo.

O desempenho deste algoritmo é complexo de analisar. Em comparação com o algoritmo Bolha, tem o mesmo desempenho em relação ao número de trocas e é mais eficiente em relação ao número de comparações. No entanto, também pertence à classe $O(N^2)$.

6.2.3 Ordenação por inserção

A ordenação por inserção é a técnica de ordenação que normalmente usamos, nomeadamente, para ordenar uma mão de cartas. Consiste em determinar para cada elemento do agregado, a posição de inserção para que ele fique ordenado e inseri-lo nessa posição, deslocando para o efeito os restantes elementos.

A Figura 6.23 apresenta o algoritmo de **Ordenação Inserção** (*Insertion Sort*) para a ordenação crescente do agregado. Durante o processo de ordenação, o agregado está dividido em duas partes. A parte inicial do agregado está ordenada e a parte final está desordenada. O algoritmo começa por considerar que o primeiro elemento do agregado já está ordenado e depois ordena os restantes elementos do agregado, ou seja, desde o segundo ao último elemento. Cada elemento a ordenar é previamente copiado para uma variável temporária, de forma a abrir uma posição livre que permita o deslocamento dos elementos que estão atrás dele e que são maiores do que ele. Enquanto houver elementos na parte ordenada do agregado, ou seja, elementos que estão atrás do elemento a ordenar que sejam maiores do que ele, estes elementos são deslocados uma posição para a frente. Quando o deslocamento dos elementos terminar, o elemento a ordenar é colocado na posição que foi aberta pelo deslocamento dos elementos. Se o elemento já estiver na posição correcta, porque é maior do que todos os que estão atrás de si, então não há deslocamento de elementos e o elemento é colocado sobre ele próprio. Nesta situação, são feitas duas instruções de cópia inutilmente. Esta ineficiência é característica desta implementação do algoritmo de inserção. O algoritmo faz praticamente tantas comparações como deslocamentos. Mais concretamente, para ordenar cada elemento, faz mais uma comparação, para poder detectar o fim dos deslocamentos.

```
void Insertion_Sort (int seq[], unsigned int nelem)
{
   unsigned int indi, indd;  int temp;

   for (indi = 1; indi < nelem; indi++)
   {
      temp = seq[indi];               /* copiar o elemento a ordenar */

            /* deslocar os elementos atrás dele que lhe são maiores */
      for (indd = indi; indd > 0 && seq[indd-1] > temp; indd--)
         seq[indd] = seq[indd-1];

      seq[indd] = temp;   /* inserir o elemento a ordenar na posição */
   }
}
```

Figura 6.23 - Algoritmo de ordenação Inserção.

Para comparar este algoritmo com os anteriores, temos que ter em consideração, que para ordenar cada elemento, são precisas duas instruções de atribuição. Uma para copiar o elemento para uma variável temporária e outra para o inserir na posição definitiva. E o deslocamento de cada elemento custa ainda uma instrução de atribuição. Pelo que, o número de instruções de atribuição é NA = ND + 2 * (N–1). Como cada instrução de atribuição é equivalente, a um terço das instruções de atribuição que são necessárias para efectuar a troca de dois elementos do agregado, nos algoritmos anteriores baseados em trocas, então temos que NT = (ND + 2 * (N–1))/3.

A Figura 6.24 apresenta a execução do algoritmo para um agregado com 10 elementos. O algoritmo faz um total de 34 comparações e de 43 instruções de atribuição para efectuar os

25 deslocamentos de elementos que são necessários para ordenar o agregado, o que é equivalente a aproximadamente 14 trocas.

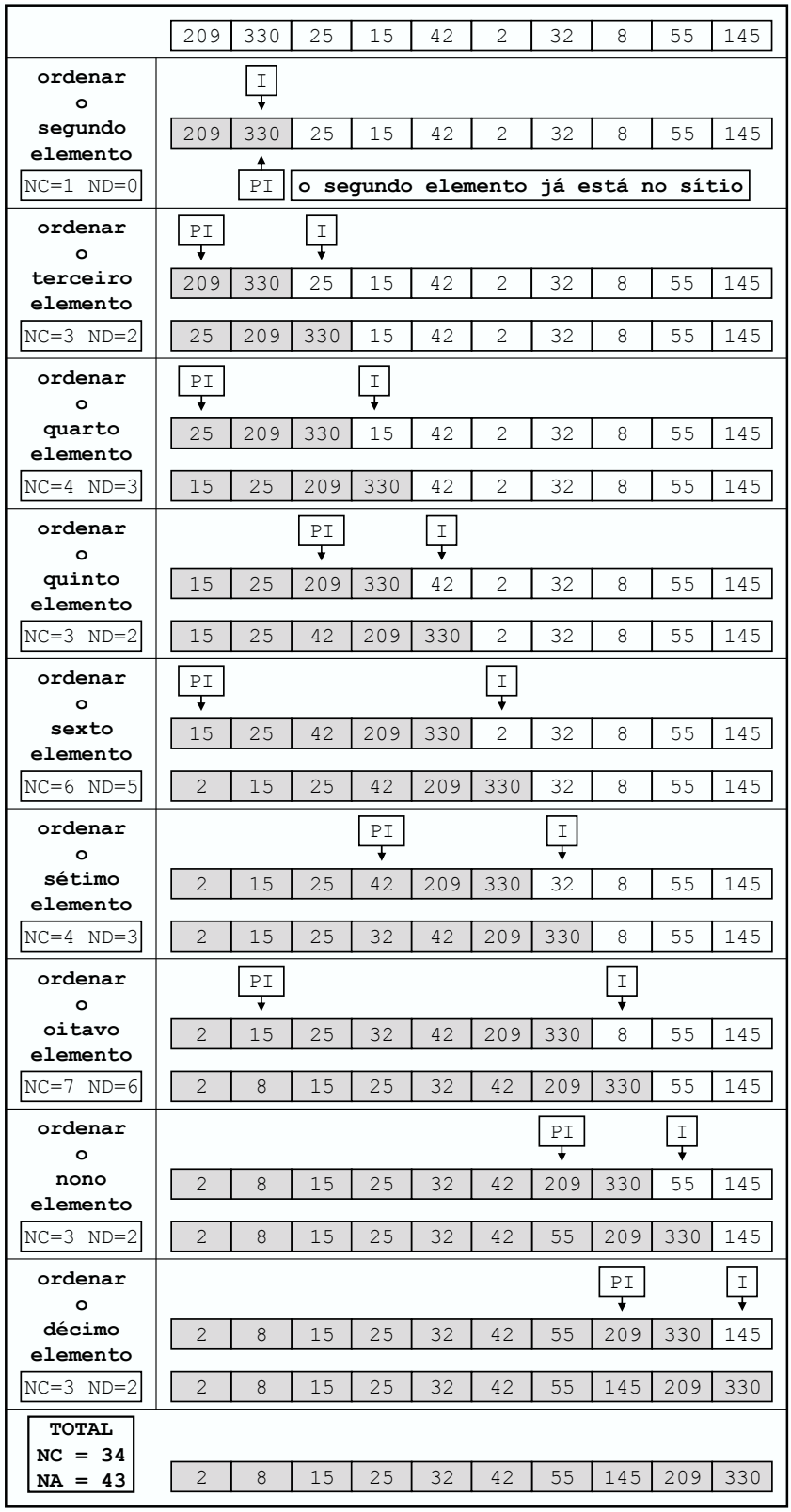

Figura 6.24 - Execução do algoritmo de ordenação Inserção.

Para um agregado com N elementos, este algoritmo faz no pior caso $(N^2–N)/2$ comparações e deslocamentos, mas, em termos médios faz aproximadamente $N^2/4$ comparações e deslocamentos, pelo que, pertence à classe $O(N^2)$. Se o agregado estiver parcialmente desordenado este é o algoritmo de ordenação mais indicado.

6.2.4 Comparação dos algoritmos de ordenação

A Tabela 6.1 apresenta os resultados dos algoritmos de ordenação apresentados. O pior algoritmo é o Sequencial, já que faz o número máximo de comparações necessárias para ordenar um agregado, e faz trocas às cegas, fazendo por vezes trocas que são feitas e refeitas vezes sem conta. O algoritmo Selecção permite minimizar o número de trocas e neste aspecto é o melhor de todos, compensando desta forma o excesso de comparações. O algoritmo Bolha tem um desempenho mau, a não ser quando o agregado está parcialmente desordenado, o que não é o caso do exemplo apresentado. A primeira versão do algoritmo Concha tem, para o exemplo utilizado, um número de comparações maior do que os outros algoritmos, se bem que com um número de trocas baixo. Mas, no caso da segunda versão, ele é o melhor algoritmo em número de comparações. Como o resultado deste algoritmo depende muito da série de incrementos utilizado, estes valores podem eventualmente ser melhorados com outra série de incrementos. O número de comparações só desce abaixo de $N^2/2$, para os algoritmos Concha (2ª versão), Crivo e Inserção. O primeiro tem um número de trocas aceitável, o segundo tem um número de trocas elevado, enquanto que o terceiro tem um número de trocas pior do que o Concha (1ª versão), mas que é aceitável se tivermos em conta a redução do número de comparações. Portanto, os algoritmos Inserção e Concha (2ª versão) são os melhores algoritmos analisados.

Algoritmo	Nº de Comparações	Nº de Trocas
Sequencial	45	25
Selecção	45	7
Bolha	45	25
Concha (1ª versão)	52	11
Concha (2ª versão)	29	≈18
Crivo	39	25
Inserção	34	≈14

Tabela 6.1 - Comparação dos algoritmos de ordenação.

Para a escolha da versão do algoritmo Concha, devemos optar pela primeira versão se a função de comparação dos elementos do agregado for simples, por exemplo uma comparação numérica e se os elementos do agregado forem estruturas pesadas, ou seja, com muitos *bytes*, com vista a minorar o número de trocas. Se pelo contrário, a função de comparação dos elementos do agregado for pesada, por exemplo uma comparação alfanumérica com muitos caracteres e se os elementos do agregado forem tipos simples, estruturas pequenas, ou ponteiros, então devemos optar pela segunda versão com vista a minorar o número de comparações. Estas considerações aplicam-se igualmente à escolha de qualquer algoritmo de ordenação.

Para a análise dos algoritmos devemos ter em conta que o pior caso na ordenação de um agregado acontece quando o agregado está invertido em relação à ordenação pretendida, enquanto que o melhor caso acontece quando o agregado está ordenado de acordo com a ordenação pretendida. Para avaliar o caso médio deve-se utilizar agregados gerados aleatoriamente e fazer uma estimativa média.

6.3 Ordenação por fusão

Uma forma de optimizar a ordenação de um agregado com muitos elementos, consiste em partir o agregado em vários agregados mais pequenos, ordená-los separadamente e depois fundi-los num agregado único. Para fundir dois ou mais agregados já ordenados, usa-se um algoritmo de ordenação por fusão. A Figura 6.25 apresenta o algoritmo de **Ordenação Fusão de Listas** (*Merge List Sort*) para a fusão de dois agregados ordenados por ordem crescente, considerando que eles não têm elementos repetidos. O agregado de saída tem que ter capacidade suficiente para armazenar os elementos dos dois agregados de entrada.

Começa-se por apontar para o primeiro elemento dos agregados a fundir, ou seja, para o índice 0. O índice do agregado de saída indk aponta sempre para a primeira posição livre do agregado, que inicialmente é a posição zero. Enquanto existirem elementos nos dois agregados, compara-se o elemento do agregado seqa, de índice indi, com o elemento do agregado seqb, de índice indj. O menor dos valores é copiado para o agregado de saída seqc, cujo índice indk é previamente incrementado para contabilizar mais um elemento. O índice do agregado, cujo elemento é copiado para o agregado de saída também é incrementado de uma posição, para sinalizar que o elemento já foi ordenado. Quando um dos agregados de entrada estiver esgotado, copiam-se os restantes elementos do outro agregado para o agregado de saída. No final da fusão, o indicador de elementos do agregado de saída nelemc, que é passado por referência, deve conter o número de elementos armazenados no agregado, que é igual a nelema mais nelemb, uma vez que estamos a considerar que não existem elementos repetidos nos agregados de entrada.

```
void Merge_List_Sort (int seqa[], unsigned int nelema,\
                      int seqb[], unsigned int nelemb,\
                      int seqc[], unsigned int *nelemc)
{
   unsigned int indi = 0, indj = 0, indk = 0, indc;

     /* copiar o elemento do agregado A ou o elemento do agregado B */
   while (indi < nelema && indj < nelemb)
      if (seqa[indi] < seqb[indj])
            seqc[indk++] = seqa[indi++];
      else  seqc[indk++] = seqb[indj++];

   if (indi < nelema)
        for (indc = indi; indc < nelema; indc++)
          seqc[indk++] = seqa[indc];
                     /* copiar os restantes elementos do agregado A */
   else for (indc = indj; indc < nelemb; indc++)
          seqc[indk++] = seqb[indc];
                     /* copiar os restantes elementos do agregado B */

   *nelemc = indk; /* armazenar o número de elementos do agregado C */
}
```

Figura 6.25 - Algoritmo de ordenação Fusão de Listas.

Este algoritmo é muito versátil e eficiente, uma vez que, pode ser generalizado para fazer a fusão de mais do que dois agregados. No caso da existência de elementos repetidos nos agregados de entrada, podemos considerar soluções alternativas, em que a ocorrência de elementos repetidos é eliminada ou mantida. Também pode ser utilizado como algoritmo de ordenação externa para fundir dois ou mais ficheiros. Para executar a fusão de dois agregados já ordenados, com N/2 elementos cada, este algoritmo faz no melhor caso N/2 comparações e no pior caso N–1 comparações.

A Figura 6.26 apresenta a execução do algoritmo na fusão de dois agregados previamente ordenados, com 5 elementos cada. O algoritmo faz 8 comparações, até que o agregado B fica esgotado, após o qual, os restantes elementos do agregado A são copiados para o agregado C.

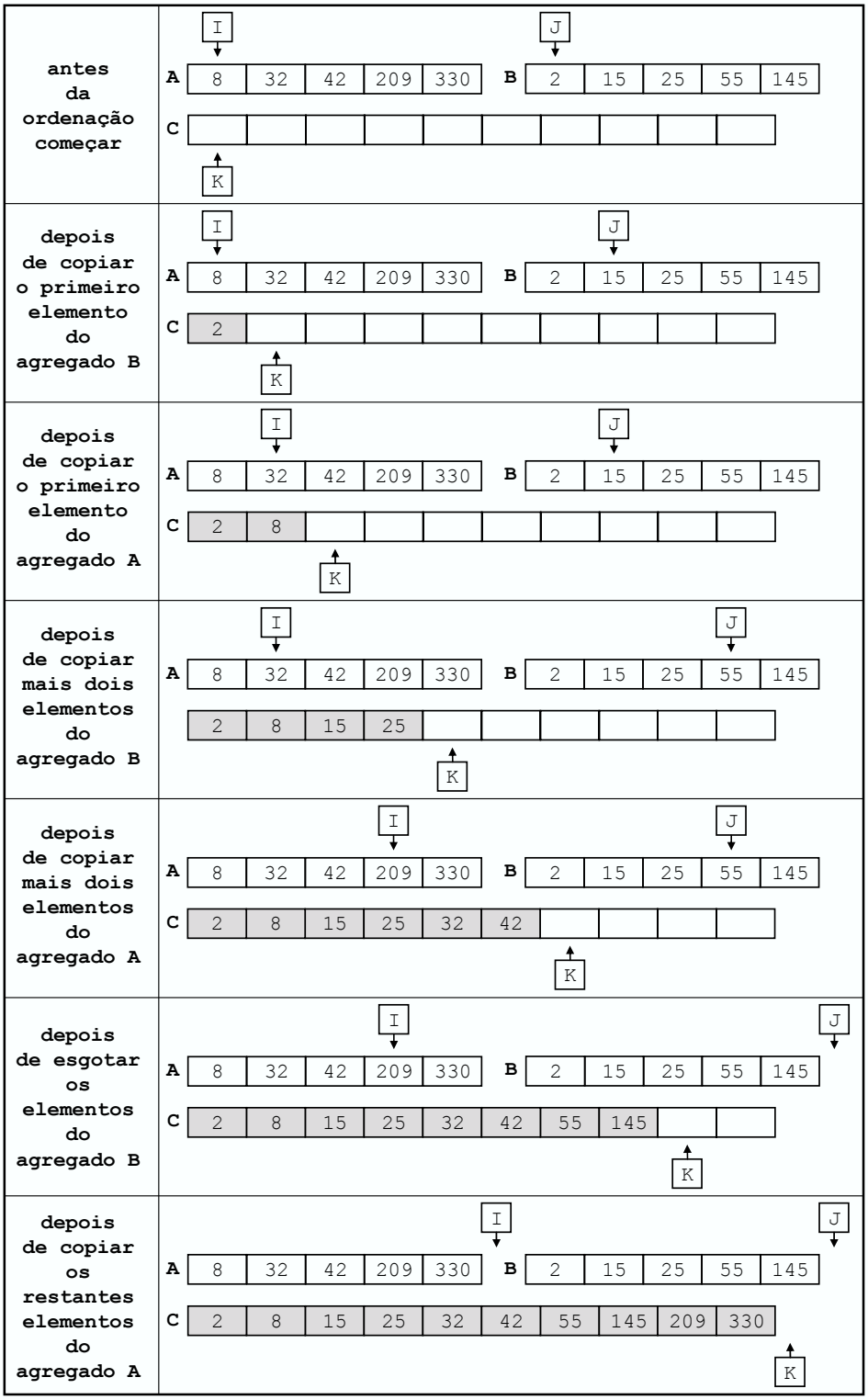

Figura 6.26 - Execução do algoritmo de ordenação Fusão de Listas.

6.4 Algoritmos de ordenação recursivos

Os algoritmos de ordenação recursivos são algoritmos de ordenação que aplicam o princípio do dividir para conquistar. São algoritmos complexos, mas muito eficientes e pertencem à classe $O(N \log_2 N)$. Vamos apresentar dois algoritmos.

6.4.1 Algoritmo de ordenação Fusão

O algoritmo de **Ordenação Fusão** (*Merge Sort*) pode ser utilizado de forma recursiva para ordenar um agregado, partindo-o sucessivamente ao meio e fazendo a fusão dos agregados parcelares já ordenados. Este princípio quando aplicado recursivamente, até que os agregados obtidos são apenas compostos por um único elemento, acaba por ordenar o agregado. Esta versão recursiva, para a ordenação crescente do agregado, é apresentada na Figura 6.27.

```
void Merge_Sort (int seq[], unsigned int inicio, unsigned int fim)
{
   unsigned int medio;

   if (inicio < fim)                          /* condição de paragem */
   {
      medio = (inicio + fim) / 2;            /* partição do agregado */

  /* invocação recursiva para ordenar a primeira metade do agregado */
      Merge_Sort (seq, inicio, medio);

   /* invocação recursiva para ordenar a segunda metade do agregado */
      Merge_Sort (seq, medio+1, fim);

                    /* fusão das duas metades ordenadas do agregado */
      Merge_List_Sort (seq, inicio, medio, fim);
   }
}
_____________________________________________________________________

void Merge_List_Sort (int seq[], unsigned int inicio,\
                      unsigned int medio, unsigned int fim)
{
   unsigned int inica = inicio, inicb = medio+1, indi = 0, indc;

                   /* atribuição de memória para o agregado local */
   int *seqtemp = (int *) calloc (fim-inicio+1, sizeof (int));

   while (inica <= medio && inicb <= fim)
      if (seq[inica] < seq[inicb])
           seqtemp[indi++] = seq[inica++];   /* elemento da 1ª parte */
      else seqtemp[indi++] = seq[inicb++];   /* elemento da 2ª parte */

     /* copiar os restantes elementos da primeira parte do agregado */
   while (inica <= medio) seqtemp[indi++] = seq[inica++];

      /* copiar os restantes elementos da segunda parte do agregado */
   while (inicb <= fim) seqtemp[indi++] = seq[inicb++];

                   /* copiar o resultado para o agregado a ordenar */
   for (indc = 0, inica = inicio; indc < indi; indc++, inica++)
      seq[inica] = seqtemp[indc];

   free (seqtemp);        /* libertação da memória do agregado local */
}
```

Figura 6.27 - Algoritmo de ordenação Fusão.

A invocação inicial da função de ordenação é **Merge_Sort (seq, 0, nelem-1);**. Cada invocação da função Merge_Sort precisa de saber o primeiro e o último elementos do agregado parcelar que está a ser ordenado, pelo que, o procedimento tem como parâmetros de entrada o início e o fim do agregado. A função Merge_List_Sort que constitui a parte não recursiva do algoritmo faz a fusão das duas partes já ordenadas para um agregado local e no fim copia-o de volta para o agregado a ordenar. É uma versão ligeiramente diferente do algoritmo apresentado na Figura 6.25, uma vez que faz a fusão de duas partes do mesmo agregado em vez de dois agregados distintos. Daí que tem como parâmetros de entrada os limites das duas partes do agregado, definidos pelos elementos inicial, médio e final do agregado. O agregado local é atribuído dinamicamente com o tamanho necessário para fazer a fusão das duas metades do agregado e libertado quando não é mais necessário. Para simplificar não é feita a validação da atribuição de memória.

Para comparar este algoritmo com os algoritmos que ordenam fazendo trocas dos elementos do agregado a ordenar, temos que ter em consideração, que para ordenar cada elemento é preciso copiá-lo de e para o agregado. Cada cópia custa uma instrução de atribuição, o que é equivalente, a um terço das instruções de atribuição que são necessárias para efectuar a troca de dois elementos do agregado. Ou seja, NT = NA/3.

A Figura 6.28 apresenta a execução do algoritmo para um agregado com 10 elementos. O algoritmo faz um total de 21 comparações para fundir os agregados parcelares e executa 68 instruções de atribuição, ou seja, cópias de valores entre o agregado a ordenar e o agregado local da função de fusão de listas, metade das quais se devem ao facto da necessidade de usar um agregado extra para fazer a fusão. Este número de cópias é equivalente a aproximadamente 23 trocas.

Quando comparado com os algoritmos, cujos resultados estão coligidos na Tabela 6.1, ele é o melhor algoritmo de ordenação no que diz respeito ao número de comparações. Tem, no entanto, um número elevado de trocas quando comparado com os algoritmos Inserção e Concha (2ª versão), que são os algoritmos com o melhor desempenho global de todos os algoritmos apresentados anteriormente.

Para executar a ordenação de um agregado com N elementos, sendo N múltiplo de 2, este algoritmo faz no melhor caso $(N \log_2 N)/2$ comparações e no pior caso $N \log_2 N - N + 1$ comparações, pelo que, pertence à classe $O(N \log_2 N)$.

6.4.2 Algoritmo de ordenação Rápido

O algoritmo de **Ordenação Rápido** (*Quick Sort*) foi criado em 1960 por C. A. R. Hoare e utiliza um principio muito simples que quando aplicado recursivamente, acaba por ordenar o agregado. O princípio é o seguinte. Vamos escolher um elemento do agregado, que vamos designar por pivot e dividir o agregado em duas partes. Na parte da esquerda colocam-se os valores menores do que o pivot e na parte direita colocam-se os valores maiores do que o pivot. Se cada parte do agregado for sucessivamente dividida ao meio e for aplicado este princípio de separação dos elementos, então quando o processo recursivo terminar ao se atingir agregados com três ou menos elementos, o agregado está ordenado por ordem crescente.

A Figura 6.29 apresenta o algoritmo para a ordenação crescente do agregado.

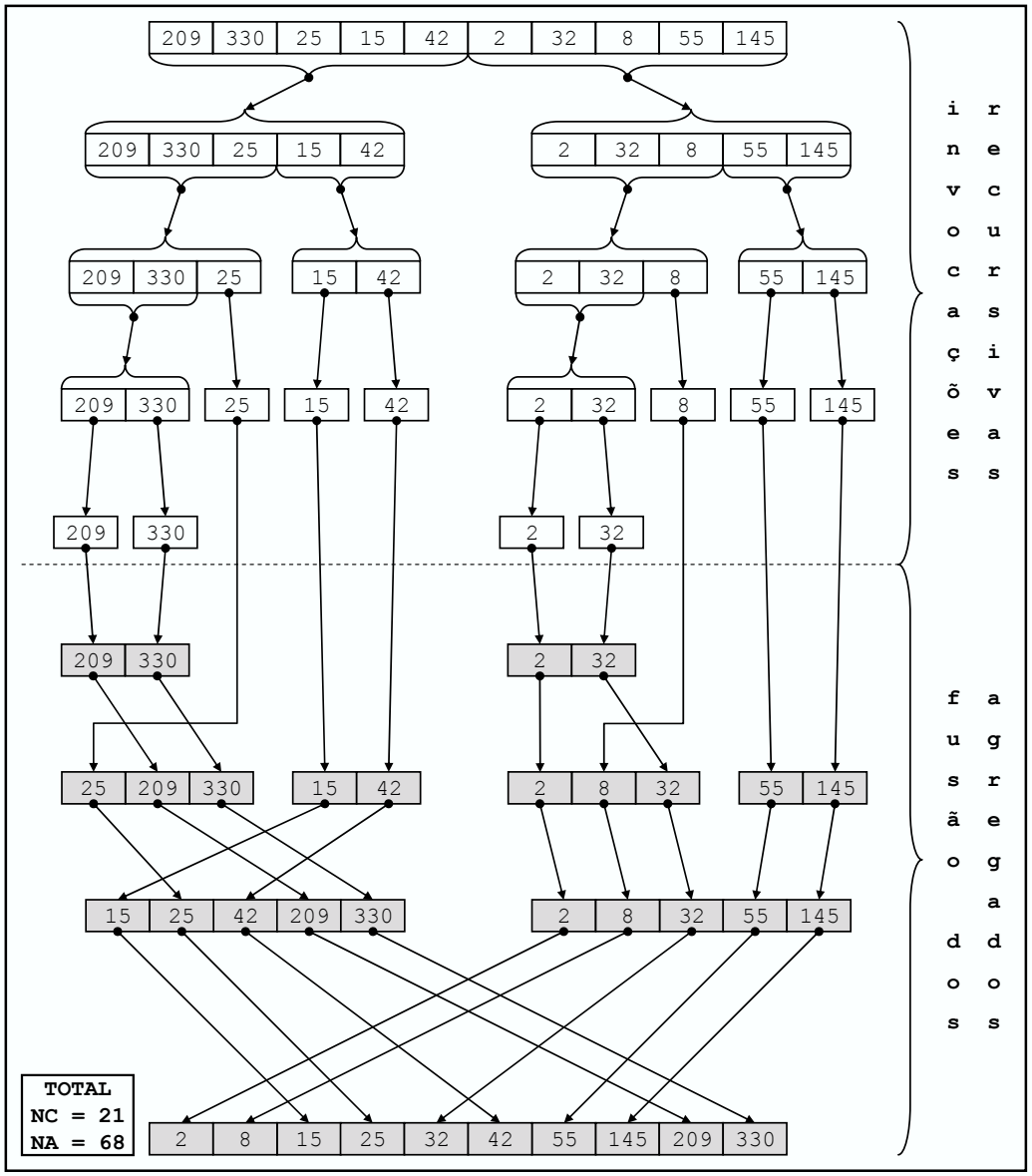

Figura 6.28 - Execução do algoritmo de ordenação Fusão.

Se o número de elementos do agregado for menor do que dois, então o agregado está automaticamente ordenado. Se existirem apenas dois elementos, eles são ordenados através de uma simples comparação e eventual troca dos dois elementos. Quando o agregado tem mais do que dois elementos, é escolhido o elemento de índice médio para servir de pivot. Os elementos das extremidades e o pivot são trocados para ficarem por ordem crescente. Se só existirem três elementos, então esta operação constituída por três comparações e eventuais três trocas, ordena o agregado. Senão, há que assegurar que todos os elementos à esquerda do pivot são menores do que ele e que todos os elementos à sua direita são maiores do que ele. Os elementos que se encontrem fora do sítio são trocados da parte direita para a parte esquerda e vice-versa, ou então são trocados com o pivot provocando que ele se desloque mais para a esquerda ou mais para a direita do que a posição média inicialmente calculada. Pelo que, quando o processamento se separação terminar não há garantia que o agregado está partido em duas metades com um número semelhante de elementos. Após este processamento, o algoritmo é invocado recursivamente para o agregado constituído pelos elementos à esquerda do pivot e para o agregado constituído

pelos elementos à direita do pivot. A invocação inicial da função de ordenação é **Quick_Sort (seq, 0, nelem-1);**.

```
void Quick_Sort (int seq[], unsigned int inicio, unsigned int fim)
{
  unsigned int medio, nelem = fim-inicio+1, indi, indj;

  if (nelem <= 1) return;   /* o agregado tem no máximo 1 elemento */
  if (nelem == 2)                 /* o agregado só tem 2 elementos */
  {
    if (seq[inicio] > seq[fim]) Swap (&seq[inicio], &seq[fim]);
    return;
  }

  medio = (inicio + fim) / 2;        /* cálculo do índice do pivot */

             /* colocar os extremos e o pivot por ordem crescente */
  if (seq[inicio] > seq[medio]) Swap (&seq[inicio], &seq[medio]);
  if (seq[inicio] > seq[fim]) Swap (&seq[inicio], &seq[fim]);
  if (seq[medio] > seq[fim]) Swap (&seq[medio], &seq[fim]);

       /* se o agregado só tem 3 elementos, então já está ordenado */
  if (nelem == 3) return;

  indi = inicio+1; indj = fim-1;

  while (indi < indj)
  {
      /* procurar elementos na parte esquerda maiores do que pivot */
    while (indi < medio)
      if (seq[indi] > seq[medio]) break;  else indi++;

      /* procurar elementos na parte direita menores do que pivot */
    while (indj > medio)
      if (seq[indj] < seq[medio]) break;  else indj--;

    if (indi != medio && indj != medio)
      Swap (&seq[indi], &seq[indj]);       /* trocar os elementos */

    else if (indi == medio && indj != medio)
         {         /* trocar o elemento de índice indj com o pivot */
                  /* e deslocar o pivot uma posição para a direita */
           Swap (&seq[medio++], &seq[indj]);
           Swap (&seq[medio], &seq[indj]);
           indi = medio;
         }

         else if (indi != medio && indj == medio)
              {    /* trocar o elemento de índice indi com o pivot */
                  /* e deslocar o pivot uma posição para a esquerda */
                Swap (&seq[medio--], &seq[indi]);
                Swap (&seq[medio], &seq[indi]);
                indj = medio;
              }
  }

          /* invocação recursiva para a parte esquerda do agregado */
  Quick_Sort (seq, inicio, medio-1);

           /* invocação recursiva para a parte direita do agregado */
  Quick_Sort (seq, medio+1, fim);
}
```

Figura 6.29 - Algoritmo de ordenação Rápido (1ª versão).

A Figura 6.30 apresenta a execução do algoritmo para um agregado com 10 elementos. Os elementos que servem de pivot, aparecem a cheio para se distinguirem dos restantes elementos e as invocações recursivas são identificadas pela sigla IR. O algoritmo faz um total de 28 comparações e de 19 trocas de elementos. Como exemplo, vamos ver como se comporta o algoritmo quando a função é invocada. Uma vez que o agregado inicial tem dez elementos, de índices 0 a 9, o elemento de índice 4, cujo valor é 42, é escolhido para pivot. Ao ordenar os valores 209, 42 e 145, o valor 145 passa a ser o novo pivot, porque é o valor mediano. Todos os valores menores do que 145 são colocados à sua esquerda, enquanto que, todos os valores maiores do que 145 são colocados à sua direita. Pelo que, o pivot vai ser deslocado para a posição de índice 7. Depois são feitas as invocações recursivas dos agregados formados pelos elementos 0 a 6 e pelos elementos 8 a 9.

Quando comparado com os algoritmos anteriores, constatamos que ele tem um desempenho praticamente semelhante ao Concha (2ª versão) e globalmente equivalente ao Fusão, com a vantagem de não necessitar de um agregado auxiliar. Quando comparado com o algoritmo de Inserção, ele tem um maior número de trocas compensado com um menor número de comparações, tendo no entanto uma implementação mais complexa.

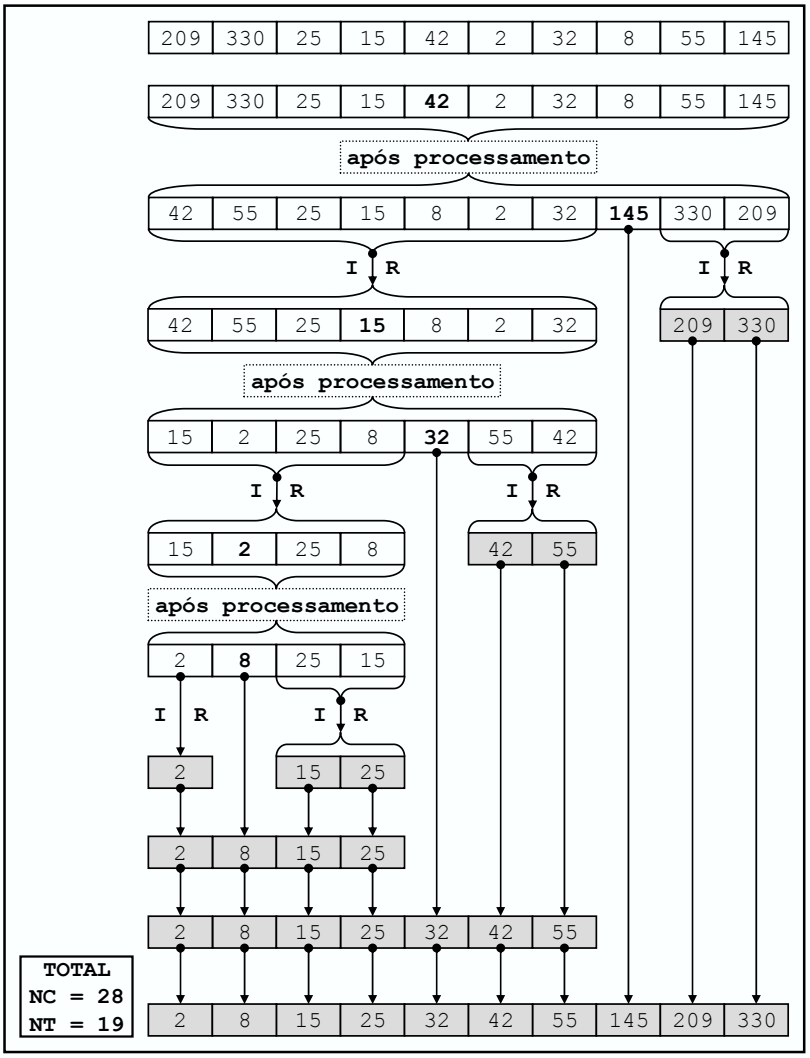

Figura 6.30 - Execução do algoritmo de ordenação Rápido.

Com o objectivo de diminuir o número de trocas, existe uma versão optimizada, que se apresenta na Figura 6.31 e que é frequentemente apresentada na literatura. A optimização consiste em evitar as trocas que envolvem o pivot e o deslocam da posição inicial. Após a escolha do pivot ele é escondido na penúltima posição do agregado. Os elementos são analisados e trocados até serem todos comparados com o pivot. Quando a análise do agregado terminar, ou seja, quando indi for maior ou igual do que indj, o pivot é colocado no sítio, por troca com o elemento que está na posição mais à esquerda do agregado que é maior do que ele, ou seja, com o elemento que está na posição indi. Esta versão do algoritmo faz um total de 30 comparações e de 17 trocas de elementos. Quando comparado com a primeira versão temos mais 2 comparações, mas menos 2 trocas.

```
void Quick_Sort (int seq[], unsigned int inicio, unsigned int fim)
{
  unsigned int medio, nelem = fim-inicio+1, indi, indj;

  if (nelem <= 1) return;   /* o agregado tem no máximo 1 elemento */
  if (nelem == 2)                 /* o agregado só tem 2 elementos */
  {
    if (seq[inicio] > seq[fim]) Swap (&seq[inicio], &seq[fim]);
    return;
  }

  medio = (inicio + fim) / 2;        /* cálculo do índice do pivot */

             /* colocar os extremos e o pivot por ordem crescente */
  if (seq[inicio] > seq[medio]) Swap (&seq[inicio], &seq[medio]);
  if (seq[inicio] > seq[fim]) Swap (&seq[inicio], &seq[fim]);
  if (seq[medio] > seq[fim]) Swap (&seq[medio], &seq[fim]);

       /* se o agregado só tem 3 elementos, então já está ordenado */
  if (nelem == 3) return;

             /* esconder o pivot na penúltima posição do agregado */
  Swap (&seq[medio], &seq[fim-1]);

  indi = inicio; medio = indj = fim-1;

  for ( ; ; )
  {
      /* procurar elementos na parte esquerda maiores do que pivot */
    while (seq[++indi] < seq[medio]) ;

      /* procurar elementos na parte direita menores do que pivot */
    while (seq[--indj] > seq[medio]) ;

    if (indi < indj) Swap (&seq[indi], &seq[indj]);  else break;
  }

            /* recuperar o pivot para a posição média do agregado */
  medio = indi;
  Swap (&seq[medio], &seq[fim-1]);

         /* invocação recursiva para a parte esquerda do agregado */
  Quick_Sort (seq, inicio, medio-1);

          /* invocação recursiva para a parte direita do agregado */
  Quick_Sort (seq, medio+1, fim);
}
```

Figura 6.31 - Algoritmo de ordenação Rápido (2ª versão).

Para executar a ordenação de um agregado com N elementos, este algoritmo faz no pior caso $(N^2–N)/2$ comparações. No entanto, no caso médio faz aproximadamente $1.4\times(N+1)\log_2 N$ comparações, pelo que, considera-se que pertence à classe $O(N \log_2 N)$.

6.5 Generalização dos algoritmos de ordenação

Os algoritmos de ordenação foram apresentados e aplicados sobre agregados de números inteiros, com vista a melhor visualizar o seu funcionamento. No entanto, em muitas aplicações há a necessidade de armazenar bases de dados em agregados e de as ordenar de acordo com as diferentes características dos seus elementos. Por vezes existe mesmo a necessidade de ordenar sucessivamente um agregado de registos por diversas chaves de ordenação diferente, como por exemplo, fazer ordenações alfabéticas, cronológicas e numéricas e fazer ordenações crescentes ou ascendentes e decrescentes ou descendentes.

Como podemos então implementar estas ordenações recorrendo ao mesmo algoritmo de ordenação? E de preferência com o mínimo de esforço. Ou seja, como podemos generalizar um algoritmo de ordenação para que ele possa ordenar um agregado com elementos de um qualquer tipo de dados, ordenar por diferentes critérios e ordenar por ordem crescente ou decrescente. Para poder generalizar um algoritmo de ordenação e de modo a estruturar melhor a solução, precisamos de um ficheiro de interface que defina o tipo de elementos constituintes do agregado e que providencie as funções de comparação de acordo com os diferentes critérios de ordenação que se pretendem efectuar.

Vamos utilizar o exemplo concreto que se apresenta na Figura 6.32. Vamos considerar que se pretende gerir uma base de dados constituída por elementos com a seguinte informação. O número de registo de uma pessoa na base de dados que é do tipo inteiro, um nome que é uma cadeia de caracteres e uma data constituída por dia, mês e ano. Vamos considerar ainda que pretendemos ordenar a base de dados por ordem numérica do número de registo, por ordem alfabética do nome e por ordem cronológica da data. Assim sendo, precisamos de implementar as três funções de comparação CompNRegisto, CompNome e CompData.

Para implementar no mesmo algoritmo de ordenação, a ordenação crescente ou ascendente e decrescente ou descendente, vamos passar à função de ordenação o parâmetro de entrada adicional inteiro tord que representa o tipo de ordenação a efectuar. Vamos definir que este parâmetro indica que se pretende a ordenação crescente com o valor 1 e indica que se pretende a ordenação decrescente com o valor −1.

Pelo que, as funções de comparação têm a particularidade de devolver um valor inteiro que é um dos seguintes três valores: o valor 0 no caso dos elementos serem iguais conforme o item comparado; o valor 1 no caso do primeiro elemento ser maior do que segundo elemento conforme o item comparado, ou seja, um maior número de registo ou um nome alfabeticamente posterior ou uma data maior, o que significa uma data mais recente; e o valor −1 no caso do primeiro elemento ser menor do que segundo elemento conforme o item comparado, ou seja, um menor número de registo ou um nome alfabeticamente anterior ou uma data menor o que significa uma data mais antiga.

Finalmente, para poder generalizar os algoritmos de ordenação precisamos de um mecanismo que permita utilizar uma qualquer função de comparação dentro da função de ordenação, de acordo com a ordenação pretendida num dado momento do programa. Para tal, a linguagem C providencia os ponteiros para funções. Um ponteiro para uma função permite invocar de uma forma simples e elegante diferentes funções. A seguinte instrução define um ponteiro para uma função inteira com dois parâmetros de entrada do tipo TElem, que é reconhecido pelo identificador PtFComp.

```
typedef int (*PtFComp) (TElem, TElem);
```

```
 /********** Interface da Estrutura de Dados do Agregado **********/
                        /* Nome: elemento.h */

/* Definição do tipo dos elementos do agregado e das funções de
comparação necessárias. Este ficheiro deve ser modificado para
adequar a definição e as funções a cada implementação específica.  */

#include <string.h>

#ifndef _ELEMENTO
#define _ELEMENTO

 /************* Definição do Tipo de Dados do Elemento *************/

typedef struct
{
   unsigned int dia;
   unsigned int mes;
   unsigned int ano;
} TData;

typedef struct
{
   unsigned int nreg;
   char nome[60];
   TData data;
} TElem;

 /***** Definição do Tipo Ponteiro para a Função de Comparação *****/

typedef int (*PtFComp) (TElem, TElem);

 /******* Definição das Funções de Comparação dos Elementos *******/

int CompNRegisto (TElem a, TElem b)
{
   if ( a.nreg > b.nreg ) return 1;
   else if ( a.nreg < b.nreg ) return -1;
        else return 0;
}

int CompNome (TElem a, TElem b)
{
   int comp = strcmp (a.nome, b.nome);

   if ( comp > 0 ) return 1;
   else if ( comp < 0 ) return -1;
        else return 0;
}

int CompData (TElem a, TElem b)
{
   if ( a.data.ano > b.data.ano ) return 1;
   else if ( a.data.ano < b.data.ano) return -1;
        else if ( a.data.mes > b.data.mes ) return 1;
             else if (a.data.mes < b.data.mes ) return -1;
                  else if ( a.data.dia > b.data.dia ) return 1;
                       else if ( a.data.dia < b.data.dia ) return -1;
                            else return 0;
}

#endif
```

Figura 6.32 - Ficheiro de interface do elemento constituinte do agregado a ordenar.

A partir desta definição podem-se declarar variáveis do tipo PtFComp e atribuir-lhe uma qualquer função, desde que seja uma função inteira com dois parâmetros de entrada do tipo TElem. A Figura 6.34 apresenta um exemplo da passagem de um parâmetro de entrada deste tipo para uma função de ordenação. A Figura 6.35 apresenta a utilização de uma variável deste tipo para ser colocada a apontar para diferentes funções ao longo da execução do programa, de forma a parametrizar a ordenação de um agregado. Tal como nos agregados, o nome de uma função também é um ponteiro para a função, pelo que, a instrução de atribuição **fcomp = CompNome;** coloca o ponteiro para função fcomp a apontar para a função CompNome.

A função de troca de elementos do agregado que foi apresentada na Figura 6.11, tem de ser alterada de maneira a poder trocar dois elementos do tipo TElem. A Figura 6.33 apresenta a nova versão que vamos designar por SwapElementos.

```
void SwapElementos (TElem *x, TElem *y)
{
   TElem temp;
   temp = *x; *x = *y; *y = temp;
}
```

Figura 6.33 - Função para trocar dois elementos de um agregado de elementos do tipo TElem.

A Figura 6.34 apresenta a versão generalizada do algoritmo de ordenação Sequencial. Escolhemos este algoritmo, apenas porque é o mais simples em termos de código. O primeiro parâmetro da função é o parâmetro de entrada-saída que representa o agregado a ordenar, o segundo parâmetro é o parâmetro de entrada que representa o número de elementos do agregado, o terceiro parâmetro é o parâmetro de entrada que representa a função de comparação e o quarto parâmetro é o parâmetro de entrada que representa o tipo de ordenação a efectuar. Estes dois últimos parâmetros de entrada configuram o algoritmo de ordenação, tornando-o assim genérico e reutilizável.

```
void Sequential_Sort (TElem seq[], unsigned int nelem,\
                      PtFComp fcomp, int tord)
{
   unsigned int indi, indj;
   for (indi = 0; indi < nelem-1; indi++)
      for (indj = indi+1; indj < nelem; indj++)
         if ( fcomp (seq[indi], seq[indj]) == tord )
            SwapElementos (&seq[indi], &seq[indj]);
}
```

Figura 6.34 - Algoritmo de ordenação Sequencial generalizado.

A Figura 6.35 apresenta a utilização sucessiva deste algoritmo de ordenação, para ordenar um agregado de elementos do tipo TElem. Antes de cada invocação, é atribuído ao ponteiro fcomp a função de comparação necessária para obter a ordenação desejada e depois a função de ordenação é invocada indicando também o tipo de ordenação pretendido. Para aumentar a legibilidade do programa, definimos as constantes simbólicas CRESCENTE e DECRESCENTE.

Como exercício de treino escreva a função Display, cuja funcionalidade é imprimir no monitor a informação relativa a um agregado de estruturas do tipo TElem, com o protótipo Display (TElem seq[], unsigned int nelem); e teste o programa.

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

#include "elemento.h"           /* caracterização do tipo elemento */

#define CRESCENTE 1
#define DECRESCENTE -1

void SwapElementos (TElem *, TElem *);
void Sequential_Sort (TElem [], unsigned int, PtFComp, int);
void Display (TElem [], unsigned int);

int main (void)
{
   TElem pintores[] = {
                        { 1, "Vincent Van Gogh", {30, 3, 1853} },
                        { 2, "Vieira da Silva", {13, 6, 1908} },
                        { 3, "Amedeo Modigliani", {12, 7, 1884} },
                        { 4, "Claude Monet", {14, 11, 1840} },
                        { 5, "Georgia O'Keeffe", {15, 11, 1887} }
                      };

   int nelem = sizeof (pintores) / sizeof (pintores[0]);

        /* ponteiro para a função de comparação inicializado a NULL */
   PtFComp fcomp = NULL;

   fcomp = CompNome;             /* ordenação alfabética ascendente */
   Sequential_Sort (pintores, nelem, fcomp, CRESCENTE);
   printf ("Ordenação Alfabética Ascendente\n");
   Display (pintores, nelem);

   fcomp = CompData;           /* ordenação cronológica decrescente */
   Sequential_Sort (pintores, nelem, fcomp, DECRESCENTE);
   printf ("Ordenação Cronológica Decrescente\n");
   Display (pintores, nelem);

   fcomp = CompNRegisto;         /* ordenação crescente por registo */
   Sequential_Sort (pintores, nelem, fcomp, CRESCENTE);
   printf ("Ordenação Numérica Crescente\n");
   Display (pintores, nelem);

   return EXIT_SUCCESS;
}
...                                        /* Definição das funções */
```

Figura 6.35 - Exemplo da utilização sucessiva do algoritmo de ordenação Sequencial.

6.6 Avaliação do desempenho dos algoritmos

Para podermos avaliar o desempenho dos algoritmos de ordenação é necessário contabilizar o número de operações de comparação e o número de operações de trocas de elementos. Ou, uma vez que existem algoritmos de ordenação que em vez de trocas fazem cópias de elementos, contabilizar em alternativa o número de instruções de atribuição, de maneira a ter uma métrica uniforme.

Para calcular o número de operações efectuadas em invocações sucessivas das funções de troca e de comparação de elementos, utilizam-se variáveis contadoras de duração permanente, ou seja, variáveis que são declaradas com o qualificativo **static**. Como a função de ordenação pode ser repetidamente invocada, então é conveniente poder reiniciar a contagem, assim como é necessário reportar o resultado final da contagem. Pelo que, as

funções têm o parâmetro de entrada modo de tipo inteiro, que indica o modo de actuação sobre a variável contadora.

De maneira a aumentar a legibilidade das funções é aconselhável utilizar constantes simbólicas que representam o modo de actuação sobre a variável contadora. Temos as três constantes simbólicas seguintes: REP para reportar o valor da variável contadora; INIC para inicializar a variável contadora; e NORM para realizar a operação da função e para incrementar o valor da variável contadora.

Para calcular o número de instruções de atribuição utiliza-se uma variável de duração permanente na função SwapCount, tal como se mostra na Figura 6.36. No modo NORM a função troca os dois elementos e incrementa o valor da variável contadora de três unidades, para contabilizar as três instruções de atribuição.

```
unsigned int SwapCount (TElem *x, TElem *y, int modo)
{
   static unsigned int cont;                    /* variável contadora */
   TElem temp;

   if (modo == REP) return cont;
   else if (modo == INIC) cont = 0;
        else if (modo == NORM)
             {
               temp = *x; *x = *y; *y = temp;    /* efectuar a troca */
               cont += 3;/* contagem das 3 instruções de atribuição */
             }

   return 0;
}
```

Figura 6.36 - Função para trocar elementos do agregado com contabilização de atribuições.

Para calcular o número de comparações utiliza-se uma variável de duração permanente em conjunção com a função de comparação pretendida. Assim encapsula-se a função de comparação dentro de uma nova função CCount, tal como se mostra na Figura 6.37, que além de efectuar a comparação pretendida, usando para o efeito o ponteiro para a função de comparação, também contabiliza o número de vezes que é invocada. Os elementos a comparar são passados à função por referência, ou seja, através de ponteiros. No modo NORM a função compara os dois elementos e incrementa a variável contadora uma unidade, para contabilizar mais uma comparação.

```
int CCount (TElem *x, TElem *y, PtFComp fcomp, int modo)
{
   static unsigned int cont;                    /* variável contadora */

   if (modo == REP) return cont;
   else if (modo == INIC)
        {
          cont = 0; return 0;
        }
        else if (modo == NORM)
             {
               cont++;      /* contagem de 1 instrução de comparação */
               return fcomp (*x, *y);       /* efectuar a comparação */
             }
}
```

Figura 6.37 - Função para comparar elementos do agregado com contabilização de comparações.

A Figura 6.38 apresenta a versão generalizada e com contabilização de instruções do algoritmo de ordenação Sequencial. Antes da ordenação começar, as funções SwapCount e CCount são invocadas para inicializar as variáveis contadoras. Durante a ordenação as funções são invocadas no modo normal para desempenharem a sua tarefa. Depois da ordenação terminar, as funções são invocadas para reportar o número de comparações e de instruções de atribuição efectuadas durante a ordenação, que são armazenados nos parâmetros de saída nc e na respectivamente. Quando as funções são invocadas para inicializar ou reportar a variável contadora, os restantes parâmetros são passados como sendo ponteiros nulos.

```
void Sequential_Sort (TElem seq[], unsigned int nelem,\
                      PtFComp fcomp, int tord, int *nc, int *na)
{
   unsigned int indi, indj;

                          /* inicialização das variáveis contadoras */
   CCount ((TElem *) NULL, (TElem *) NULL, (PtFComp) NULL, INIC);
   SwapCount ((TElem *) NULL, (TElem *) NULL, INIC);

         /* execução da ordenação com contabilização das operações */
   for (indi = 0; indi < nelem-1; indi++)
      for (indj = indi+1; indj < nelem; indj++)
         if (CCount (&seq[indi], &seq[indj], fcomp, NORM) == tord )
            SwapCount (&seq[indi], &seq[indj], NORM);

                              /* relatório das variáveis contadoras */
   *nc = CCount ((TElem *) NULL, (TElem *) NULL, (PtFComp) NULL, REP);
   *na = SwapCount ((TElem *) NULL, (TElem *) NULL, REP);
}
```

Figura 6.38 - Algoritmo de ordenação Sequencial generalizado e com contabilização de operações.

Como exercício de treino altere o programa apresentado na Figura 6.35 de modo a utilizar estas novas versões das funções SwapCount, CCount e Sequential_Sort. Acrescente ainda ao programa a impressão no monitor do número de comparações e do número de trocas, que são um terço das instruções de atribuição, depois de cada ordenação.

6.7 Exercícios

1. Pretende-se escrever uma função de ordenação Fusão de Listas que contemple a situação da existência de elementos repetidos nos agregados de entrada, e que nessa situação copie apenas um dos elementos repetidos para o agregado de saída.

6.8 Leituras recomendadas

- 13º capítulo do livro "Data Structures, Algorithms and Software Principles in C", de Thomas A: Standish, da editora Addison-Wesley Publishing Company, 1995.

- 7º capítulo do livro "Data Structures and Algorithm Analysis in C", 2ª edição, de Mark Allen Weiss, da editora Addison-Wesley Publishing Company, 1997.

Capítulo 7

FILAS E PILHAS

Sumário

Este capítulo é dedicado às estruturas de dados lineares que são as filas e as pilhas. Apresentamos as implementações estática e semiestática baseadas em agregados e a implementação dinâmica baseada em listas ligadas. Mostramos exemplos de aplicação que utilizam filas e pilhas como elementos de armazenamento e que tiram partido da sua organização interna para a resolução de problemas. Finalmente apresentamos uma implementação abstracta, dinâmica e com capacidade de múltipla instanciação.

7.1 Introdução

As filas e as pilhas são estruturas de dados lineares que têm implementações em tudo semelhantes, sendo que diferem apenas na prioridade da retirada de informação. Enquanto que uma fila implementa a política do primeiro a chegar primeiro a sair (*first in first out*), a pilha implementa a política do último a chegar primeiro a sair (*last in first out*).

As filas são muito usadas nos sistemas operativos para funcionarem como áreas de armazenamento de informação que deve ser processada por ordem de chegada, como por exemplo, o atendimento de pedidos de utilização de uma impressora de rede que recebe ficheiros para imprimir, enviados por diferentes computadores instalados na rede. Para sincronizar a interacção entre processos concorrentes que executam a diferentes velocidades de processamento. São também utilizadas para simular modelos que descrevem o comportamento de situações reais de atendimento de pedidos que são processados por ordem de chegada, como por exemplo, simular o atendimento de uma fila de espera num qualquer serviço do dia a dia.

As pilhas são usadas para gerir algoritmos em que existem processos que invocam subprocessos do mesmo tipo, como é o caso dos algoritmos recursivos. Para processar estruturas imbricadas, ou seja, estruturas que contém outras estruturas do mesmo tipo dentro delas, como por exemplo, expressões aritméticas que são compostas por subexpressões aritméticas do mesmo tipo. Assim uma pilha pode ser usada para verificar o balanceamento dos parênteses numa expressão aritmética, que pode conter vários níveis de parênteses, que podem ser de tipos diferentes como parênteses curvos, rectos e chavetas. Podem ser usadas para fazer a avaliação de expressões, que contém expressões internas que têm de ser avaliadas previamente, antes do cálculo da expressão final. Por exemplo, para calcular uma expressão em notação polaca, que utiliza os símbolos das operações depois dos operadores. Para fazer a análise sintáctica durante a compilação de um programa, onde existem ciclos repetitivos dentro de ciclos repetitivos, blocos dentro de blocos e portanto, existem estruturas imbricadas que têm de estar balanceadas e correctamente imbricadas.

Vamos apresentar as implementações estática, semiestática e dinâmica de filas e de pilhas, considerando que estamos perante estruturas de dados concretas, cujo tipo dos elementos de armazenamento é concretizado pelo utilizador através de um ficheiro de interface, que vamos designar por elemento.h.

Este ficheiro de interface, que se apresenta na Figura 7.1, define a constante que parametriza a dimensão da estrutura de dados de suporte, necessário apenas no caso das implementações estática e semiestática, bem como o tipo de dados do elemento constituinte da memória. No caso da implementação dinâmica, este ficheiro precisa apenas de definir o tipo de dados do elemento constituinte da memória. Assim o utilizador do módulo, pode concretizá-lo para uma estrutura de dados que corresponda às suas necessidades, sem ter a necessidade de reprogramar o ficheiro de implementação do módulo. Em relação à criação de um módulo abstracto, esta solução exige a recompilação do módulo, sempre que este ficheiro é modificado.

```
 /*********** Interface da Estrutura de Dados do Módulo ***********/
                        /* Nome: elemento.h */

/* Definição da dimensão da estrutura de dados para o caso das
implementações estática e semiestática e definição do tipo de dados
dos seus elementos. Este ficheiro deve ser modificado para adequar a
definição a cada implementação específica.                          */

#ifndef _ELEMENTO
#define _ELEMENTO

 /****** Constantes de Parametrização das Estruturas de Dados ******/

#define  N_ELEMENTOS  100       /* número de elementos do agregado */

 /************* Definição do Tipo de Dados do Elemento *************/

typedef  ...  TElem;                /* tipo de dados dos elementos */

#endif
```

Figura 7.1 - Ficheiro de interface do elemento constituinte da memória.

7.2 Filas

Uma memória fila (***queue*/*FIFO***) é uma memória em que só é possível processar a informação pela ordem de chegada, daí que, também seja apelidada de memória do primeiro a chegar primeiro a sair. Numa memória fila, o posicionamento para a colocação de um novo elemento na fila, que vamos designar por **Fifo_In**, é a cauda da fila (***fifo tail***), e o posicionamento para a remoção de um elemento da fila, que vamos designar por **Fifo_Out**, é a cabeça da fila (***fifo head***).

7.2.1 Implementação estática

A Figura 7.2 apresenta o ficheiro de interface da implementação estática de uma fila, que é baseada num agregado de elementos usado de forma circular. Os indicadores de cabeça e cauda da fila são variáveis de tipo inteiro positivo.

Como um agregado tem uma dimensão fixa, antes de se colocar um elemento na fila é necessário verificar se ela está cheia. Em caso afirmativo, mais nenhum elemento pode ser colocado na fila e é assinalada a situação de erro, usando o código de erro FIFO_FULL. O elemento que se pretende copiar para a fila é passado à função **Fifo_In** por valor. Por outro lado, o elemento que vai receber a cópia do elemento que se pretende retirar da fila é passado à função **Fifo_Out** por referência. Pode acontecer que o ponteiro passado à função seja um ponteiro nulo. Nestas circunstâncias a função não pode retirar o elemento da fila, pelo que, não faz nada e assinala esta anomalia, usando o código de erro NULL_PTR. Antes de se retirar um elemento da fila é preciso detectar se ela está vazia. Em caso afirmativo, nenhum elemento pode ser retirado da fila e é assinalada a situação de erro, usando o código de erro FIFO_EMPTY. Sempre que é colocado ou retirado um elemento da fila é devolvido o código OK sinalizando que a operação foi realizada com sucesso.

De seguida vamos apresentar graficamente o comportamento das operações de colocação de um elemento na fila e de remoção de um elemento da fila.

```
/******************* Interface da FILA Estática ******************/
                        /* Nome : fila_est.h */

#ifndef _FILA_ESTATICA
#define _FILA_ESTATICA

#include "elemento.h"   /* caracterização do tipo elemento da fila */

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define FIFO_EMPTY  4   /* fila vazia */
#define FIFO_FULL   5   /* fila cheia */

            /* Alusão às Funções Exportadas pelo Módulo */

int Fifo_In (TElem elemento);
/* Coloca o elemento elemento na cauda da fila. Valores de retorno:
OK ou FIFO_FULL. */

int Fifo_Out (TElem *pelemento);
/* Retira o elemento da cabeça da fila para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou FIFO_EMPTY. */

#endif
```

Figura 7.2 - Ficheiro de interface da fila estática.

A Figura 7.3 mostra o estado inicial da fila. Por uma questão de implementação, vamos considerar que a cauda da fila indica sempre a primeira posição livre para a próxima operação de colocação de um elemento na fila, pelo que, inicialmente aponta para a posição 0 do agregado, enquanto que a cabeça da fila indica sempre a posição da próxima operação de remoção de um elemento da fila, pelo que, inicialmente aponta para a posição N do agregado, ou seja, a posição depois do fim do agregado, como indicação de fila vazia. A Figura 7.3 mostra também a colocação do primeiro elemento na fila e o estado após a operação. A cabeça da fila vai ficar a apontar para o elemento acabado de colocar, que é a posição 0 do agregado, e, fica nesta posição enquanto este elemento não for retirado da fila. A cauda da fila é deslocada para a posição 1 do agregado, que é a primeira posição livre para colocar o próximo elemento.

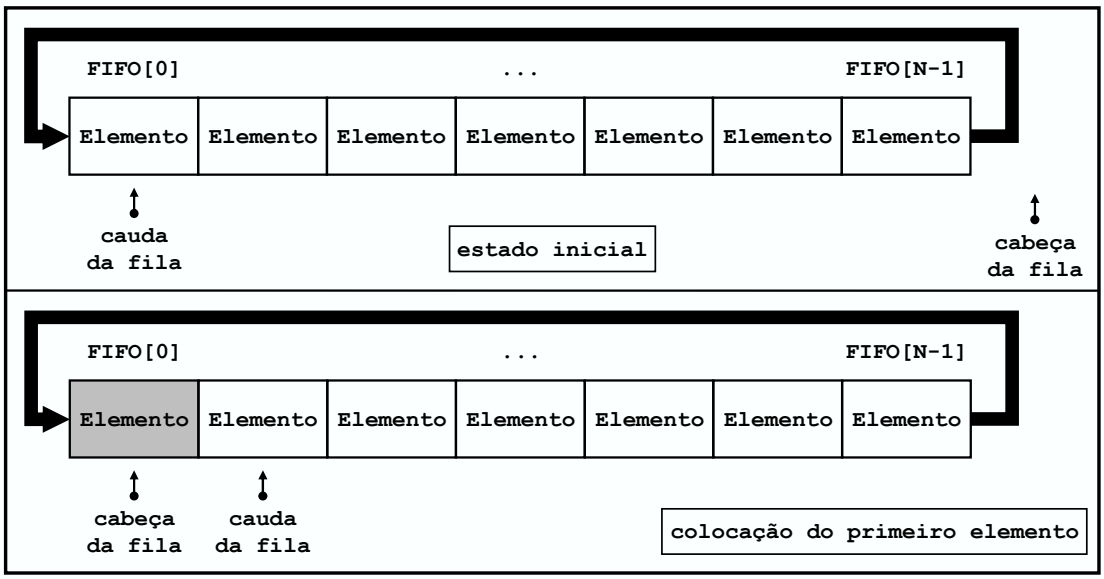

Figura 7.3 - Situação inicial da fila e após a colocação do primeiro elemento.

Sempre que se coloca um elemento na fila, a cauda da fila é deslocada para o elemento seguinte da fila. Existem duas situações distintas. Na primeira situação, que se apresenta na Figura 7.4, a posição seguinte da fila está ainda dentro da dimensão do agregado, pelo que, a cauda continua atrás da cabeça.

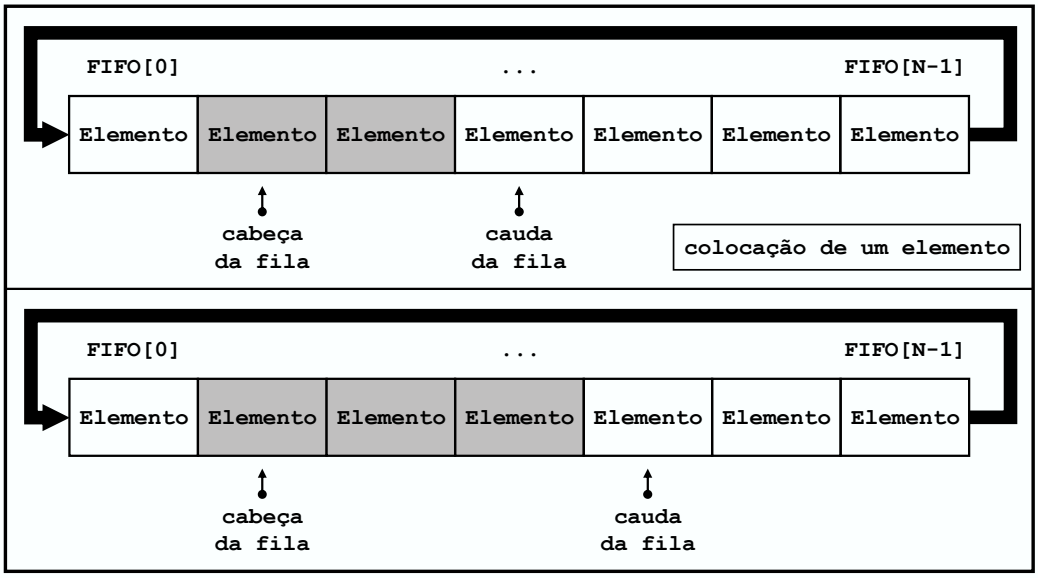

Figura 7.4 - Colocação de um elemento na fila.

Na segunda situação, que se apresenta na Figura 7.5, a posição seguinte da fila está fora da dimensão do agregado, pelo que, a cauda da fila passa para o início do agregado e fica à frente da cabeça. O que parece uma situação anómala quando comparada com o funcionamento de uma fila de espera no dia a dia, em que a cabeça está sempre à frente da cauda. Mas, é preciso ter em consideração que estamos perante uma fila circular. Para conseguir este efeito circular da cauda da fila utiliza-se o operador módulo, para calcular sempre um valor que está entre 0 e N_ELEMENTOS−1. Em alternativa à utilização circular do agregado, a fila poderia ser implementada de forma linear, o que implicaria deslocar todos os elementos da fila, para o início do agregado, sempre que se retirasse um elemento da fila. Mas, tal implementação seria muito ineficiente para filas grandes.

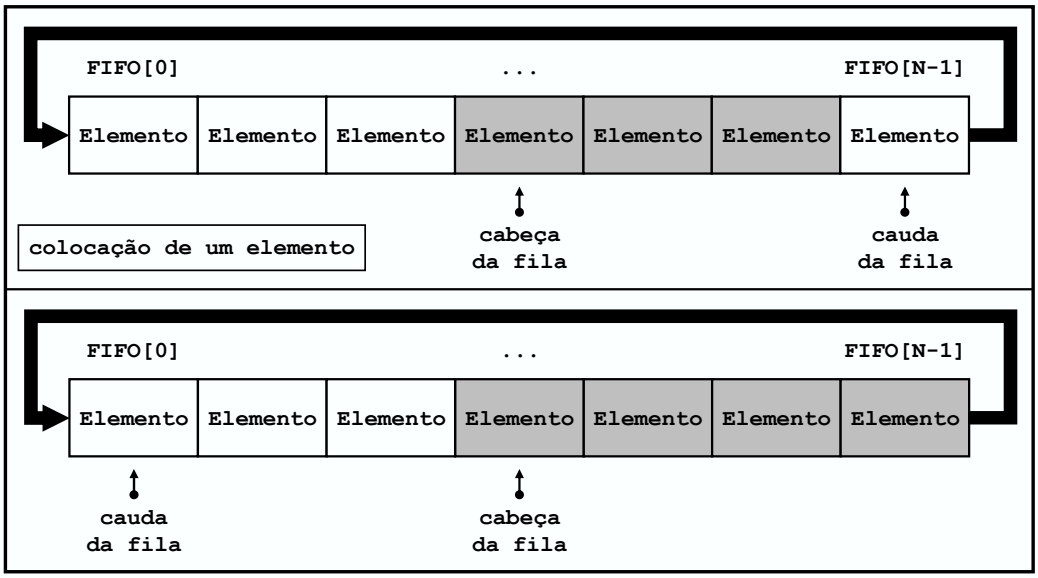

Figura 7.5 - Colocação de um elemento na fila com movimentação circular.

A Figura 7.6 apresenta a situação de colocação do último elemento na fila. Nesta situação a cauda da fila fica a apontar para o mesmo elemento que a cabeça da fila, o que significa que a fila ficou cheia. Enquanto este estado durar, não é possível colocar mais elementos na fila.

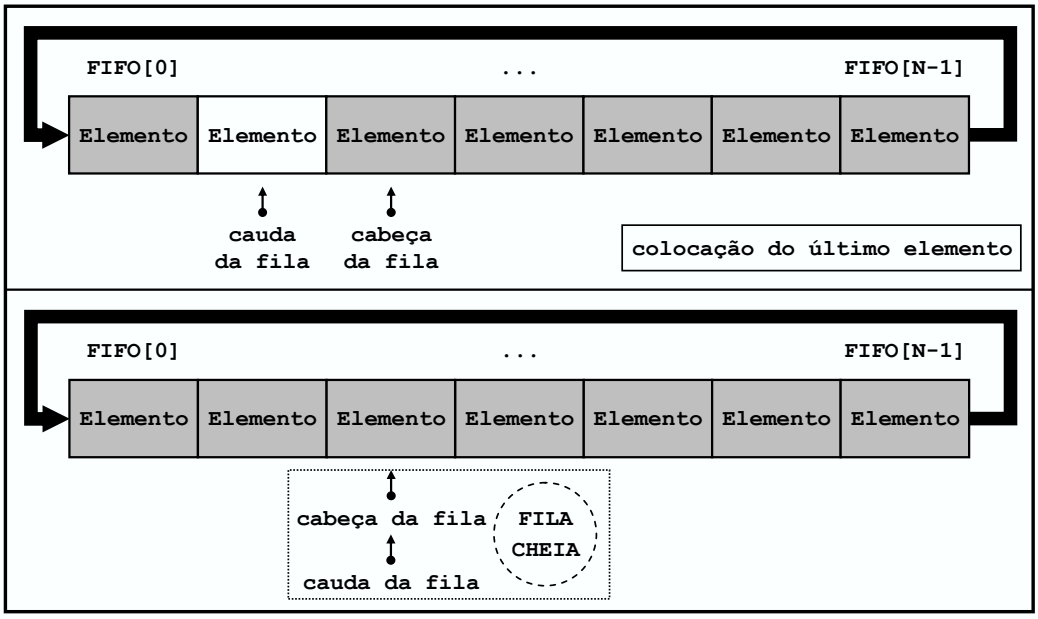

Figura 7.6 - Colocação do último elemento na fila.

Sempre que um elemento é retirado da fila, o indicador de cabeça da fila é deslocado para o elemento seguinte. Como já foi referido, esta implementação não é a forma habitual de funcionamento de um fila, onde sempre que o elemento da cabeça da fila sai da fila, toda a fila é deslocada para a frente. Existem duas situações distintas quando se retira um elemento da fila.

Na primeira situação, que se apresenta na Figura 7.7, a posição seguinte da fila está ainda dentro da dimensão do agregado, pelo que, a cabeça ainda não deu a volta ao agregado. Neste caso ainda está atrás da cauda, porque a cauda já excedeu o fim do agregado.

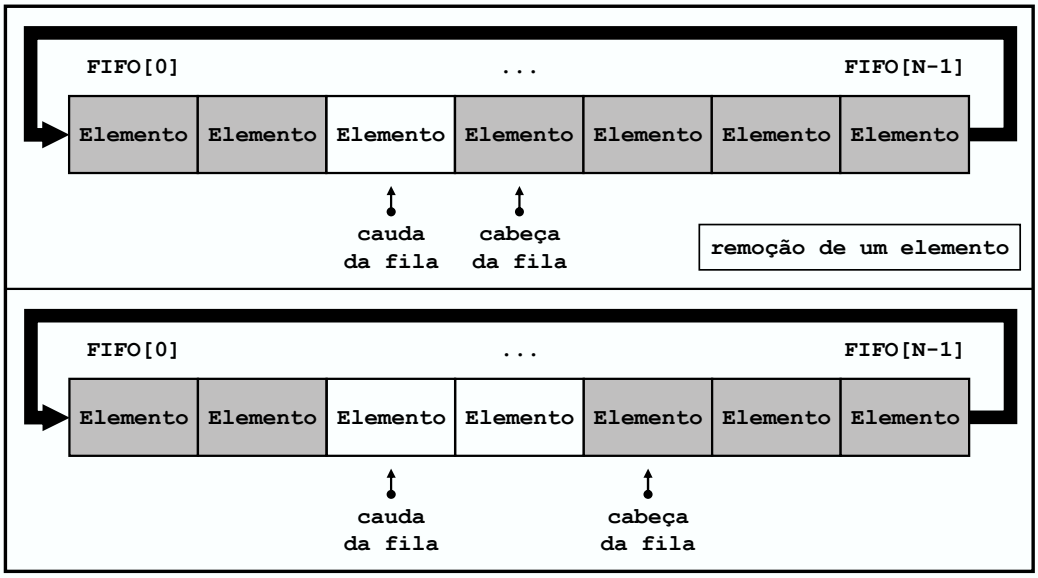

Figura 7.7 - Remoção de um elemento da fila.

Na segunda situação, que se apresenta na Figura 7.8, a posição seguinte da fila está fora da dimensão do agregado, pelo que, a cabeça passa para o início do agregado e neste caso fica de novo à frente da cauda. Para conseguir este efeito circular da cabeça da fila utiliza-se o operador módulo, para calcular sempre um valor que está entre 0 e N_ELEMENTOS−1.

Figura 7.8 - Remoção de um elemento da fila com movimentação circular.

A Figura 7.9 apresenta a situação de remoção do último elemento da fila. Nesta situação a cabeça da fila fica a apontar para o mesmo elemento que a cauda da fila, o que significa que a fila ficou vazia. Enquanto este estado durar, não é possível retirar mais elementos da fila. Nesta situação, a cabeça da fila deve ficar a apontar para a posição N do agregado, ou seja, a posição depois do fim do agregado, como indicação de fila vazia. Assim, fica reposta o estado inicial da fila, com a diferença que neste caso, a cauda da fila não está a apontar para a posição 0 do agregado. Quando se colocar um elemento na fila, na posição apontada pela cauda da fila, a cabeça da fila será colocada de novo a apontar para o elemento acabado de colocar.

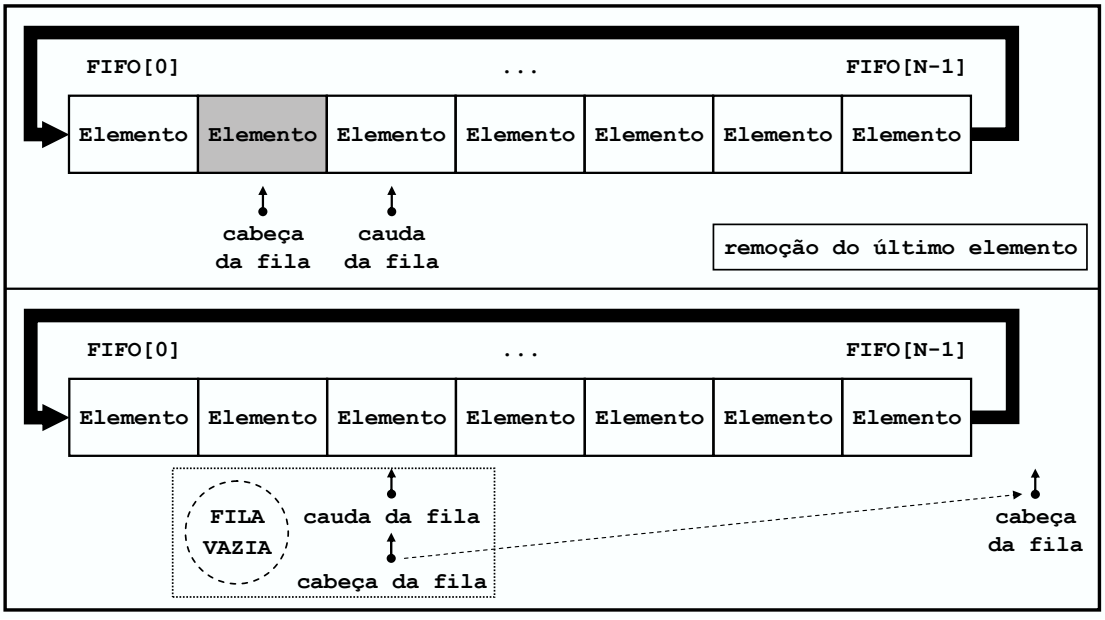

Figura 7.9 - Remoção do último elemento da fila.

A Figura 7.10 apresenta o ficheiro de implementação da fila estática. A estrutura de dados do módulo é constituída pelo agregado **FILA**, que é um agregado de elementos do tipo TElem definido no ficheiro de interface elemento.h, e pelas variáveis inteiras positivas **head** para a cabeça da fila e **tail** para a cauda da fila. A estrutura de dados é declarada com o qualificativo **static**, de modo a estar protegida do exterior.

```
/***************** Implementação da FILA Estática *****************/
                        /* Nome : fila_est.c */

#include <stdio.h>

#include "fila_est.h"           /* Ficheiro de interface do módulo */

        /* Definição da Estrutura de Dados Interna da FILA */

static TElem FILA[N_ELEMENTOS];           /* área de armazenamento */
static unsigned int head = N_ELEMENTOS;          /* cabeça da fila */
static unsigned int tail = 0;                     /* cauda da fila */

                      /* Definição das Funções */

int Fifo_In (TElem elemento)
{
   if (head == tail) return FIFO_FULL;
   FILA[tail] = elemento;
   if (head == N_ELEMENTOS) head = tail;
   tail = ++tail % N_ELEMENTOS;
   return OK;
}

int Fifo_Out (TElem *pelemento)
{
   if (pelemento == NULL) return NULL_PTR;
   if (head == N_ELEMENTOS) return FIFO_EMPTY;
   *pelemento = FILA[head];
   head = ++head % N_ELEMENTOS;
   if (head == tail) head = N_ELEMENTOS;
   return OK;
}
```

Figura 7.10 - Ficheiro de implementação da fila estática.

7.2.2 Implementação semiestática

A Figura 7.11 e a Figura 7.12 apresentam respectivamente o ficheiro de interface e o ficheiro de implementação da fila semiestática. A implementação semiestática da fila é baseada no agregado **FILA**, que é um agregado de ponteiros para elementos do tipo de dados TElem, que está definido no ficheiro de interface elemento.h, e pelas variáveis inteiras positivas **head** para a cabeça da fila e **tail** para a cauda da fila. Tal como na implementação estática, o agregado é usado de forma circular.

A implementação é em tudo semelhante à da implementação estática. A única diferença é que quando se coloca um elemento na fila é necessário atribuir memória para o armazenamento do elemento. A atribuição da memória é feita recorrendo à função **malloc**, uma vez que, a atribuição é feita elemento a elemento. Se não existir memória para essa atribuição, o elemento não pode ser colocado na fila, pelo que, é assinalada a situação de inexistência de memória, usando o código de erro NO_MEM. Por outro lado, quando se retira um elemento da fila é necessário libertar a memória ocupada com o armazenamento do elemento, recorrendo à função **free** e colocar o elemento do agregado a apontar para NULL.

```
/***************** Interface da FILA Semiestática *****************/
                    /* Nome : fila_semiest.h */

#ifndef _FILA_SEMIESTATICA
#define _FILA_SEMIESTATICA

#include "elemento.h"   /* caracterização do tipo elemento da fila */

                   /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NO_MEM      3   /* memória esgotada */
#define FIFO_EMPTY  4   /* fila vazia */
#define FIFO_FULL   5   /* fila cheia */

           /* Alusão às Funções Exportadas pelo Módulo */
int Fifo_In (TElem elemento);
/* Coloca o elemento elemento na cauda da fila. Valores de retorno:
OK, FIFO_FULL ou NO_MEM. */

int Fifo_Out (TElem *pelemento);
/* Retira o elemento da cabeça da fila para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou FIFO_EMPTY. */

#endif
```

Figura 7.11 - Ficheiro de interface da fila semiestática.

```
/*************** Implementação da FILA Semiestática ***************/
                    /* Nome : fila_semiest.c */

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

#include "fila_semiest.h"       /* Ficheiro de interface do módulo */

        /* Definição da Estrutura de Dados Interna da FILA */

static TElem *FILA[N_ELEMENTOS];          /* área de armazenamento */
static unsigned int head = N_ELEMENTOS;          /* cabeça da fila */
static unsigned int tail = 0;                     /* cauda da fila */

                    /* Definição das Funções */

int Fifo_In (TElem elemento)
{
  if (head == tail) return FIFO_FULL;
  if ((FILA[tail] = (TElem *) malloc (sizeof (TElem))) == NULL)
    return NO_MEM;
  *FILA[tail] = elemento;
  if (head == N_ELEMENTOS) head = tail;
  tail = ++tail % N_ELEMENTOS;
  return OK;
}

int Fifo_Out (TElem *pelemento)
{
  if (pelemento == NULL) return NULL_PTR;
  if (head == N_ELEMENTOS) return FIFO_EMPTY;
  *pelemento = *FILA[head];
  free (FILA[head]);
  FILA[head] = NULL;
  head = ++head % N_ELEMENTOS;
  if (head == tail) head = N_ELEMENTOS;
  return OK;
}
```

Figura 7.12 - Ficheiro de implementação da fila semiestática.

7.2.3 Implementação dinâmica

A Figura 7.13 apresenta o ficheiro de interface da implementação dinâmica de uma fila, que é baseada numa lista ligada de elementos. Uma lista ligada é uma estrutura constituída por elementos, a que vamos chamar nós, ligados através de ponteiros. Cada nó da lista ligada é constituído por dois ponteiros, um para o elemento que armazena a informação e outro para o nó seguinte da lista. O último nó da lista aponta para NULL, para servir de indicador de finalização da fila. A memória para os nós e para os elementos é atribuída, quando um elemento é colocado na fila e é libertada quando um elemento é retirado da fila. Os indicadores de cabeça e cauda da fila são ponteiros. A cabeça da fila aponta para o elemento mais antigo que se encontra na fila e que é o primeiro a ser retirado. A cauda da fila aponta sempre para o elemento mais recente que se encontra na fila e à frente do qual se insere um novo elemento. Quando são ambos ponteiros nulos, é sinal que a fila está vazia. Uma fila dinâmica nunca está cheia. Quando muito, pode não existir memória para continuar a acrescentar-lhe mais elementos. Portanto, as situações de erro são as mesmas da implementação semiestática, com excepção que não existe o código de erro FIFO_FULL.

```
/******************* Interface da FILA Dinâmica *******************/
                        /* Nome : fila_din.h */
#ifndef _FILA_DINAMICA
#define _FILA_DINAMICA

#include "elemento.h"   /* caracterização do tipo elemento da fila */

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NO_MEM      3   /* memória esgotada */
#define FIFO_EMPTY  4   /* fila vazia */

            /* Alusão às Funções Exportadas pelo Módulo */

int Fifo_In (TElem elemento);
/* Coloca o elemento apontado por elemento na cauda da fila. Valores
de retorno: OK ou NO_MEM. */

int Fifo_Out (TElem *pelemento);
/* Retira o elemento da cabeça da fila para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou FIFO_EMPTY. */

#endif
```

Figura 7.13 - Ficheiro de interface da fila dinâmica.

Vamos apresentar de forma gráfica as operações de colocação e remoção de elementos numa fila dinâmica. A Figura 7.14 apresenta o estado inicial da fila e o estado após a colocação do primeiro elemento. Para colocar um elemento na fila primeiro é necessário atribuir memória para o armazenamento do nó da lista e depois para o armazenamento do elemento. Se não existir memória para essas atribuições, então o elemento não pode ser colocado na fila e é assinalada esta situação de erro. Após a atribuição da memória, o elemento é ligado ao nó e o nó é ligado à fila. O nó fica a apontar para NULL, uma vez que é o último nó da fila, ou seja, ele é o elemento finalizador da fila. A cauda da fila é posta a apontar para ele e finalmente, a informação a armazenar é copiada para o elemento.

No caso da colocação do primeiro elemento na fila, ele não precisa de ser ligado à fila, mas, em contrapartida a cabeça da fila que estava a apontar para NULL é colocada a apontar para o nó deste elemento. Ele é simultaneamente o primeiro e último elemento da fila.

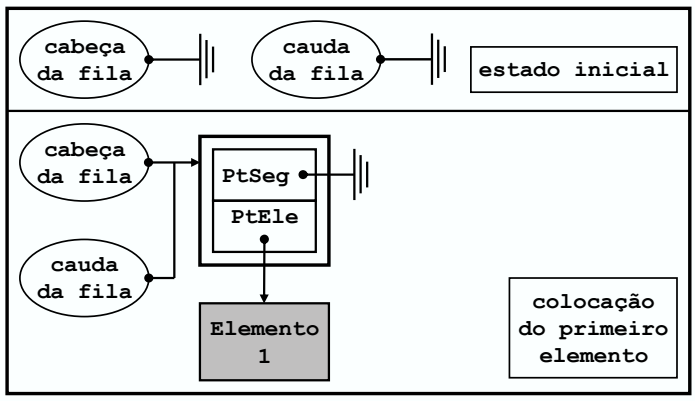

Figura 7.14 - Situação inicial da fila e após a colocação do primeiro elemento.

A Figura 7.15 apresenta a colocação de mais um elemento na fila. Neste caso, este elemento tem de ser ligado à fila, pelo que, o nó do elemento que está apontado pela cauda e que aponta para NULL, é posto a apontar para o novo nó, que passa agora a ser o último elemento da fila. A cauda da fila é actualizada e a cabeça da fila continua inalterada.

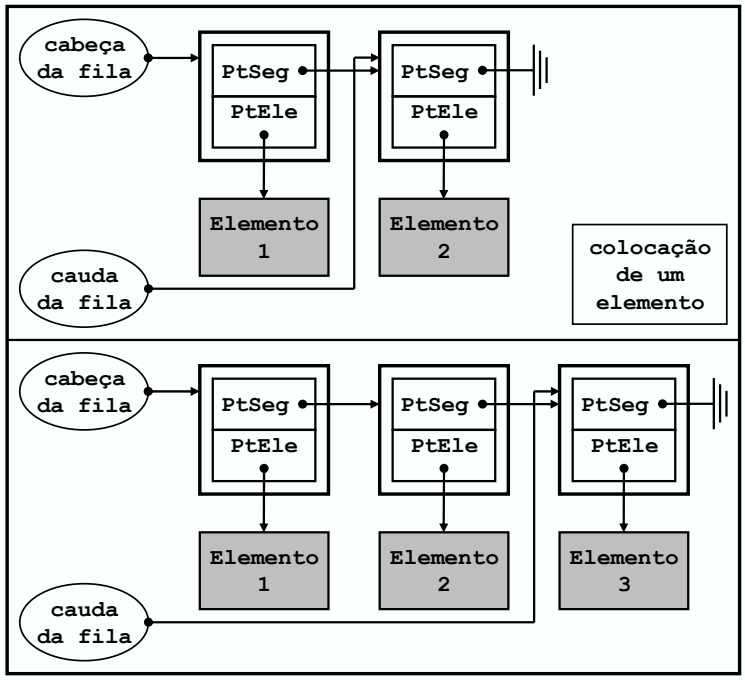

Figura 7.15 - Colocação de mais um elemento na fila.

A Figura 7.16 apresenta a remoção de um elemento da fila. A informação armazenada no elemento é copiada e depois a cabeça da fila é colocada a apontar para o elemento seguinte que vai passar agora a ser a nova cabeça da fila. Esta operação é feita atribuindo à cabeça da fila o valor apontado pelo ponteiro PtSeg, que aponta para o nó seguinte da fila. Toda a memória ocupada pelo elemento e pelo nó é libertada.

A Figura 7.17 mostra a remoção do último elemento da fila. Como este último nó aponta para NULL, ao ser atribuído o valor NULL à cabeça da fila isso é sinal de que a fila ficou vazia, pelo que, a cauda da fila também é colocada a apontar para NULL. Após esta operação, a fila fica num estado igual ao estado inicial.

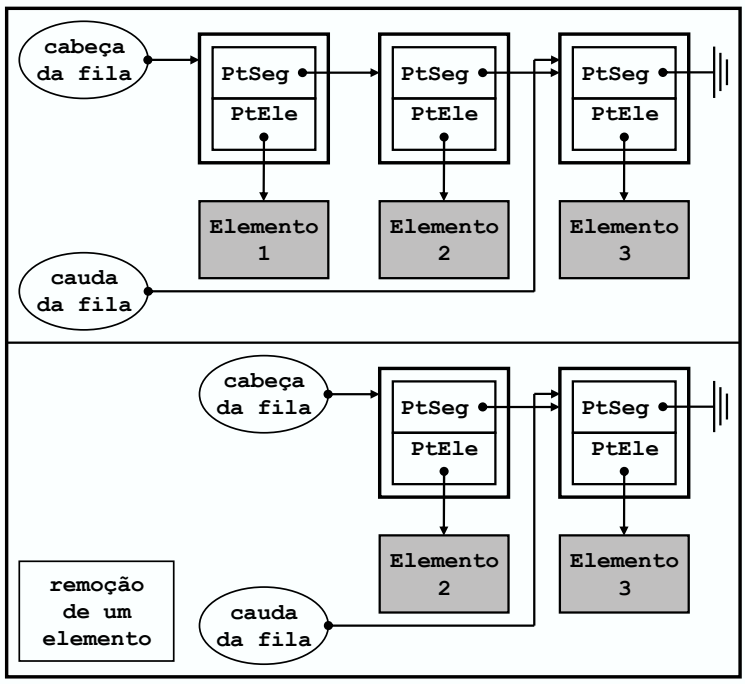

Figura 7.16 - Remoção de um elemento da fila.

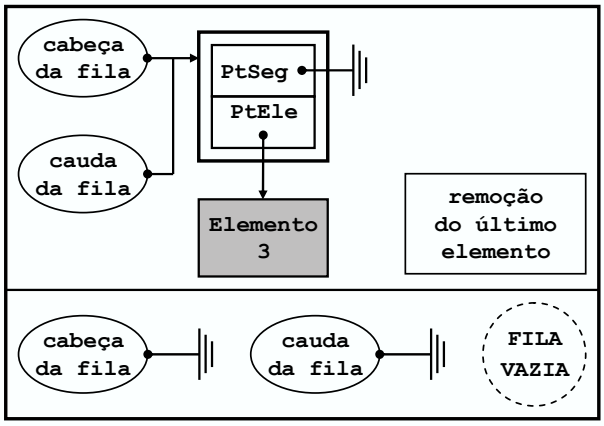

Figura 7.17 - Remoção do último elemento da fila.

A Figura 7.18 apresenta o ficheiro de implementação da fila dinâmica. A estrutura de dados do módulo é constituída por uma lista ligada de nós do tipo **struct** no, sendo cada nó constituído pelo ponteiro **pelemento** para um elemento do tipo TElem, que está definido no ficheiro de interface elemento.h, e pelo ponteiro **pseg** para fazer a ligação do nó ao nó seguinte da fila, caso ele exista. Para controlar a fila existem os ponteiros, **head** e **tail** respectivamente para a cabeça e para a cauda da fila, que são declarados com o qualificativo **static**, de modo a estarem protegidas do exterior. Como a fila inicialmente está vazia, estes ponteiros são inicializadas a NULL. Quando um elemento é colocado na fila, a atribuição de memória para o seu crescimento, começa pela atribuição de memória para o nó, que caso seja bem sucedida, prossegue com a atribuição de memória para o elemento. Caso não exista memória para o elemento, a colocação de mais um elemento na fila é cancelada e a memória anteriormente atribuída para o nó é libertada, antes da função terminar a sua execução com o código de erro NO_MEM. Não faz sentido ter um nó na fila, senão podemos colocar também o elemento que vai armazenar a informação. O nó do novo elemento aponta para NULL, uma vez que é o último elemento da fila. A função prossegue com a ligação do último elemento que estava na fila ao novo elemento e com a actualização do ponteiro cauda da fila que fica a apontar para o novo elemento. Finalmente é feita a

cópia da informação para o elemento. Quando um elemento é retirado da fila, primeiro é verificado se o ponteiro passado não é nulo e só depois é que a informação armazenada no elemento é retirada da fila. Caso a fila não esteja vazia, o ponteiro cabeça da fila é colocado a apontar para o elemento seguinte. Quando o último elemento é retirado da fila, então a cabeça da fila fica a apontar para NULL, o que significa que a fila ficou vazia, pelo que, a cauda da fila também é colocada a apontar para NULL. Depois a memória ocupada pelo elemento é libertada e só depois é que memória ocupada pelo nó é libertada.

```
/***************** Implementação da FILA Dinâmica *****************/
                        /* Nome : fila_din.c */
#include <stdio.h>
#include <stdlib.h>
#include "fila_din.h"           /* Ficheiro de interface do módulo */

        /* Definição da Estrutura de Dados Interna da FILA */

typedef struct no *PtNo;

struct no
{
   TElem *pelemento;                    /* ponteiro para o elemento */
   PtNo pseg;                         /* ponteiro para o nó seguinte */
};

static PtNo head = NULL;                            /* cabeça da fila */
static PtNo tail = NULL;                             /* cauda da fila */

                      /* Definição das Funções */

int Fifo_In (TElem elemento)
{
   PtNo tmp;

   if ((tmp = (PtNo) malloc (sizeof (struct no))) == NULL)
      return NO_MEM;

   if ((tmp->pelemento = (TElem *) malloc (sizeof(TElem))) == NULL)
   {
      free (tmp); return NO_MEM;
   }

   tmp->pseg = NULL;
   if (tail == NULL) head = tmp;
   else tail->pseg = tmp;
   tail = tmp;
   *tail->pelemento = elemento;
   return OK;
}

int Fifo_Out (TElem *pelemento)
{
   PtNo tmp;

   if (pelemento == NULL) return NULL_PTR;
   if (head == NULL) return FIFO_EMPTY;

   *pelemento = *head->pelemento;
   tmp = head;
   head = head->pseg;
   if (head == NULL) tail = NULL;
   free (tmp->pelemento);
   free (tmp);
   return OK;
}
```

Figura 7.18 - Ficheiro de implementação da fila dinâmica.

7.3 Pilhas

Uma memória pilha (***stack*/*LIFO***) é uma memória em que só é possível processar a informação pela ordem inversa à ordem de chegada. Daí que, também seja apelidada de memória do último a chegar primeiro a sair. Numa memória pilha, o posicionamento para a colocação de um elemento na pilha, que vamos designar por **Stack_Push**, e o posicionamento para a remoção de um elemento da pilha, que vamos designar por **Stack_Pop**, é o topo da pilha (***top of the stack***).

7.3.1 Implementação estática

A Figura 7.19 apresenta o ficheiro de interface da implementação estática de uma pilha, que é baseada num agregado de elementos. O indicador de topo da pilha é uma variável de tipo inteiro positivo.

Como um agregado tem uma dimensão fixa, antes de se colocar um elemento na pilha é necessário verificar se a pilha está cheia. Em caso afirmativo, mais nenhum elemento pode ser colocado na pilha e é assinalada a situação de erro, usando o código de erro STACK_FULL. O elemento que se pretende copiar para a pilha é passado à função **Stack_Push** por valor. Por outro lado, o elemento que vai receber a cópia do elemento que se pretende retirar da pilha é passado à função **Stack_Pop** por referência. Pode acontecer que o ponteiro passado à função seja um ponteiro nulo. Nestas circunstâncias a função não pode retirar o elemento da pilha, pelo que, não faz nada e assinala esta anomalia, usando o código de erro NULL_PTR. Antes de se retirar um elemento da pilha é preciso detectar se a pilha está vazia. Em caso afirmativo, nenhum elemento pode ser retirado da pilha e é assinalada a situação de erro, usando o código de erro STACK_EMPTY. Sempre que é colocado ou retirado um elemento da pilha é devolvido o código OK sinalizando que a operação foi realizada com sucesso.

```
 /****************** Interface do PILHA Estática ******************/
                       /* Nome : pilha_est.h */

#ifndef _PILHA_ESTATICA
#define _PILHA_ESTATICA

#include "elemento.h"  /* caracterização do tipo elemento da pilha */

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define STACK_EMPTY 4   /* pilha vazia */
#define STACK_FULL  5   /* pilha cheia */

            /* Alusão às Funções Exportadas pelo Módulo */

int Stack_Push (TElem elemento);
/* Coloca o elemento apontado por elemento no topo da pilha. Valores
de retorno: OK ou STACK_FULL. */

int Stack_Pop (TElem *pelemento);
/* Retira o elemento do topo da pilha para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou STACK_EMPTY. */

#endif
```

Figura 7.19 - Ficheiro de interface da pilha estática.

De seguida, vamos apresentar graficamente o comportamento das operações de colocação de um elemento na pilha e de remoção de um elemento da pilha.

A Figura 7.20 mostra o estado inicial da pilha. Por uma questão de implementação, vamos considerar que o topo da pilha indica sempre a primeira posição livre para a próxima operação de colocação de um elemento na pilha, pelo que, inicialmente aponta para a posição 0 do agregado. A Figura 7.20 mostra também a colocação do primeiro elemento na pilha e o estado da pilha após a operação. O topo da pilha é deslocado para a posição 1 do agregado, que é a primeira posição livre para colocar o próximo elemento.

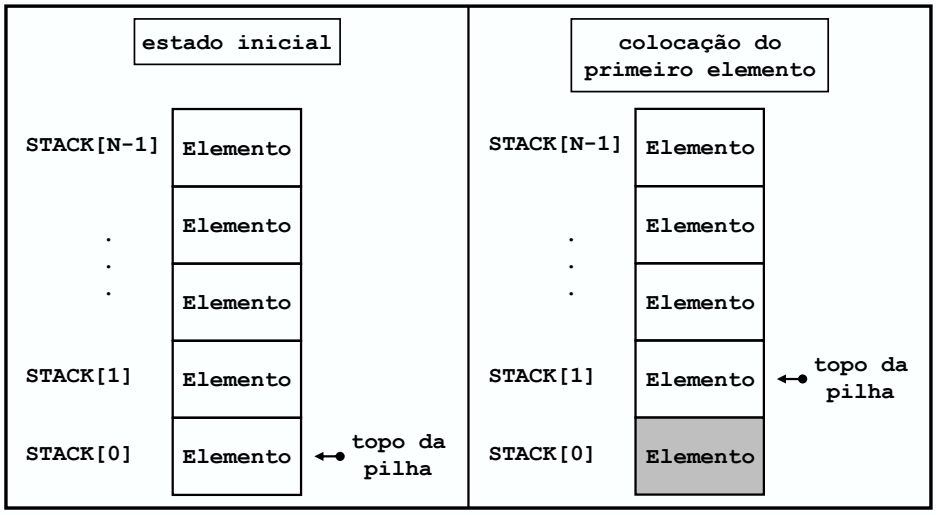

Figura 7.20 - Situação inicial da pilha e após a colocação do primeiro elemento.

Sempre que se coloca um elemento na pilha, o topo da pilha é deslocada para a posição seguinte da pilha, depois da cópia do elemento para a pilha. A Figura 7.21 apresenta a situação limite de utilização da pilha, quando se coloca o último elemento na pilha. Nesta situação o topo da pilha fica a apontar para a posição do agregado de índice N, o que significa que a pilha ficou cheia. Enquanto este estado durar, não é possível colocar mais elementos na pilha.

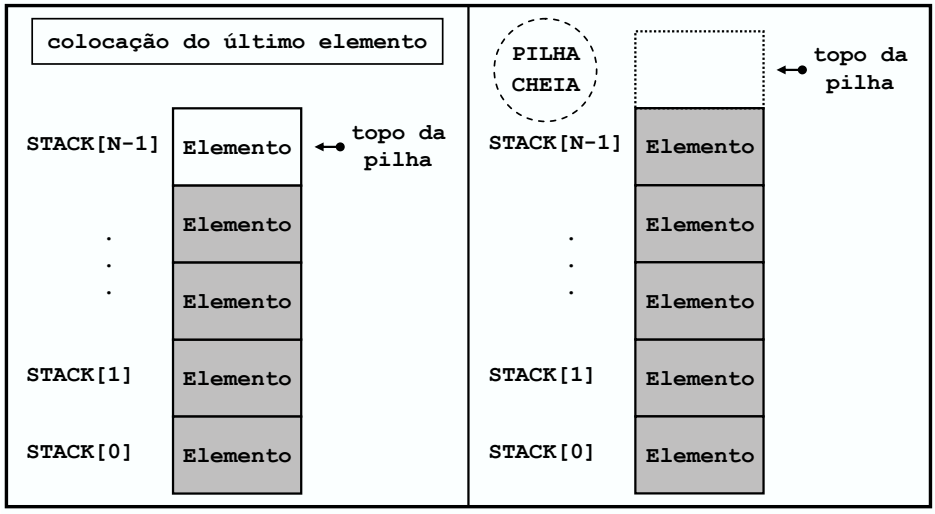

Figura 7.21 - Colocação do último elemento na pilha.

Sempre que um elemento é retirado da pilha, primeiro é preciso deslocar o topo da pilha para a posição anterior da pilha, onde está armazenado o último elemento da pilha. Só depois é que o elemento é copiado para fora da pilha.

A Figura 7.22 apresenta a situação de remoção de um elemento, quando existe mais do que um elemento na pilha. O topo da pilha é deslocado para a posição anterior e a informação armazenada no elemento é retirado da pilha.

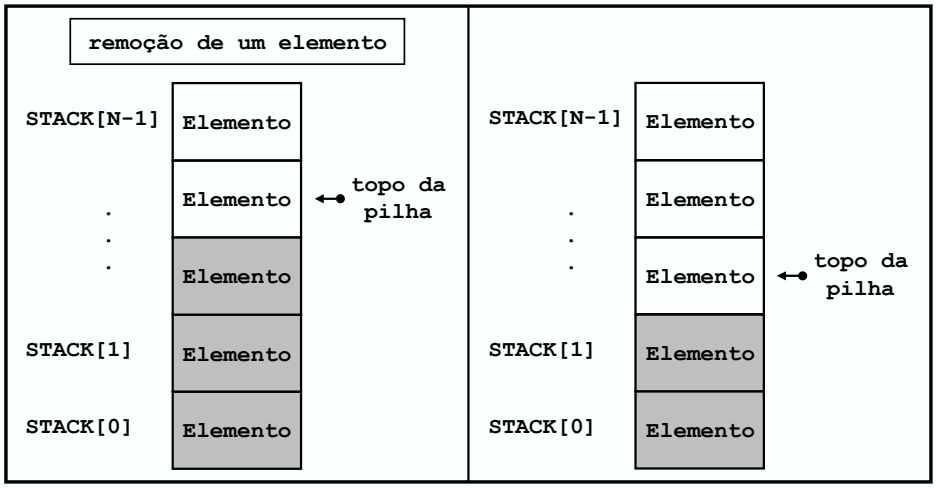

Figura 7.22 - Remoção de um elemento da pilha.

A Figura 7.23 apresenta a situação de remoção do último elemento da pilha. Após a operação, o topo da pilha fica a apontar para a posição do agregado de índice 0, o que significa que ficou reposta o estado inicial da pilha, ou seja, a pilha ficou vazia. Enquanto este estado durar, não é possível retirar mais elementos da pilha.

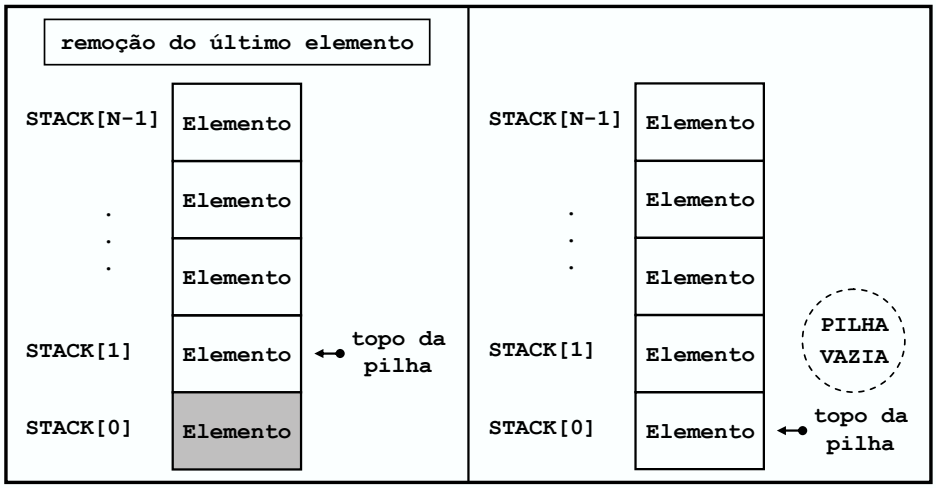

Figura 7.23 - Remoção do último elemento da fila.

A Figura 7.24 apresenta o ficheiro de implementação da pilha estática. A estrutura de dados do módulo é constituída pelo agregado **PILHA**, que é um agregado de elementos do tipo TElem, que está definido no ficheiro de interface elemento.h, e pela variável inteira positiva **top** para o topo da pilha. A estrutura de dados é declarada com o qualificativo **static**, de modo a estar protegida do exterior.

```
 /**************** Implementação da PILHA Estática ***************/
                      /* Nome : pilha_est.c */
#include <stdio.h>
#include "pilha_est.h"          /* Ficheiro de interface do módulo */
       /* Definição da Estrutura de Dados Interna da PILHA */
static TElem PILHA[N_ELEMENTOS];          /* área de armazenamento */
static unsigned int top = 0;                      /* topo da pilha */
                     /* Definição das Funções */
int Stack_Push (TElem elemento)
{
   if (top == N_ELEMENTOS) return STACK_FULL;
   PILHA[top++] = elemento;
   return OK;
}
int Stack_Pop (TElem *pelemento)
{
   if (pelemento == NULL) return NULL_PTR;
   if (top == 0) return STACK_EMPTY;
   *pelemento = PILHA[--top];
   return OK;
}
```

Figura 7.24 - Ficheiro de implementação da pilha estática.

7.3.2 Implementação semiestática

A Figura 7.25 e a Figura 7.26 apresentam respectivamente o ficheiro de interface e o ficheiro de implementação da pilha semiestática. A implementação semiestática da pilha é baseada no agregado **PILHA**, que é um agregado de ponteiros para elementos do tipo de dados TElem, que está definido no ficheiro de interface elemento.h, e pela variável inteira positiva **top** para o topo da pilha. O agregado é usado tal como na implementação estática.

A implementação é em tudo semelhante à da implementação estática. A única diferença é que quando se coloca um elemento na pilha é necessário atribuir memória para o armazenamento do elemento. A atribuição da memória é feita recorrendo à função **malloc**, uma vez que, a atribuição é feita elemento a elemento. Se não existir memória para essa atribuição, o elemento não pode ser colocado na pilha e é assinalada a situação de inexistência de memória, usando o código de erro NO_MEM. Por outro lado, quando se retira um elemento da pilha é necessário libertar a memória ocupada com o armazenamento do elemento, recorrendo à função **free** e colocar o elemento do agregado a apontar para NULL.

```
 /**************** Interface do PILHA Semiestática ***************/
                    /* Nome : pilha_semiest.h */

#ifndef _PILHA_SEMIESTATICA
#define _PILHA_SEMIESTATICA

#include "elemento.h"  /* caracterização do tipo elemento da pilha */

                   /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NO_MEM      3   /* memória esgotada */
#define STACK_EMPTY 4   /* pilha vazia */
#define STACK_FULL  5   /* pilha cheia */

           /* Alusão às Funções Exportadas pelo Módulo */

int Stack_Push (TElem elemento);
/* Coloca o elemento apontado por elemento no topo da pilha. Valores
de retorno: OK, STACK_FULL ou NO_MEM. */

int Stack_Pop (TElem *pelemento);
/* Retira o elemento do topo da pilha para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou STACK_EMPTY. */

#endif
```

Figura 7.25 - Ficheiro de interface da pilha semiestática.

```
 /************** Implementação da PILHA Semiestática *************/
                    /* Nome : pilha_semiest.c */

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

#include "pilha_semiest.h"      /* Ficheiro de interface do módulo */

       /* Definição da Estrutura de Dados Interna da PILHA */

static TElem *PILHA[N_ELEMENTOS];         /* área de armazenamento */
static unsigned int top = 0;                      /* topo da pilha */

                    /* Definição das Funções */

int Stack_Push (TElem elemento)
{
  if (top == N_ELEMENTOS) return STACK_FULL;

  if ((PILHA [top] = (TElem *) malloc (sizeof (TElem))) == NULL)
    return NO_MEM;

  *PILHA[top++] = elemento;
  return OK;
}

int Stack_Pop (TElem *pelemento)
{
  if (pelemento == NULL) return NULL_PTR;

  if (top == 0) return STACK_EMPTY;

  *pelemento = *PILHA[--top];
  free (PILHA[top]);
  PILHA[top] = NULL;
  return OK;
}
```

Figura 7.26 - Ficheiro de implementação da pilha semiestática.

7.3.3 Implementação dinâmica

A Figura 7.27 apresenta o ficheiro de interface da implementação dinâmica, que é baseada numa lista ligada de elementos. Uma lista ligada é uma estrutura constituída por elementos, a que vamos chamar nós, ligados através de ponteiros. Cada nó da lista ligada é constituído por dois ponteiros, um para o elemento que armazena a informação e outro para o nó anterior da lista. O primeiro nó da lista aponta para NULL, para servir de indicador de finalização da pilha. A memória para os nós e para os elementos da pilha é atribuída, quando um elemento é colocado na pilha e é libertada quando um elemento é retirado da pilha. O indicador de topo da pilha é um ponteiro, que aponta para o elemento mais recentemente colocado na pilha e que é o primeiro a ser retirado. Quando o topo da pilha é um ponteiro nulo, é sinal que a pilha está vazia. Uma pilha dinâmica nunca está cheia. Quando muito, pode não existir memória para continuar a acrescentar-lhe mais elementos. Portanto, as situações de erro são as mesmas da implementação semiestática, com excepção que não existe o código de erro STACK_FULL.

```
 /****************** Interface da PILHA Dinâmica ******************/
                        /* Nome : pilha_din.h */
#ifndef _PILHA_DINAMICA
#define _PILHA_DINAMICA

#include "elemento.h"  /* caracterização do tipo elemento da pilha */

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NO_MEM      3   /* memória esgotada */
#define STACK_EMPTY 4   /* pilha vazia */

            /* Alusão às Funções Exportadas pelo Módulo */

int Stack_Push (TElem elemento);
/* Coloca o elemento apontado por elemento no topo da pilha. Valores
de retorno: OK ou NO_MEM. */

int Stack_Pop (TElem *pelemento);
/* Retira o elemento do topo da pilha para o elemento apontado por
pelemento. Valores de retorno: OK, NULL_PTR ou STACK_EMPTY. */

#endif
```

Figura 7.27 - Ficheiro de interface da pilha dinâmica.

Vamos apresentar de forma gráfica as operações de colocação e remoção de elementos numa pilha dinâmica. A Figura 7.28 apresenta o estado inicial da pilha e o estado após a colocação do primeiro elemento.

Para colocar um elemento na pilha é necessário atribuir memória, primeiro para o armazenamento do nó da lista e depois para o armazenamento do elemento. Se não existir memória para essas atribuições, então o elemento não pode ser colocado na pilha e é assinalada esta situação de erro. Após a atribuição da memória, o elemento é ligado ao nó e o nó é ligado à pilha. O nó é fica a apontar para o nó anterior que é o topo da pilha, que depois é actualizado, sendo colocado a apontar para este novo nó. Finalmente, a informação a armazenar é copiada para o elemento.

No caso da colocação do primeiro elemento na pilha, ele não precisa de ser ligado à pilha, mas, em contrapartida é colocado a apontar para NULL, indicando que ele é o primeiro elemento da pilha. Ele é simultaneamente o primeiro e o último elemento da pilha.

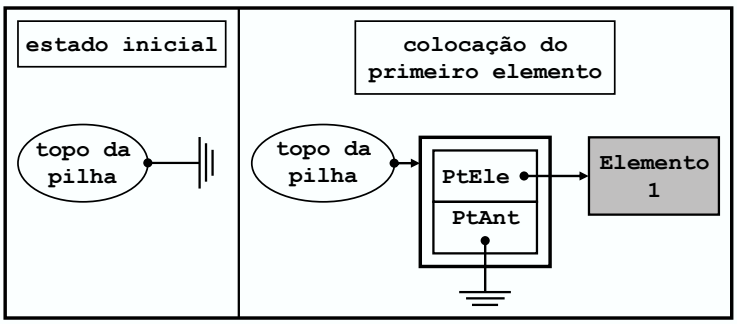

Figura 7.28 - Situação inicial da pilha e após a colocação do primeiro elemento na pilha.

A Figura 7.29 apresenta a colocação de mais um elemento na pilha. O nó do novo elemento é posto a apontar para o antigo topo da pilha, através do ponteiro PtAnt, e depois passa a ser o novo topo da pilha. Ou seja, o topo da pilha é actualizado, ficando a apontar para este novo nó.

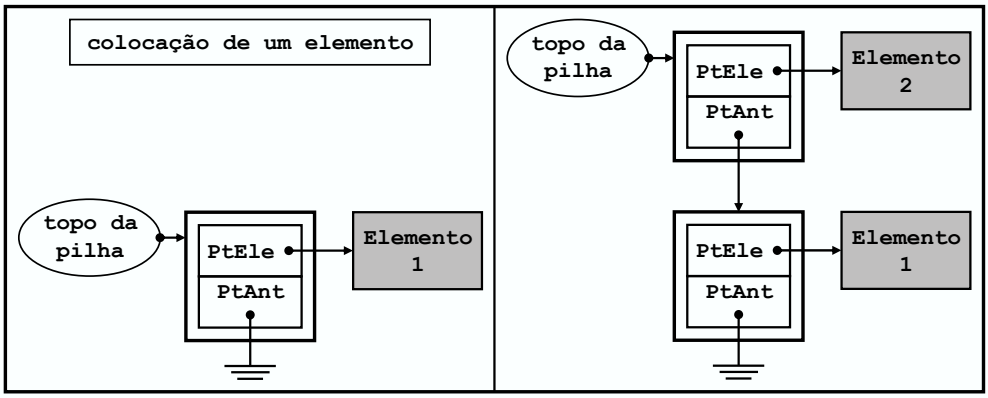

Figura 7.29 - Colocação de mais um elemento na pilha.

A Figura 7.30 apresenta a remoção de um elemento da pilha. A informação armazenada no elemento é copiada. Depois o topo da pilha é colocada a apontar para o elemento anterior que vai passar agora a ser o novo topo da pilha. Esta operação é feita atribuindo ao topo da pilha o valor apontado pelo ponteiro PtAnt, que aponta para o nó anterior da pilha. Depois toda a memória ocupada pelo elemento e pelo nó é libertada.

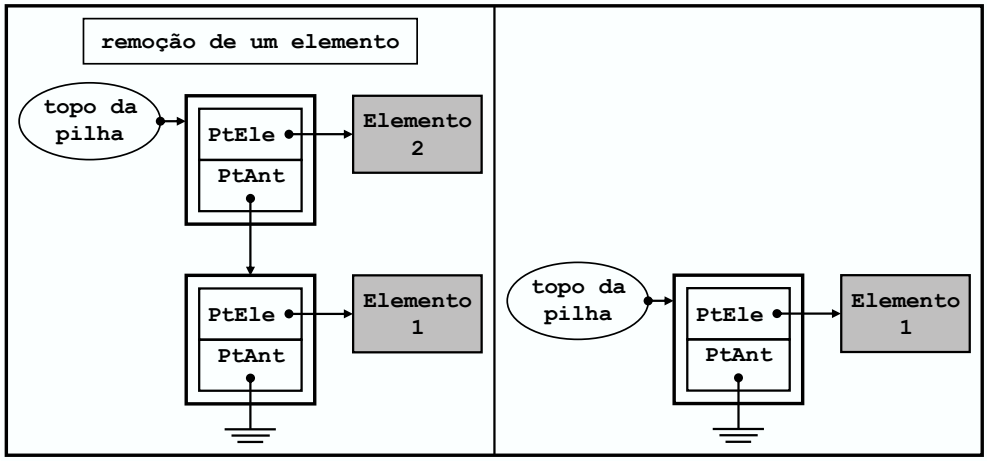

Figura 7.30 - Remoção de um elemento da pilha.

A Figura 7.31 mostra a remoção do último elemento da pilha. Como este último nó aponta para NULL, ao ser atribuído o valor NULL ao topo da pilha isso é sinal de que a pilha ficou vazia, e a pilha fica num estado igual ao estado inicial.

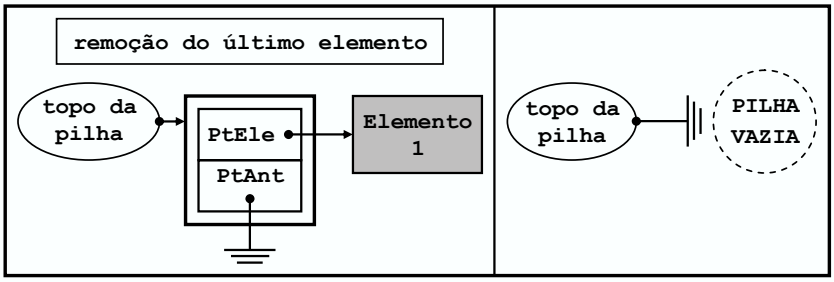

Figura 7.31 - Remoção do último elemento da pilha.

A Figura 7.32 apresenta o ficheiro de implementação da pilha dinâmica. A estrutura de dados do módulo é constituída por uma lista ligada de nós do tipo **struct** no, sendo cada nó constituído pelo ponteiro **pelemento** para um elemento do tipo TElem, que está definido no ficheiro de interface elemento.h, e pelo ponteiro **pant** para fazer a ligação do nó ao nó anterior da pilha, caso ele exista. Para controlar a pilha existe o ponteiro **top** para o topo da pilha, que é declarado com o qualificativo **static**, de modo a estar protegido do exterior. Como a pilha inicialmente está vazia, este ponteiro é inicializado a NULL.

Quando um elemento é colocado na pilha, a atribuição de memória para o seu crescimento, começa pela atribuição de memória para o nó, que caso seja bem sucedida, prossegue com a atribuição de memória para o elemento. Caso não exista memória para o elemento, a colocação de mais um elemento na pilha é cancelada e a memória anteriormente atribuída para o nó é libertada, antes da função terminar a sua execução com o código de erro NO_MEM. Não faz sentido ter um nó na pilha, senão podemos colocar também o elemento que vai armazenar a informação. A função continua com a ligação do novo elemento ao último elemento que estava na pilha, pelo que, o nó do primeiro elemento da pilha fica a apontar para NULL. Depois é feita a actualização do ponteiro topo da pilha que fica a apontar para o novo elemento e finalmente é feita a cópia da informação para o elemento.

Quando um elemento é retirado da pilha, primeiro é verificado se o ponteiro passado não é nulo e só depois é que a informação armazenada no elemento é retirada da pilha. O ponteiro topo da pilha é colocado a apontar para o elemento anterior. Quando o último elemento é retirado da pilha, então o topo da pilha fica a apontar para NULL, o que significa que a pilha ficou vazia. Depois a memória ocupada pelo elemento é libertada e só depois é que a memória ocupada pelo nó da pilha é libertada.

```
 /**************** Implementação da PILHA Dinâmica ***************/
                      /* Nome : pilha_din.c */
#include <stdio.h>
#include <stdlib.h>
#include "pilha_din.h"          /* Ficheiro de interface do módulo */
       /* Definição da Estrutura de Dados Interna da PILHA */
typedef struct no *PtNo;
struct no
{
   TElem *pelemento;                    /* ponteiro para o elemento */
   PtNo pant;                         /* ponteiro para o nó anterior */
};
static PtNo top = NULL;                              /* topo da pilha */
                     /* Definição das Funções */
int Stack_Push (TElem elemento)
{
   PtNo tmp;
   if ((tmp = (PtNo) malloc (sizeof (struct no))) == NULL)
      return NO_MEM;
   if ((tmp->pelemento = (TElem *) malloc (sizeof(TElem))) == NULL)
   {
      free (tmp); return NO_MEM;
   }
   tmp->pant = top;
   top = tmp;
   *top->pelemento = elemento;
   return OK;
}
int Stack_Pop (TElem *pelemento)
{
   PtNo tmp;
   if (pelemento == NULL) return NULL_PTR;
   if (top == NULL) return STACK_EMPTY;
   *pelemento = *top->pelemento;
   tmp = top;
   top = top->pant;
   free (tmp->pelemento);
   free (tmp);
   return OK;
}
```

Figura 7.32 - Ficheiro de implementação da pilha dinâmica.

7.4 Exemplos de aplicação de filas e pilhas

Uma das aplicações das pilhas é a análise e processamento de estruturas imbricadas. Estruturas imbricadas são estruturas que são compostas internamente por outras estruturas do mesmo tipo, onde existe a necessidade de garantir que uma estrutura interna é finalizada antes da estrutura externa. Um exemplo típico de uma estrutura imbricada é uma expressão aritmética que é normalmente composta por subexpressões aritméticas entre parênteses

curvos. Por vezes para melhor identificar os vários níveis de subexpressões, utilizam-se parênteses curvos, rectos e chavetas, tal se mostra na seguinte expressão.

$$\{ A + B * [C / (F+A) + (C+D) * (E-F)] \} / [(A+B) * C]$$

Para que uma expressão aritmética possa ser correctamente calculada é necessário assegurar o correcto balanceamento dos parênteses, como é o caso da expressão anterior. Uma expressão com um balanceamento correcto de parênteses, tem tantos parênteses a abrir, ou seja, parênteses esquerdos, como parênteses a fechar, ou seja, parênteses direitos. Mas também é preciso assegurar que um nível interno de parênteses é fechado antes que um nível externo e com um parêntese direito equivalente ao parêntese esquerdo. Pelo que, a análise não pode apenas limitar-se a contar o número de parênteses direitos e esquerdos de cada tipo. Por exemplo, a expressão seguinte tem o mesmo número de parênteses direitos e esquerdos e não está balanceada, porque a chaveta é fechada antes do parêntese recto.

$$\{ A + B * [C / (F+A) + (C+D) * (E-F) \}] / [(A+B) * C]$$

Uma vez que é necessário assegurar que cada parêntese direito é equivalente ao último parêntese esquerdo, esta análise pode ser feita usando uma pilha O algoritmo é o seguinte. Sempre que aparece um parêntese esquerdo na expressão, este é colocado na pilha. Quando aparece um parêntese direito, tira-se da pilha o último parêntese esquerdo lá colocado e verifica-se se são equivalentes. Caso o teste seja positivo então este nível interno de parênteses está balanceado e ambos os parênteses são descartados. Caso os parênteses não sejam equivalentes, então o processo é interrompido porque este nível interno de parênteses não está balanceado. Se por acaso não existir um parêntese esquerdo na pilha, então é sinal que existem mais parênteses direitos que esquerdos até esta posição da expressão, pelo que, a expressão também não está balanceada. Quando a análise da expressão terminar é preciso verificar se a pilha está vazia. Porque, caso não esteja vazia, então é sinal que existem mais parênteses esquerdos que direitos na expressão, pelo que, a expressão também não está balanceada.

A Figura 7.34 apresenta o programa que implementa este algoritmo. De maneira a estruturar melhor o programa, foi implementada uma função inteira que compara o parêntese esquerdo com o parêntese direito e que devolve 1, caso eles sejam equivalentes e 0 no caso contrário. Neste caso a pilha utilizada é a implementação estática, mas podia ser qualquer outra implementação. Para utilizar a pilha, primeiro é preciso editar o ficheiro de interface **elemento.h**, e definir o número de elementos da estrutura de dados da pilha com um valor suficiente para armazenar os parênteses da expressão, que no pior caso podem ser tantos quantos o número de caracteres da expressão e definir também o tipo dos elementos da pilha como sendo do tipo ***char***. Depois o módulo é compilado, com a opção **−c**, para ficar concretizado para uma pilha de caracteres. Finalmente, o programa é compilado, indicando no comando de compilação o ficheiro objecto do módulo **pilha_est.o**. A Figura 7.33 apresenta o resultado de execução do programa para várias expressões.

```
Expressão algébrica -> {A+B*[C/(F+A)+(C+D)*(E-F)]}/[(A+B)*C]
Expressão com parênteses balanceados

Expressão algébrica -> {A+B*[C/(F+A)+(C+D)*(E-F)}]/[(A+B)*C]
Parênteses [ e } discordantes

Expressão algébrica -> {A+B*[C/(F+A)+(C+D)*(E-F)]}/[(A+B)*C]}
Mais parênteses direitos do que esquerdos

Expressão algébrica -> {A+B*[C/(F+A)+(C+D)*(E-F)]}/[(A+B)*C
Mais parênteses esquerdos do que direitos
```

Figura 7.33 - Exemplos de utilização do programa.

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

#include "pilha_est.h" /* ficheiro de interface do módulo da pilha */

int parenteses_equivalentes (char, char);

int main (void)
{
  char exp[81],          /* expressão lida do teclado para analisar */
       cpilha;            /* parêntese esquerdo armazenado na pilha */
  int  nc,                       /* número de símbolos da expressão */
       c,                                 /* contador do ciclo for */
       st;             /* estado de realização da operação da pilha */

  printf ("Expressão algébrica -> ");
  scanf ("%80s", exp);

  nc = strlen (exp);

  for (c = 0; c < nc; c++)
    if ( exp[c] == '(' || exp[c] == '[' || exp[c] == '{' )
         Stack_Push (exp[c]);
    else if ( exp[c] == ')' || exp[c] == ']' || exp[c] == '}' )
         {
           st = Stack_Pop (&cpilha);
           if (st == STACK_EMPTY)
           {
             printf ("Mais parênteses direitos do que esquerdos\n");
             return EXIT_SUCCESS;
           }
           else if ( !parenteses_equivalentes (cpilha, exp[c]) )
                {
                  printf ("Parênteses %c e %c discordantes\n",\
                           cpilha, exp[c]);
                  return EXIT_SUCCESS;
                }
         }

  st = Stack_Pop (&cpilha);
  if (st == STACK_EMPTY)
       printf ("Expressão com parênteses balanceados\n");
  else printf ("Mais parênteses esquerdos do que direitos\n");

  return EXIT_SUCCESS;
}

int parenteses_equivalentes (char pesquerdo, char pdireito)
{
  switch (pesquerdo)
  {
    case '('  : return pdireito == ')'; break;
    case '['  : return pdireito == ']'; break;
    case '{'  : return pdireito == '}'; break;
    default   : return 0;
  }
}
```

Figura 7.34 - Programa que verifica o balanceamento dos parênteses numa expressão.

Pretende-se determinar se uma palavra é um palíndromo, que é uma palavra que se lê da mesma maneira da esquerda para a direita e da direita para esquerda. Ou seja, aquilo que normalmente se designa por capicua. Para que uma palavra seja uma capicua, os seus caracteres estão reflectidos em relação ao carácter central. Uma forma de detectar esta característica, consiste em comparar cada carácter com o seu simétrico. Por outras palavras, se a palavra for comparada com a palavra invertida, carácter a carácter, temos que os caracteres são coincidentes. Podemos fazer esta detecção utilizando uma fila e uma pilha conjuntamente. O algoritmo é o seguinte. Coloca-se todos os caracteres da palavra na fila e na pilha. Depois retira-se o carácter da cabeça da fila e o carácter do topo da pilha e comparam-se. Desta maneira, compara-se o primeiro carácter da palavra, ou seja, o carácter que se encontra na fila com o último carácter da palavra, ou seja, o carácter que se encontra na pilha. Se os caracteres forem iguais até se esgotar os caracteres da fila e da pilha, então a palavra é uma capicua. Caso contrário, a palavra não é uma capicua.

A Figura 7.35 apresenta o programa que implementa este algoritmo. A pilha e a fila utilizadas são a implementação estática. Para utilizar a fila e a pilha, primeiro é preciso editar o ficheiro de interface **elemento.h**, e definir o número de elementos das estruturas de dados da fila e da pilha, com um valor suficiente para armazenar o número de caracteres da palavra e definir também o tipo dos elementos da fila e da pilha como sendo do tipo ***char***. Depois os módulos são compilados, com a opção **−c**, para ficarem concretizados para uma fila e uma pilha de caracteres. Finalmente, o programa é compilado, indicando no comando de compilação os ficheiros objectos dos módulos **fila_est.o** e **pilha_est.o**.

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

#include "fila_est.h"   /* ficheiro de interface do módulo da fila */
#include "pilha_est.h" /* ficheiro de interface do módulo da pilha */

int main (void)
{
  char palavra[81],                      /* palavra lida do teclado */
       cfila,                          /* carácter armazenado na fila */
       cpilha;                        /* carácter armazenado na pilha */
  int  nc, c;

  printf ("Palavra -> "); scanf ("%80s", palavra);
  nc = strlen (palavra);

  for (c = 0; c < nc; c++)
  {
    Fifo_In (palavra[c]); Stack_Push (palavra[c]);
  }

  for (c = 0; c < nc; c++)
  {
    Fifo_Out (&cfila); Stack_Pop (&cpilha);
    if (cfila != cpilha)
    {
      printf ("A palavra não é uma capicua\n"); return EXIT_SUCCESS;
    }
  }

  printf ("A palavra é uma capicua\n");
  return EXIT_SUCCESS;
}
```

Figura 7.35 - Programa que verifica se uma palavra é uma capicua.

7.5 Implementações abstractas

Uma alternativa para poder reutilizar um módulo de memória sem a necessidade de o recompilar sempre que o utilizador precisa de mudar o tipo dos elementos da memória, consiste na criação de um módulo abstracto. Um módulo abstracto pode ser concretizado durante a execução do programa, permitindo assim a sua reutilização para diferentes tipos de dados. Para que um módulo de memória seja reutilizável sem limitações, ele deve também ter uma implementação dinâmica para que a sua capacidade de armazenamento seja ilimitada. Por outro lado, para implementar certos algoritmos, por vezes existe a necessidade de utilizar mais do que uma memória do mesmo tipo, pelo que, o módulo de memória também deve ter a capacidade de múltipla instanciação. Portanto, um módulo de memória abstracto, com uma implementação dinâmica e com capacidade de múltipla instanciação é um módulo muito versátil e com uma reutilização praticamente ilimitada.

Com a linguagem C é possível criar um módulo de memória usando ponteiros de tipo ***void***, uma vez que ele pode ser colocado a apontar para qualquer tipo de dados. Portanto, estamos perante um ponteiro genérico que possibilita a construção de um módulo de memória, em que o seu elemento de armazenamento pode ser de qualquer tipo de dados. Aquando da utilização do módulo é preciso concretizá-lo para o tipo de dados necessário à aplicação. Esta operação é feita através de uma função de criação do módulo, que é responsável pela caracterização do tamanho em *bytes* do elemento de armazenamento da memória. Para permitir manipular estruturas de dados, sem conhecimento do seu tipo concreto, a biblioteca de execução ANSI ***string*** providencia a função **memcpy**, que copia blocos de memória, sem atribuir qualquer interpretação ao conteúdo dos *bytes* copiados.

A Figura 7.36 apresenta o ficheiro de interface da fila dinâmica, abstracta e com capacidade de múltipla instanciação, cujo ficheiro de implementação se apresenta na Figura 7.37 e na Figura 7.38.

Para que o módulo possa criar e manipular mais do que uma fila, é necessário que exista uma referência para cada fila criada de forma a identificar de forma inequívoca a fila onde se pretende colocar ou retirar elementos. Para isso é preciso uma estrutura de suporte que possa armazenar os elementos que controlam a fila e que são os ponteiros para a cabeça e para a cauda da fila e um indicador do tamanho em número de *bytes* do elemento da fila. Esta estrutura vai ser criada na memória na altura em que é pedido a criação de uma fila e o seu endereço é devolvido de forma a permitir o posterior acesso à fila criada, quer para a colocação e remoção de elementos, quer para a destruição da fila, quando ela deixa de ser necessária. Pelo que, o ficheiro de interface define o tipo PtFifo, que é um ponteiro para **struct** fifo, que permite ao utilizador do módulo manipular as filas criadas, sem no entanto ter acesso à estrutura de dados da fila.

A função de criação da fila **Fifo_Create**, cria a estrutura de suporte da fila, coloca os ponteiros para a cabeça e a cauda da fila a apontar para NULL, ou seja, cria uma fila sem elementos, concretiza o tipo de elementos através da especificação do seu tamanho em *bytes* e devolve a referência da estrutura de suporte da fila criada.

Por sua vez, a função de destruição da fila **Fifo_Destroy**, liberta toda a memória ocupada pelos elementos da fila e pela estrutura de suporte da fila e coloca a referência da fila a NULL, de maneira a não ser mais possível aceder à fila.

A função de colocação de elementos na fila **Fifo_In** começa por assegurar que a fila existe e depois comporta-se tal como na implementação dinâmica. Mas, a cópia do elemento para a fila é feita pela função **memcpy** indicando o ponteiro para o elemento a colocar na fila, o ponteiro para a cauda da fila e o número de *bytes* a copiar, que está armazenado no campo que especifica o tamanho dos elementos da fila. Desta maneira é possível manipular os elementos da fila sem que se saiba de que tipo eles são.

A função de remoção de elementos da fila **Fifo_Out** começa por assegurar que a fila existe e depois comporta-se tal como na implementação dinâmica. Tal como na função **Fifo_In**, a cópia do elemento da fila é feita pela função **memcpy**.

```
 /****************** Interface da FILA Abstracta ******************/
                       /* Nome : fila_abs.h */

#ifndef _FILA_ABSTRACTA
#define _FILA_ABSTRACTA

typedef struct fifo *PtFifo;

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NULL_SIZE   2   /* tamanho nulo */
#define NO_MEM      3   /* memória esgotada */
#define FIFO_EMPTY  4   /* fila vazia */
#define NO_FIFO     7   /* não foi instanciada qualquer fila */

           /* Alusão às Funções Exportadas pelo Módulo */

PtFifo Fifo_Create (unsigned int sz);
/* Concretiza a fila para elementos de sz bytes. Devolve a
referência da fila criada ou NULL em caso de erro. */

int Fifo_Destroy (PtFifo *fila);
/* Destrói a fila referenciada por fila e coloca a referência a
NULL. Valores de retorno: OK ou NO_FIFO. */

int Fifo_In (PtFifo fila, void *pelemento);
/* Coloca o elemento apontado por pelemento na cauda da fila
referenciada por fila. Valores de retorno: OK, NO_FIFO, NULL_PTR ou
NO_MEM. */

int Fifo_Out (PtFifo fila, void *pelemento);
/* Retira o elemento da cabeça da fila referenciada por fila, para o
elemento apontado por pelemento. Valores de retorno: OK, NO_FIFO,
NULL_PTR ou FIFO_EMPTY. */

#endif
```

Figura 7.36 - Ficheiro de interface da fila abstracta com múltipla instanciação.

```
 /**************** Implementação da FILA Abstracta ***************/
                        /* Nome : fila_abs.c */

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

#include "fila_abs.h"           /* Ficheiro de interface do módulo */

        /* Definição da Estrutura de Dados Interna da FILA */

typedef struct no *PtNo;

struct no
{
  void *pelemento;                    /* ponteiro para o elemento */
  PtNo pseg;                        /* ponteiro para o nó seguinte */
};

struct fifo
{
  unsigned int size;/* tamanho em número de bytes de cada elemento */
  PtNo head;                                      /* cabeça da fila */
  PtNo tail;                                       /* cauda da fila */
};

                     /* Definição das Funções */

PtFifo Fifo_Create (unsigned int sz)
{
  PtFifo fifo;

  if (sz == 0) return NULL;

  if ((fifo = (PtFifo) malloc (sizeof (struct fifo))) == NULL)
    return NULL;

  fifo->size = sz;
  fifo->head = NULL;
  fifo->tail = NULL;

  return fifo;
}

int Fifo_Destroy (PtFifo *fila)
{
  PtFifo fifo = *fila; PtNo tmp;

  if (fifo == NULL) return NO_FIFO;

  while ( fifo->head != NULL )
  {
    tmp = fifo->head;
    fifo->head = fifo->head->pseg;
    free (tmp->pelemento);
    free (tmp);
  }

  free (fifo);
  *fila = NULL;

  return OK;
}
```

Figura 7.37 - Ficheiro de implementação da fila abstracta com múltipla instanciação (1ª parte).

```
int Fifo_In (PtFifo fila, void *pelemento)
{
   PtFifo fifo = fila; PtNo tmp;

   if (fifo == NULL) return NO_FIFO;

   if (pelemento == NULL) return NULL_PTR;

   if ((tmp = (PtNo) malloc (sizeof (struct no))) == NULL)
      return NO_MEM;

   if ((tmp->pelemento = (void *) malloc (fifo->size)) == NULL)
   {
      free (tmp); return NO_MEM;
   }

   tmp->pseg = NULL;
   if (fifo->tail == NULL) fifo->head = tmp;
   else fifo->tail->pseg = tmp;
   fifo->tail = tmp;

   memcpy (fifo->tail->pelemento, pelemento, fifo->size);

   return OK;
}

int Fifo_Out (PtFifo fila, void *pelemento)
{
   PtFifo fifo = fila; PtNo tmp;

   if (fifo == NULL) return NO_FIFO;

   if (pelemento == NULL) return NULL_PTR;

   if (fifo->head == NULL) return FIFO_EMPTY;

   memcpy (pelemento, fifo->head->pelemento, fifo->size);

   tmp = fifo->head;
   fifo->head = fifo->head->pseg;
   if (fifo->head == NULL) fifo->tail = NULL;

   free (tmp->pelemento);
   free (tmp);

   return OK;
}
```

Figura 7.38 - Ficheiro de implementação da fila abstracta com múltipla instanciação (2ª parte).

A Figura 7.39 apresenta o ficheiro de interface da pilha dinâmica, abstracta e com capacidade de múltipla instanciação, cujo ficheiro de implementação se apresentam na Figura 7.40 e na Figura 7.41.

Para que o módulo possa criar e manipular mais do que uma pila, é necessário que exista uma referência para cada pilha criada de forma a identificar de forma inequívoca a pilha onde se pretende colocar ou retirar elementos. Para isso é preciso uma estrutura de suporte que possa armazenar os elementos que controlam a pilha e que são o ponteiro para o topo da pilha e um indicador do tamanho em número de *bytes* do elemento da pilha. Esta estrutura vai ser criada na memória na altura em que é pedido a criação de uma pilha e o seu endereço é devolvido de forma a permitir o posterior acesso à pilha criada, quer para a colocação e remoção de elementos, quer para a destruição da pilha, quando ela deixa de ser necessária. Pelo que, o ficheiro de interface define o tipo PtStack, que é um ponteiro para **struct** stack, que permite ao utilizador do módulo manipular as filas criadas, sem no entanto ter acesso à estrutura de suporte da fila.

A função de criação da pilha **Stack_Create**, cria a estrutura de suporte da pilha, coloca o ponteiro para o topo da pilha a apontar para NULL, ou seja, cria uma pilha sem elementos, concretiza o tipo de elementos através da especificação do seu tamanho em *bytes* e devolve a referência da estrutura de suporte da pilha criada.

Por sua vez, a função de destruição da pilha **Stack_Destroy**, liberta toda a memória ocupada pelos elementos da pilha e pela estrutura de suporte da pilha e coloca a referência da pilha a NULL, de maneira a não ser mais possível aceder à pilha.

A função de colocação de elementos na pilha **Stack_Push** começa por assegurar que a pilha existe e depois comporta-se tal como na implementação dinâmica. Mas, a cópia do elemento para a pilha é feita pela função **memcpy** indicando o ponteiro para o elemento a colocar na pilha, o ponteiro para o topo da pilha e o número de *bytes* a copiar, que está armazenado no campo que especifica o tamanho dos elementos da pilha. Desta maneira é possível manipular os elementos da pilha sem que se saiba de que tipo eles são.

A função de remoção de elementos da pilha **Stack_Pop** começa por assegurar que a pilha existe e depois comporta-se tal como na implementação dinâmica. Tal como na função **Stack_Push**, a cópia do elemento da pilha é feita pela função **memcpy**.

```
/****************** Interface da PILHA Abstracta ******************/
                        /* Nome : pilha_abs.h */

#ifndef _PILHA_ABSTRACTA
#define _PILHA_ABSTRACTA

typedef struct stack *PtStack;

                    /* Definição de Constantes */

#define OK          0   /* operação realizada com sucesso */
#define NULL_PTR    1   /* ponteiro nulo */
#define NULL_SIZE   2   /* tamanho nulo */
#define NO_MEM      3   /* memória esgotada */
#define STACK_EMPTY 4   /* pilha vazia */
#define NO_STACK    7   /* não foi instanciada qualquer pilha */

            /* Alusão às Funções Exportadas pelo Módulo */

PtStack Stack_Create (unsigned int sz);
/* Concretiza a pilha para elementos de sz bytes. Devolve a
referência da pilha criada ou NULL em caso de erro. */

int Stack_Destroy (PtStack *pilha);
/* Destrói a pilha referenciada por pilha e coloca a referência a
NULL. Valores de retorno: OK ou NO_STACK. */

int Stack_Push (PtStack pilha, void *pelemento);
/* Coloca o elemento apontado por pelemento no topo da pilha
referenciada por pilha. Valores de retorno: OK, NO_STACK, NULL_PTR
ou NO_MEM. */

int Stack_Pop (PtStack pilha, void *pelemento);
/* Retira o elemento do topo da pilha referenciada por pilha, para o
elemento apontado por pelemento. Valores de retorno: OK, NO_STACK,
NULL_PTR ou STACK_EMPTY. */

#endif
```

Figura 7.39 - Ficheiro de interface da pilha abstracta com múltipla instanciação.

```
/*************** Implementação da PILHA Abstracta ***************/
                      /* Nome : pilha_abs.c */

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

#include "pilha_abs.h"          /* Ficheiro de interface do módulo */

       /* Definição da Estrutura de Dados Interna da PILHA */

typedef struct no *PtNo;

struct no
{
  void *pelemento;                    /* ponteiro para o elemento */
  PtNo pant;                        /* ponteiro para o nó anterior */
};

struct stack
{
  unsigned int size;/* tamanho em número de bytes de cada elemento */
  PtNo top;                                        /* topo da pilha */
};
                     /* Definição das Funções */

PtStack Stack_Create (unsigned int sz)
{
  PtStack stack;

  if (sz == 0) return NULL;

  if ((stack = (PtStack) malloc (sizeof (struct stack))) == NULL)
    return NULL;

  stack->size = sz;
  stack->top = NULL;

  return stack;
}

int Stack_Destroy (PtStack *pilha)
{
  PtStack stack = *pilha; PtNo tmp;

  if (stack == NULL) return NO_STACK;

  while (stack->top != NULL)
  {
    tmp = stack->top;
    stack->top = stack->top->pant;
    free (tmp->pelemento);
    free (tmp);
  }

  free (stack);
  *pilha = NULL;

  return OK;
}
```

Figura 7.40 - Ficheiro de implementação da pilha abstracta com múltipla instanciação (1ª parte).

```
int Stack_Push (PtStack pilha, void *pelemento)
{
   PtStack stack = pilha; PtNo tmp;

   if (stack == NULL) return NO_STACK;

   if (pelemento == NULL) return NULL_PTR;

   if ((tmp = (PtNo) malloc (sizeof (struct no))) == NULL)
      return NO_MEM;

   if ((tmp->pelemento = (void *) malloc (stack->size)) == NULL)
   {
      free (tmp); return NO_MEM;
   }

   tmp->pant = stack->top;
   stack->top = tmp;

   memcpy (stack->top->pelemento, pelemento, stack->size);

   return OK;
}

int Stack_Pop (PtStack pilha, void *pelemento)
{
   PtStack stack = pilha; PtNo tmp;

   if (stack == NULL) return NO_STACK;

   if (pelemento == NULL) return NULL_PTR;

   if (stack->top == NULL) return STACK_EMPTY;

   memcpy (pelemento, stack->top->pelemento, stack->size);

   tmp = stack->top;
   stack->top = stack->top->pant;

   free (tmp->pelemento);
   free (tmp);

   return OK;
}
```

Figura 7.41 - Ficheiro de implementação da pilha abstracta com múltipla instanciação (2ª parte).

A Figura 7.42 apresenta a utilização da pilha abstracta para a resolução do problema da dupla inversão de uma linha de texto. É um exemplo meramente académico, uma vez que para inverter uma cadeia de caracteres não é necessário uma pilha, mas serve para mostrar a utilização de duas pilhas em simultâneo no mesmo programa. Os caracteres são colocados numa pilha e depois ao serem retirados são escritos no monitor por ordem inversa. Mas, se forem de novo colocados noutra pilha, depois ao serem retirados da segunda pilha são escritos no monitor pela ordem inicial. Temos assim uma dupla inversão.

Para podermos usar duas pilhas, primeiro é preciso declarar duas variáveis de tipo PtStack, que vão apontar para as pilhas criadas através da invocação da função **Stack_Create**. Na invocação desta função é indicado que os elementos da pilha devem ter o tamanho de um ***char***, pelo que estamos a criar pilhas para caracteres. Para colocar e retirar caracteres numa pilha é obrigatório passar às funções **Stack_Push** e **Stack_Pop**, como parâmetro de entrada, a pilha que se está a processar usando o respectivo ponteiro. Assim que as pilhas não são mais precisas, elas devem ser destruídas invocando a função **Stack_Destroy** para cada uma das pilhas. Para simplificar o programa, o resultado de saída das operações de colocação e remoção de caracteres nas pilhas não é testado para conferir se são bem sucedidas ou não.

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

#include "pilha_abs.h" /* ficheiro de interface do módulo da pilha */

int main (void)
{
  char exp[81], cpilha;
  int nc, c, st;
  PtStack pilha1 = NULL, pilha2 = NULL;

  pilha1 = Stack_Create (sizeof (char));     /* Criação da pilha 1 */
  pilha2 = Stack_Create (sizeof (char));     /* Criação da pilha 2 */

  printf ("Texto de entrada -> ");  scanf ("%80s", exp);
  nc = strlen (exp);

  for (c = 0; c < nc; c++)
    st = Stack_Push (pilha1, &exp[c]); /* Colocar texto na pilha 1 */

  printf ("Texto de saída depois de colocado na pilha 1 -> ");
  for (c = 0; c < nc; c++)
  {
    st = Stack_Pop (pilha1, &cpilha);  /* Retirar texto da pilha 1 */
    printf ("%c", cpilha);
    st = Stack_Push (pilha2, &cpilha); /* Colocar texto na pilha 2 */
  }
  printf ("\n");

  printf ("Texto de saída depois de colocado na pilha 2 -> ");
  for (c = 0; c < nc; c++)
  {
    st = Stack_Pop (pilha2, &cpilha);  /* Retirar texto da pilha 2 */
    printf ("%c", cpilha);
  }
  printf ("\n");

  Stack_Destroy (&pilha1);                /* Destruição da pilha 1 */
  Stack_Destroy (&pilha2);                /* Destruição da pilha 2 */

  return EXIT_SUCCESS;
}
```

Figura 7.42 - Exemplo de utilização da pilha abstracta com múltipla instanciação.

7.6 Leituras recomendadas

• 7º capítulo do livro "Data Structures, Algorithms and Software Principles in C", de Thomas A. Standish, da editora Addison-Wesley Publishing Company, 1995.