Livro *bestseller* de programação!

ti

Luís Damas

# Linguagem C

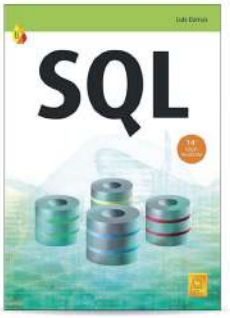
25.ª Edição Atualizada e Aumentada
Com 12 novos capítulos

FCA

**Luís Damas**

Licenciado em Informática pela Faculdade de Ciências da Universidade de Lisboa e mestre em Gestão de Informação pela Universidade Católica Portuguesa. Além do desenvolvimento de aplicações e consultadoria na área de Informática, tem lecionado disciplinas de programação, bases de dados, algoritmos e estruturas de dados e sistemas de informação em diversas instituições de ensino superior.

*Do mesmo autor:*



# ÍNDICE GERAL

# SOBRE O LIVRO

Nesta 25.ª edição, procedeu-se a uma atualização do texto e a uma revisão detalhada do conteúdo do livro, tendo como objetivo principal fornecer aos leitores uma abordagem ainda mais clara, prática e completa sobre a linguagem C e promovendo uma aprendizagem sólida e eficaz, o que reflete a evolução que tem existido na linguagem e nas práticas do setor.

Todos os capítulos da edição anterior foram revistos, aplicando o novo acordo ortográfico, e os conteúdos de alguns deles foram significativamente expandidos, como operações *bit-a-bit*, apontadores para funções e outros.

Com o intuito de aprofundar o conhecimento técnico dos leitores, nesta edição foi adicionada uma segunda parte que contém 12 novos capítulos inteiramente dedicados aos algoritmos e estruturas de dados e algumas técnicas de programação, fornecendo a preparação necessária para a evolução dos leitores para tópicos de programação avançada.

Os novos capítulos incluem:

- Recursividade;
- Notação *Big-O*;
- Programação modular;
- Tipos Abstratos de Dados (TAD);
- Ordenação (*Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Merge Sort*);
- Pesquisa (sequencial e binária);
- Pilhas, filas e listas (normais, circulares e de prioridades);
- Conjuntos;
- Árvores e árvores binárias de pesquisa;
- Tabelas de dispersão (*Hashing*).

Reconhecendo a importância de algumas ferramentas especiais no desenvolvimento moderno em C, foram incluídas breves referências práticas aos utilitários *valgrind* (para deteção de fugas de memória) e *nm* (para inspeção de símbolos em ficheiros binários).

O código de todos os exercícios práticos encontra-se disponível para *download* na página do livro em www.fca.pt.

Alguns números a considerar:

- Mais de 370 programas completos;
- Mais de 1000 funções implementadas;
- Mais de 70 páginas com as soluções dos exercícios propostos, acedidas através de códigos QR;
- Mais de 35 000 linhas de código.

# NOTA INTRODUTÓRIA

Bem-vindos ao mundo da programação em C!

Este livro irá apresentar as características de uma linguagem de programação que se tornou um fenómeno de popularidade, a linguagem C.

Dicas e truques usados pelos programadores mais experientes serão também apresentados através de múltiplos exemplos explicados e comentados.

A primeira edição deste livro descreveu as características da linguagem. Nesta nova edição, foram acrescentados 12 novos capítulos direcionados para a área dos algoritmos e das estruturas de dados implementados em C. Existem também capítulos adicionais que cobrem tópicos genéricos, como recursividade, notação *Big-O* e tipos abstratos de dados.

No final de alguns capítulos, serão propostos exercícios práticos relacionados com a matéria dos mesmos, cuja solução se encontra disponível através de códigos QR.

**NOTA**

→ As soluções vão ser acedidas através de um código QR disponibilizado no final de cada capítulo.

## BREVE HISTÓRIA DA LINGUAGEM C

Embora possua um nome estranho quando comparada com outras linguagens de terceira geração, como o FORTRAN, o PASCAL ou o COBOL, a linguagem C foi criada em 1972, nos Bell Telephone Laboratories, por Dennis Ritchie e tinha por finalidade permitir a escrita de um sistema operativo (o Unix), utilizando uma linguagem de relativo alto nível e evitando, assim, o recurso ao *Assembly*.

Devido às suas capacidades e características, e através da divulgação do sistema Unix pelas universidades dos Estados Unidos da América, a linguagem C cedo deixou as portas dos Laboratórios Bell, disseminou-se e tornou-se conhecida entre os programadores, independentemente dos projetos em que estivessem envolvidos, sendo o livro *The C Programming Language*, de Kernighan e Ritchie (1978)[i], a principal fonte de informação sobre a linguagem editado por essa altura.

Esta dispersão levou a que diferentes organizações desenvolvessem e utilizassem diferentes versões da linguagem C, criando, assim, alguns problemas de portabilidade, e não só.

Em 1983, o ANSI (American National Standards Institute) criou um Comité para a definição de um padrão para a linguagem C, o que culminou no desenvolvimento do padrão ANSI C em 1989, conhecido como C89, revisto posteriormente nos padrões C99, C11 e C17.

O nome da linguagem (e a própria linguagem) resulta da evolução de uma outra linguagem de programação, desenvolvida por Ken Thompson também nos Laboratórios Bell e denominada B. Assim, é perfeitamente natural que a evolução da linguagem **B** viesse a dar origem a uma nova linguagem, denominada **C**.

[i] Kernighan, B. W. & Ritchie, D. M. (1978). *The C Programming Language.* Prentice-Hall.

# PORQUÊ PROGRAMAR EM C?

A linguagem C é de tal forma impactante na evolução das linguagens de programação que quase todas as linguagens de programação que lhe sucederam usam a sintaxe igual ou baseada nas instruções da linguagem C.

Qual é a área de desenvolvimento a que a linguagem se destina?

A resposta é "nenhuma em particular". É aquilo que, habitualmente, se denomina por *general purpose*, o que representa uma das suas grandes vantagens, pois a linguagem C adapta-se ao desenvolvimento de qualquer tipo de projeto, como sistemas operativos, interfaces gráficas e processamento de registos. A linguagem C foi também utilizada para escrever o próprio compilador da linguagem (e também para escrever os compiladores de outras linguagens).

Principais características da linguagem C:

- **Paradigma imperativo** – O C é uma linguagem imperativa extremamente potente e flexível na qual as instruções são executadas numa determinada sequência, alterando o estado das variáveis de um programa, começando cada instrução apenas quando a anterior termina;
- **Rapidez** – A linguagem C consegue obter *performances* semelhantes às obtidas pelo *Assembly*, usando instruções de alto nível;
- **Simples** – A sua sintaxe é extremamente simples e o número de palavras reservadas, tipos de dados básicos e operadores é diminuto, reduzindo, assim, a quantidade de tempo e esforço necessários à aprendizagem da linguagem;
- **Portável** – Existem *standards* [ANSI e ISO (International Organization for Standardization)] específicos que definem as características a que qualquer compilador da linguagem deve obedecer. Deste modo, o código escrito numa máquina pode ser transportado para outra máquina e compilado sem qualquer alteração (ou com um número reduzido de alterações);
- **Popular** – A linguagem C é universalmente conhecida e reconhecida como um marco no desenvolvimento das linguagens de programação, tendo servido de base para um grande número de linguagens de programação que evoluíram a partir dela, como o C++, o Java, o JavaScript, o C#, o Kotlin e o Python. Está muito documentada em livros, revistas da especialidade, manuais, etc. Existem compiladores para todo o tipo de arquiteturas e computadores;
- **Modular** – A linguagem C permite o desenvolvimento modular de aplicações, facilitando a separação de projetos em módulos distintos e independentes, com recurso à utilização de funções específicas dentro de cada módulo;
- **Alto nível** – A linguagem C permite também o acesso à maior parte das funcionalidades do *Assembly*, utilizando expressões e instruções de alto nível. É possível, por exemplo, aceder à memória diretamente, utilizando o endereço de qualquer objeto (seja variável ou função), e manipular diretamente a memória pertencente a esses objetos sem qualquer tipo de restrições, o que aumenta a flexibilidade da linguagem;
- **Bibliotecas muito poderosas** – A linguagem C contém um número reduzido de palavras-chave, o que indica que as capacidades da linguagem são muito limitadas, e, na realidade, são, mas esta é precisamente uma característica da linguagem. A maior parte das funcionalidades que a linguagem tem são-lhe adicionadas pela utilização de funções que existem em bibliotecas adicionais e que realizam todo o tipo de tarefas, desde a escrita de um simples caráter no ecrã, até ao processamento de ficheiros, à movimentação de dados na memória do computador, etc.;

- Criar o executável final (opção *-o*) *myprog* a partir do ficheiro-objeto *myprog.o*:

```
$ cc myprog.o -o myprog
```

- Compilar e "linkar" o ficheiro *myprog.c*, gerando o executável final (opção *-o=output*) *myprog*:

```
$ cc myprog.c -o myprog
ou
$ make myprog
```

4. **Execução do programa** – Se o processo de "linkagem" terminar com sucesso, temos, então, disponível um ficheiro executável. Se é executável, podemos executá-lo digitando o seu nome:

```
$ myprog
ou
$ ./myprog
```

Se o programa não faz aquilo que deveria fazer, é porque o elemento mais fraco de todo este processo – o programador – se enganou. Deverá, então, editar o código fonte, alterá-lo e voltar a realizar todo o processo.

**NOTA**

O ciclo de desenvolvimento de uma aplicação anteriormente apresentado é verdadeiro para a maioria das linguagens. No entanto, a linguagem C adiciona uma fase antes da compilação e que tem por missão expandir todas as macros e executar todas as diretivas para o pré-processador. Mais informações sobre esta fase podem ser obtidas no Capítulo 13.

Para ver o resultado do trabalho do pré-processador, pode usar a opção *-E* do compilador:

```
$ cc -E myprog.c
```

Podemos, então, afirmar que, no caso da linguagem C, o ciclo de desenvolvimento de uma aplicação decorre nas seguintes cinco fases:

1. Escrita do código-fonte (programador).
2. Pré-processamento (pré-processador).
3. Compilação (compilador).
4. "Linkagem" (*linker*).
5. Execução do programa.

## COMPOSIÇÃO DESTE LIVRO

Este livro foi escrito com o objetivo de servir como fonte de referência da linguagem para estudantes, sendo, por isso, apresentados múltiplos exemplos de programas escritos na linguagem ao longo de todo o documento e para todos os assuntos em análise.

Pretende-se que a aprendizagem seja baseada em conceitos teóricos claros e que estes sejam solidificados com a realização de um número alargado de exercícios práticos.

Sempre que possível, será apresentado o código de um ou mais programas para exemplificar a matéria descrita.

Na maior parte dos casos, cada programa é denominado seguindo a seguinte terminologia: ***prog****nnkk****.c***, em que *nn* representa o número do capítulo a que o programa pertence e *kk* o número do programa dentro desse capítulo.

Por exemplo, `prog0305.c` é o nome do quinto programa apresentado no Capítulo 3 deste livro.

Os programas possuem sempre o seguinte aspeto, sendo o nome do programa colocado por baixo da listagem do código:

```
1: #include <stdio.h>
2: main()
3: {
4:   printf("Hello World\n");
5: }
```

**PROG0103.C**

Antes de cada linha de código, é colocado o número da linha, apenas por simplicidade de representação e referência. No entanto, ao escrever um programa, nunca coloque o número da linha, pois irá obter um erro de compilação.

A sintaxe das várias instruções ao longo do livro é apresentada através do seguinte formato:

**Exemplo:** A definição de variáveis faz-se utilizando a seguinte sintaxe:

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

O uso de parênteses retos na sintaxe indica que a componente dentro dos parênteses é opcional.

Sempre que se pretende representar a execução de um programa, este é representado como sendo executado no *prompt* **$**[ii]:

```
$ prog0103
Hello World
$
```

No exemplo anterior, é executado o programa `prog0103`, que escreveu no ecrã a mensagem *Hello World*, deixando, em seguida, o cursor na linha seguinte (onde aparece novamente o *prompt*).

Embora seja possível executar os comandos no ambiente Linux tal como mostrados no exemplo anterior, normalmente, é necessário indicar que o executável se encontra na pasta atual, colocando o nome do programa precedido por um ponto e uma barra:

```
$ ./prog0103
Hello World
$
```

Por simplicidade, a forma como se mostra a execução de comandos/programas ao longo do livro é apresentada no formato `$ progxxyy`.

Contudo, muito provavelmente, para executar algo no seu sistema Linux deverá preceder o comando da pasta atual onde se encontra: `$ ./progxxyy`.

[ii] O *prompt* varia de sistema para sistema e é, habitualmente, **C:\>** na linha de comando do Windows e **#** ou **$** nos sistemas Linux mais comuns.

# 3

# INSTRUÇÕES CONDICIONAIS *if* E *switch*

## OBJETIVOS

- Valores lógicos *verdade* e *falso*
- Expressões e condições
- Operadores relacionais (**==**, **!=**, **>, <, >=**, **<=**)
- Operadores lógicos (**&&**, **||**, **!**)
- Instruções *if-else* e *switch*
- Instrução *break*
- Instruções *if-else* encadeadas
- Operador ternário *(cond) ? exp1 : exp2*
- Precedência entre operadores
- Blocos de instruções
- Indentação

## 3.1 INTRODUÇÃO

Os programas que escrevemos até este momento estão particularmente adaptados a um mundo "perfeito", sem erros, dúvidas ou qualquer tipo de variações. As instruções seguem-se umas às outras, seguindo sempre a mesma ordem de execução, quaisquer que sejam os valores de entrada.

Com a matéria apresentada até agora, podíamos escrever um programa que tivesse como objetivo preparar uma pessoa para o seu dia de trabalho. Seria algo parecido a:

```
Vestir camisa;
Vestir camisola;
Vestir calças;
Calçar sapatos;
Pentear
...
```

No entanto, basta pensar num dia comum e verificar que, mal nos levantamos, uma das primeiras coisas que fazemos é verificar o estado do tempo, de forma a decidir se devemos ir mais ou menos agasalhados.

No caso de nos encontrarmos em pleno inverno, o programa anterior poderia estar mais ou menos adaptado, no entanto, caso o sol brilhasse de forma abrasadora, talvez não fosse a melhor maneira de nos equiparmos para o trabalho, pois não faz qualquer sentido vestir camisolas durante o verão.

Qual seria, então, a solução? Se retirarmos a instrução "*Vestir camisola*", o programa fica adaptado ao verão, mas não ao inverno.

A solução passa por manter a linha de código com a instrução "*Vestir camisola*" no programa, mas executá-la apenas quando a temperatura for suficientemente baixa.

Ao contrário dos programas anteriormente apresentados, em que todas as instruções eram sempre executadas, podemos escrever programas que tenham a capacidade de tomar decisões sobre que instruções executar:

```
Vestir camisa;
Se estiver um frio de rachar então
  Vestir camisola;
Vestir calças;
Calçar capatos;
Pentear;
…
```

## 3.2 VALORES LÓGICOS – *VERDADE* E *FALSO*

A linguagem C possui apenas quatro tipos de dados (*int*, *float*, *char* e *double*). Não existe, por isso, nenhum tipo que permita representar os valores lógicos *verdade* e *falso*.

Existem linguagens de programação que disponibilizam um tipo específico para representar valores lógicos (por exemplo, *Boolean*, *bool*).

**NOTA**

Em C, não existe um tipo específico de dados para armazenar valores lógicos.

Em C, qualquer valor *0* (zero) poderá ser interpretado como o valor lógico *falso*.

Qualquer valor diferente de *0* (zero) representa o valor lógico *verdade*.

**Exemplos:**

```
Falso   : 0, 0.0, '\0', NULL[4]
Verdade : 2, -3, 123.45, 0.000001, "hello", '@', '\n', ""
```

**NOTA**

O valor lógico *verdade* em C não é o valor *1*, mas qualquer valor diferente de *0* (zero). O valor *1* é apenas um dos valores possíveis para representar o valor lógico *verdade*.

Os valores lógicos podem resultar da avaliação de afirmações:

- A terra é quadrada. (*falso*);

[4] '\0' representa um *char* com o número zero (ASCII 0). *NULL* é uma constante especial em C que representa o endereço de memória número 0.

# 5

# FUNÇÕES

## OBJETIVOS

- Funções e procedimentos
- Cabeçalho e corpo de uma função
- Parâmetros e argumentos
- Passagem de parâmetros de tipos básicos
- Valor de retorno (*return*)
- O "tipo" *void*
- Protótipos de funções
- Variáveis locais
- *Scope* das variáveis

## 5.1 INTRODUÇÃO

Embora ainda não saibamos como escrever uma função, já as temos utilizado ao longo dos programas apresentados neste livro. São exemplos as funções *printf*, *scanf*, *getchar*, *putchar*, etc., que fazem parte da biblioteca de funções *standard* do C e que estão disponíveis e acompanham o compilador da linguagem.

Neste capítulo, iremos aprender como escrever funções e procedimentos, como comunicar com as funções através da passagem de parâmetros e como devolver algum valor como resultado do processamento de uma função.

É indispensável que qualquer programador de C (ou de outra qualquer linguagem de programação) domine em absoluto a escrita de programas de forma modular, através de procedimentos e funções.

**Problema:** Escreva um programa que coloque no ecrã o seguinte *output*, escrevendo cada linha de 20 asteriscos através de um ciclo *for*.

```
********************
Números entre 1 e 5
********************
1
2
3
4
5
********************
```

```
 1: #include <stdio.h>
 2:
 3: int main(void)
 4: {
 5:   // Escrever o cabeçalho
 6:   for (int i=1 ; i<=20 ; i++)
 7:     putchar('*');
 8:   putchar('\n');
 9:
10:   puts("Números entre 1 e 5");
11:
12:   for (int i=1 ; i<=20 ; i++)
13:     putchar('*');
14:   putchar('\n');
15:
16:   // Escrever os valores 1..5
17:   for (int i=1; i<=5 ; i++)
18:     printf("%d\n",i);
19:
20:   // Escrever o rodapé
21:   for (int i=1 ; i<=20 ; i++)
22:     putchar('*');
23:   putchar('\n');
24:
25:   return 0;
26: }
```

**PROG0501.C**

Como se pode observar, o conjunto de código utilizado para escrever uma linha de asteriscos no ecrã aparece repetido três vezes:

```
for (int i=1 ; i<=20 ; i++)
  putchar('*');
putchar('\n');
```

O ideal seria escrever esta porção de código uma única vez e poder fazer a sua invocação sempre que fosse necessário.

**Problema:** Escreva um programa que coloque uma linha com 20 asteriscos no ecrã.

```
 1: #include <stdio.h>
 2:
 3: int main(void)
 4: {
 5:   for (int i=1 ; i<=20 ; i++)
 6:     putchar('*');
 7:   putchar('\n');
 8:
 9:   return 0;
10: }
```

**PROG0502.C**

O programa anterior coloca no ecrã uma linha com 20 asteriscos. Foi implementado no contexto da escrita de um programa e todo o código foi colocado dentro da função *main*. Como a sua função é escrever uma linha, em vez de lhe chamarmos *main*, vamos chamar-lhe *linha*.

```
1: #include <stdio.h>
2:
3: int linha(void)
4: {
```

**5.5** ***float getVAL(float x, int n, float t)***

Devolve o VAL (Valor Atual Líquido) para *n* anos, à taxa *t*, que é definido através da seguinte fórmula:

$$VAL = \frac{x}{(1+t)} + \frac{x}{(1+t)^2} + \frac{x}{(1+t)^3} + \ldots + \frac{x}{(1+t)^n}$$

**Sugestão:** Utilize a função *my_power* implementada anteriormente (consultar `PROG0514.C`).

**5.6** ***long int n_segundos(int n_horas)***

Devolve o número de segundos que um conjunto de horas tem:

```
n_segundos(0)      → 0
n_segundos(1)      → 3600
n_segundos(2)      → 7200
```

**5.7** ***long int num(int n_horas, char tipo)***

Semelhante à função anterior, mas recebe um parâmetro adicional que indica o que se pretende obter: 'h' (horas), 'm' (minutos) e 's' (segundos):

```
num(3, 'h')      → 3
num(3, 'm')      → 180
num(3, 's')      → 10800
```

Resolva este exercício de três formas distintas, usando:

**a)** *if-else*;

**b)** *switch* com *break*;

**c)** *switch* sem *break*.

**NOTA**

Supõe-se que o valor do parâmetro *tipo* está sempre correto.

**5.8** ***float max3(float x, float y, float w)***

Devolve o maior dos valores *x*, *y* e *w*.

**5.9** ***int impar(int x)***

Devolve *verdade*, se *x* for ímpar, e *falso*, caso contrário.

**5.10** ***int Entre(int x, int lim_inf, int lim_sup)***

Verifica se *x* se encontra no intervalo **`lim_inf <= x <= lim_sup`**.

**5.11** Escreva as seguintes funções:

| Função | Devolve |
|---|---|
| ***int isdigit(int ch)*** | *Verdade*, caso *ch* seja um dígito, e *falso*, caso contrário |
| ***int isalpha(int ch)*** | *Verdade*, caso *ch* seja uma letra do alfabeto, maiúscula ou minúscula, e *falso*, caso contrário |
| ***int isalnum(int ch)*** | *Verdade*, caso *ch* seja um dígito ou uma letra do alfabeto, e *falso*, caso contrário |
| ***int islower(int ch)*** | *Verdade*, caso *ch* seja uma letra minúscula, e *falso*, caso contrário |
| ***int isupper(int ch)*** | *Verdade*, caso *ch* seja uma letra maiúscula, e *falso*, caso contrário |

(continua)

(continuação)

| Função | Devolve |
|---|---|
| ***int isspace(int ch)*** | *Verdade*, caso *ch* seja um espaço ou um *tab*, e *falso*, caso contrário |
| ***int tolower(int ch)*** | Devolve o valor do caráter *ch* transformado em minúsculas |
| ***int toupper(int ch)*** | Devolve o valor do caráter *ch* transformado em maiúsculas |

**NOTA**

Para ter acesso às funções apresentadas no Exercício resolvido 5.11 deste capítulo, bastará incluir no início do seu programa a seguinte linha:

```
#include <ctype.h>      // Funções sobre o tipo char  (ctype = char type)
```

**5.12** ***int isSquare(int x, int y)***

Devolve um valor lógico que indica se *x* é ou não igual a $y^2$.

**5.13** ***int Minus(int valor)***

Devolve o *valor* recebido sempre como número negativo:

```
Minus(10)      → -10
Minus(-10)     → -10
```

**5.14** ***int isSpecial(int x)***

Devolve um valor lógico que indica se o dobro de *x* é igual a $x^2$.

**5.15** ***int Cubo(int x)***

Devolve o valor de $x^3$.

**5.16** ***int isVowel(char ch)***

Verifica se *ch* é uma das vogais do alfabeto (em minúsculas ou maiúsculas).

**5.17** ***double Inverso(int x)***

Devolve o valor inverso de *x (1/x)* ou zero:

```
Inverso(0) → 0
Inverso(5) → 0.2
```

**5.18** ***void Triangulo(int n)***

Desenha no ecrã um triângulo encostado à direita, com uma base de *n* asteriscos:

```
Triangulo(3)      Triangulo(5)
  *                    *
 **                   **
***                  ***
                    ****
                   *****
```

Aceda aqui às soluções

# 20

# ORDENAÇÃO (*SORTING*)

## OBJETIVOS

- Algoritmo *Bubble Sort*
- Algoritmo *Selection Sort*
- Algoritmo *Insertion Sort*
- Algoritmo *Merge Sort*
- Algoritmo *Quick Sort*
- Ordenação crescente e decrescente
- Ordenação de estruturas complexas
- Ordenação indireta de *arrays*

## 20.1 INTRODUÇÃO

Neste capítulo, iremos estudar um assunto de primordial importância na ciência da computação: a ordenação.

Poderá parecer estranho que, no século XXI, se estudem algoritmos de ordenação, mas, na verdade, a ordenação é um processo de tal maneira importante que a própria palavra "computador" existe, em algumas línguas, com o sentido de "ordenador"[49], pois a tarefa de ordenar seja o que for sempre foi demasiado dispendiosa, e os computadores foram originalmente usados, de forma mais ou menos generalizada, para efetuar tarefas de ordenação, daí a associação ao nome "ordenador".

Embora o conceito de ordenação seja simples e se consiga implementar e resolver de forma quase natural, não deixa de ser surpreendente que existam tantos algoritmos de ordenação para resolver um problema que, aparentemente, é tão simples e que consiste na colocação de um conjunto de elementos por uma determinada ordem.

Ao longo deste capítulo, iremos estudar os principais algoritmos de ordenação. O resultado final da execução de cada um deles terá de ser o mesmo, isto é, todos deverão apresentar o conjunto de dados originais mas já ordenados. Contudo, a forma como cada um dos algoritmos o faz e a estratégia adotada por cada um deles podem levar a enormes ganhos ou custos em termos de tempos de execução, memória consumida, complexidade, etc.

[49] Por exemplo, *ordenador*, em espanhol, e *ordinateur*, em francês.

Com base no estudo dos vários algoritmos e na análise de resultados, será percetível que poderá ser muito vantajoso optar por usar um algoritmo em particular para resolver um problema específico. Perante a necessidade de ordenar um conjunto de valores, pode ser vantajoso ou não optar por uma determinada estratégia, em detrimento de outra.

Ao longo dos capítulos deste livro, podemos verificar que existe uma relação estreita entre o tipo de estrutura onde os dados são armazenados e a forma como os algoritmos que os manipulam são implementados.

Existe, assim, geralmente, uma estreita dependência entre a estrutura de dados e os algoritmos que a processam, por norma nos dois sentidos, isto é, a estrutura influencia o algoritmo e o algoritmo também pode influenciar a estrutura de dados.

Neste capítulo e no Capítulo 21, iremos, por momentos, esquecer-nos desta forte ligação. Vamos concentrar-nos no estudo de diferentes algoritmos que permitem a resolução de problemas específicos de programação: ordenação; e pesquisa. A componente de estruturas de dados terá uma menor importância, pois todos os casos a estudar irão basear-se num conjunto de elementos armazenados sequencialmente num *array*.

Tendo sempre como base a mesma estrutura de dados, iremos verificar que é a qualidade do algoritmo, ou seja, a qualidade da estratégia de ataque e resolução do problema, que irá acentuar a diferença entre melhores ou piores resultados nos vários casos a estudar.

Neste capítulo, iremos sempre partir de um *array* simples com dados para ordenar e obter como resultado um *array* – que, em princípio, será o mesmo – com os dados originais já ordenados.

## 20.2 PORQUÊ ORDENAR?

Ordenar corresponde ao processo de rearranjo de um conjunto de elementos colocando-os numa ordem específica.

O objetivo da ordenação de elementos é, normalmente, o de facilitar e tornar mais célere o processo posterior de pesquisa desses elementos.

Existem muitas razões pelas quais poderemos ter de ordenar dados numa aplicação, entre as quais as seguintes:

- Os dados necessários a uma aplicação podem ser obtidos do exterior e pode não existir a garantia de que estes dados estejam ordenados, tal como pode ser requerido pela aplicação;
- A maior parte das aplicações informáticas na fase de introdução de dados necessita de apresentar listas de valores para o utilizador escolher. É aconselhável que os dados presentes nessas listas estejam ordenados por algum critério lógico, de modo a facilitar a vida ao utilizador;
- Ainda que os dados de *input* de uma aplicação estejam ou não ordenados, o utilizador poderá querer que os diversos *outputs* gerados pela aplicação reflitam a aplicação de diferentes critérios e formas de ordenação dos dados de saída, sendo, por isso, necessário que a própria aplicação trate do processo de ordenação dos dados para a geração correta dos *outputs*;

```
104:
105: void hash_print(HASH_TABLE *h)
106: {
107:   printf("Imprimir a HASH TABLE (%d) registos\n",
108:          hash_count(h));
109:
110:   for (int i=0; i<HASH_SIZE; i++)
111:     item_print(i, h->arr[i], 1);
112: }
```

A aplicação de teste recebe dois argumentos na linha de comando:

- ***argv[1]*** – Nome do ficheiro de dados;
- ***argv[2]*** – Função de *probing* a usar: (***l***)*inear*|(***q***)*uadratic*|(***p***)*rime*].

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <ctype.h>
 4: #include "hash.h"
 5:
 6: #define LEN(x) (sizeof(x)/sizeof(x[0]))
 7:
```

**PROG2802.C**

Foram definidas três constantes que irão identificar a função de *probing* a usar:

```
 8: #define LINEAR    'l'
 9: #define QUADRATIC 'q'
10: #define PRIME     'p'
11:
12: int load_data(ITEM arr[], char *filename)
13: {
14:   FILE *fp = fopen(filename, "rt");
15:   if (fp==NULL)
16:   {
17:     fprintf(stderr, "Impossível abrir [%s]\n", filename);
18:     exit(2);
19:   }
20:
21:   int index=0;
22:   while (fscanf(fp, "%s %s", arr[index].name,
23:                              arr[index].phone)==2)
24:     index++;
25:   fclose(fp);
26:
27:   return index; // count
28: }
29:
30: void print_array(ITEM arr[], int count)
31: {
32:   printf("Imprimir o ARRAY (%d registos)\n", count);
33:   for (int i=0; i<count; i++)
34:     item_print(i, arr+i, 1);
35: }
36:
37: void load_hash_table(HASH_TABLE *h, ITEM arr[], int count)
38: {
39:   for (int i=0; i<count; i++)
40:     hash_add(h, arr+i);
41: }
42:
```

Tecnologias de Informação

# Linguagem C

## 25.ª Edição
## Atualizada e Aumentada

*Linguagem C*, da autoria de Luís Damas, é um clássico na literatura técnica em português sobre programação.

Amplamente utilizado como uma obra de referência e estudo, especialmente por estudantes e profissionais em áreas como Engenharia e Ciência da Computação, este é um livro essencial que cobre os fundamentos da linguagem C de forma clara e estruturada, abordando **desde os conceitos básicos até tópicos mais avançados**, incluindo inúmeros exercícios práticos, com acesso às resoluções, e exemplos que ajudam a consolidar a aprendizagem da matéria.

Esta nova edição mantém a apresentação detalhada da linguagem C, refletindo a evolução da linguagem e a forma como deve ser usada na atualidade, e conta com **12 novos capítulos com matérias avançadas** diretamente relacionadas com técnicas de programação ou com a implementação de estruturas de dados clássicas.

Baseado na longa experiência de programação e ensino do seu autor, este livro é destinado a estudantes de programação e a programadores que pretendem reforçar o seu conhecimento e as suas competências nas técnicas de desenvolvimento de *software* e na implementação de estruturas de dados usando a linguagem que serviu de base a quase todas as linguagens modernas e que poderão aplicar estes conhecimentos às linguagens que usam atualmente.

**Aprendeu com o *Linguagem C*?**
**Então atualize os seus métodos e desenvolva os seus conhecimentos com a 25.ª edição do livro de programação mais vendido em Portugal!**

**O que pode encontrar neste livro:**

- Apresentação e evolução da linguagem C
- Estruturas de dados (um livro completo 2 em 1!)
- 12 novos capítulos
- Mais de 370 programas completos
- Mais de 1000 funções implementadas
- Mais de 70 páginas de soluções dos exercícios propostas (com acesso via código QR)
- Mais de 35 000 linhas de código

**Porquê programar em C?**

ISBN 978-972-722-945-1
9 789727 229451
FCA
LIDEL
www.fca.pt