```
1
2
3   //                  *****   Resolução: E-fólio A  *****
4
5   /*
6   *   Segue-se um comentário sobre as resoluções e alternativas consideradas.
7   *   É essencial que reflitam sobre as formas alternativas que teriam de realizar
8   *   o e-fólio, de modo a poderem capitalizar todo o esforço despendido.
9   *   Vou dividir o comentário por alíneas, iniciando com um método aconselhado,
10  *   mas incluindo também métodos desaconselhados.
11  */
12
13  //-------------------------------------------------------------------------------
14  //                      Alínea A - verificar se uma casa é válida
15  //-------------------------------------------------------------------------------
16  /*
17  *   É solicitado um teste, só há dois possíveis resultados, pelo que o programa
18  *   resume-se a um condicional. A versão mais simples é fazer uso do facto da
19  *   numeração do tabuleiro ser sequencial, e utilizar simplesmente uma expressão:
20  */
21
22    if(strlen(casa) == 2 && casa[0]>='a' && casa[0]<='f' && casa[1]>='1' &&
      casa[1]<='5')
23      printf("Casa valida.");
24    else
25      printf("Casa invalida.");
26
27  /*
28  *   Temos que garantir que há apenas dois caracteres, e que a primeira letra seja
29  *   entre 'a' e 'f' e a segunda letra seja um dígito entre '1' e '5'.
30  *
31  *   Esperava que mais estudantes conseguissem esta solução, dado que o programa 4-3
32  *   demonstra que os caracteres são números sequenciais, pelo que pode-se fazer
33  *   comparações de desigualdade.
34  *
35  *   Houve também vários estudantes a trocar 'a' pelo inteiro correspondente ao
36  *   caracter 'a' (97) e o mesmo para outros caracteres. Não encontro
37  *   justificação/motivação para esta opção, o compilador converte o caracter
38  *   para número, a utilização do número apenas força mais atenção para quem lê o
    código.
39  *   Outra situação desagradável ocorrida neste método, vários estudantes
    esqueceram-se
40  *   da condicionante casa[0]>='a', embora esse erro não leve a perder caso de teste
    nenhum.
41  *   É essencial que façam o código de acordo com a especificação e não apenas para
42  *   passar nos casos de teste disponíveis.
43  *   Este código foi também feito com vários condicionais seguidos, o que força a
    duplicação
44  *   de um dos outputs, mas é igualmente bom. Os casos não aconselhados são os
    seguintes:
45  *
46  *   Método desaconselhando 1:
47  */
48
49  char str[][3] = {
50    "A1","B1","C1","D1","E1","F1","A2","B2","C2","D2","E2","F2","A3",
51
      "B3","C3","D3","E3","F3","A4","B4","C4","D4","E4","F4","A5","B5","C5","D5","E5","F5"
      , ""
52    };
53    ...
54    for(i=0; str[i][0]!=0; i++)
55      if(strcmp(casa, str[i]) == 0) {
56        printf("Casa valida.");
57        return 0;
58      }
59    ...
60  /*
61  *   Este método teria custos de atualizar o vetor no caso das especificações mudarem,
62  *   é lento dado que tem de verificar cada possibilidade, e cresce de tamanho sempre
63  *   que o tamanho do tabuleiro aumentar. Por outro lado, a ideia de ter todos os
```

```
    inputs,
64  *   é contrária a ter um programa genérico que funciona para qualquer situação,
65  *   portanto incontáveis casos. Nunca deve ser a primeira ideia que vos passa pela
    cabeça,
66  *   o método de enumerar todas as possibilidades.
67  *
68  *   Método desaconselhando 2:
69  */
70
71  char *str = "a5 b5 c5 d5 e5 f5 a4 b4 c4 d4 e4 f4 a3 b3 c3 d3 e3 f3 a2 b2 c2 d2 e2 f2
    a1 b1 c1 d1 e1 f1";
72  ...
73  if(strstr(str,casa)!=NULL) {
74    printf("Casa valida.");
75    return 0;
76  }
77  /*
78  *   O método 2 é outra forma do mesmo método 1, neste caso o input permitido é
79  *   colocado todo numa string, para utilizar o strstr em vez do ciclo for.
80  *   No entanto permite falsos positivos, por exemplo " b" seria aceite, para além
81  *   de manter os problemas da versão anterior.
82  *
83  *   Método desaconselhado 3:
84  */
85  ...
86    if((casa[0]=='a' || casa[0]=='b' || casa[0]=='c' || casa[0]=='d' || casa[0]=='e'
      || casa[0]=='f') &&
87      (casa[1]=='1' || casa[1]=='2' || casa[1]=='3' || casa[1]=='4' || casa[1]=='5' ||
      casa[1]=='6'))
88      printf("Casa valida.");
89    else
90      printf("Casa invalida.");
91  /*
92  *   Este método teria também os mesmos custos que o método 1,mas ainda assim
93  *   separa o que é diferente, permitindo um menor custo de manutenção.
94  *   Ao utilizar este método, o estudante revela que não sabe que os caracteres
95  *   são sequenciais.
96  */
97
98
99  //--------------------------------------------------------------------------------
100 //                      Alínea B - mostrar a casa num tabuleiro
101 //--------------------------------------------------------------------------------
102 /*
103 *   Esta alínea ainda era possível realizar sem abstrações funcionais, nem estrutura
104 *    de dados. Houve resoluções pouco genéricas, mas julgo que a mostra da alínea A
105 *   é suficiente. Aqui vamos realçar a vantagem que teriam de construir duas
    abstrações
106 *   funcionais óbvias, já que são referidas na especificação:
107 */
108
109 int CasaValida(char *casa);
110 void MostrarTabuleiro(char *casa);
111
112 /*
113 *   A primeira é uma função, e irá responder à questão da casa ser válida (alínea A).
114 *   A segunda é um procedimento, e irá executar uma função, mostrar o tabuleiro.
115 *   Alguns estudantes realizaram boas abstrações, mas outros ficaram bastante longe,
116 *   não tendo colocado um verbo num procedimento, nem o nome da grandeza que é
117 *   calculada na função. Assim torna-se o código muito mais complicado de ler,
118 *   já que não se sabe o que esperar.
119 *
120 *   Uma possível realização seria desta forma (sem estrutura de dados):
121 */
122
123 void MostrarTabuleiro(char *casa)
124 {
125     char linha, coluna;
126     for (linha = '5'; linha >= '1'; linha--) {
127         printf("\n%c:", linha);
128         for (coluna = 'a'; coluna <= 'f'; coluna++) {
```

```
            if (coluna == casa[0] && linha == casa[1])
                printf("X");
            else
                printf(".");
        }
    }
    printf("\n  ");
    for (coluna = 'a'; coluna <= 'f'; coluna++)
        printf("%c", coluna);
}

/*
*   Houve quem já tivesse feito a estrutura de dados
*   para reutilizar sem alterações na alínea seguinte.
*/

//--------------------------------------------------------------------------------
//                  Alínea C - mostrar um tabuleiro com peças colocadas
//--------------------------------------------------------------------------------

/*  Esta alínea requeria que fosse definida uma estrutura de dados, para guardar
*   que peça está em cada posição, ou se se quiser, a posição de cada peça. Estas
*   duas frases dão origem a dois tipos de estrutura de dados, ambas naturais e úteis:
*
*   Estrutura aconselhada 1:
*/

char tabuleiro[6][5]; // cada posição do tabuleiro tem a letra da peça respetiva
Estrutura aconselhada 2:

char reis[2][3]; // branco é o 0, o preto é o 1
char peoes[2][6][3]; // quando não há peões, é zero

/*
*   Houve quem tivesse optado pela estrutura 2 mas colocado todas as peças num só
vetor,
*   reis e peões e ambos os jogadores, portanto num total de 14 peças.
*   Acaba por ser equivalente a ter uma dimensão para cada cor, e dividido os reis
*   e os peões, como indicado aqui. Prefiro esta separação já que resulta no código
*   expressões mais simples.
*
*   Naturalmente que todas as funções teriam de estar dependentes
*   da estrutura de dados. Uma possibilidade para a função main:
*/

int main() {
    char tabuleiro[6][5]; // cada posição do tabuleiro tem a letra da peça respetiva
    if (!LerPosicao(tabuleiro))
        printf("Posicao invalida.");
    else
        MostrarTabuleiro(tabuleiro);
}
Evidentemente ser a estrutura for outra, a função main praticamente não se altera:
int main() {
    char reis[2][3]; // branco é o 0, o preto é o 1
    char peoes[2][6][3]; // quando não há peões, é zero
    if (!LerPosicao(reis, peoes))
        printf("Posicao invalida.");
    else
        MostrarTabuleiro(reis, peoes);
}

/*
*   Nesta função main, apenas é preciso atualizar a função MostrarTabuleiro
*   para mostrar as peças corretas, e implementar a função LerPosicao,
*   que carrega os dados de input para a estrutura de dados.
*   Por exemplo para a estrutura de dados 2:
*/

int LerPosicao(char reis[2][3], char peoes[2][6][3])
{
```

```c
    int i, j;
    char casa[80];
    // ler reis
    for (i = 0; i < 2; i++) {
        scanf("%s", casa);
        if (CasaValida(casa))
            strcpy(reis[i], casa);
        else
            return 0;
    }
    // ler peões
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 6; j++) {
            scanf("%s", casa);
            if (CasaValida(casa))
                strcpy(peoes[i][j], casa);
            else
                strcpy(peoes[i][j], "");
        }
    }
    return 1;
}

/*
*   Reutiliza-se a função da alínea A. Reparem também na reutilização da entrada de
*   dados, através do scanf. Houve quem tivesse lido a string completa com o gets,
*   e processado as tokens. É um método igualmente válido, utilizando o strtok,
*   dado no exercício somanumeros.c.
*   Não vale a pena forçar que exista apenas um espaço entre tokens.
*
*   Relativo a esta alínea, apenas mais um detalhe.
*   Quem tenha optado pela estrutura de dados 2, teria de ter forma
*   de saber que peça está em cada casa. Portanto, há uma abstração
*   claramente interessante a implementar, caso contrário mesmo a
*   função para mostrar o tabuleiro ficaria muito grande.
*/

char Letra(char linha, char coluna, char reis[2][3], char peoes[2][6][3])
{
    char *letraReis = "Rr", *letraPeoes="Pp";
    int i, j;
    // verificar se está lá um rei
    for (i = 0; i < 2; i++)
        if (reis[i][0] == coluna && reis[i][1] == linha)
            return letraReis[i];
    // verificar se está lá um peão
    for (i = 0; i < 2; i++)
        for (j = 0; j < 6; j++)
            if (peoes[i][j][0] == coluna && peoes[i][j][1] == linha)
                return letraPeoes[i];
    return '.';
}

/*
*   Este nome talvez não seja o  melhor, mas como devolve a letra que
*   está em cada posição, deixei-lhe esse nome. Assim, não há qualquer
*   problema em utilizar a estrutura 2 em vez da estrutura 1, já que sempre
*   que for necessário saber que peça está numa posição, utilizamos esta
*   abstração funcional. Claro que esta opção trás custos em termos de eficiência.
*/


//-----------------------------------------------------------------------------
//                            Alínea D - validar um jogo
//-----------------------------------------------------------------------------
/*
*   Nesta alínea após as primeiras 14 posições, segue-se um jogo.
*   É portanto necessário saber se as jogadas são válidas. Nesta alínea, na altura
*   em que as abstrações funcionais são mais precisas, quando a complexidade aumenta,
*   foi aqui que muitos estudantes deixaram as abstrações de lado e desenvolveram
*   funções muito grandes. Voltaram às origens, desperdiçando quando mais precisavam,
```

```
*   da principal ferramenta da programação, a abstração funcional. Estamos a falar de
*   estudantes com praticamente ou mesmo toda a funcionalidade correta, portanto com
*   elevada capacidade de programação. Um alerta para pensar sempre se não há uma
*   via mais simples, antes de entrar por uma via mais complexa.
*
*   Vamos primeiro ver a função main da estrutura de dados 1:
*/

int main() {
    char tabuleiro[6][5]; // estrutura com o tabuleiro, um caracter por cada casa
    char posicaoReis[2][3];
    int jogador = 0, jogadas = 0;
    char *jogadores[] = { "brancas", "pretas" };
    if (!LerPosicao(tabuleiro))
        printf("Posicao invalida.");
    else {
        while (LerJogada(tabuleiro, jogador)) {
            jogador = 1 - jogador;
            jogadas++;
            CasaRei(tabuleiro, posicaoReis);
            if (posicaoReis[0][0] == 0 || posicaoReis[1][0] == 0) // rei tomado,
            parar o jogo
                break;
        }
        jogador = 1 - jogador; // trocar o jogador, o atual fez um lance inválido
        MostrarTabuleiro(tabuleiro);
        printf("\nGanham as %s. Partida com %d jogadas validas.",
        jogadores[jogador], jogadas);
    }
}
/*
*   Após ler a posição tal como na alínea C, aqui precisamos de continuar a ler
*   jogadas, e no caso da jogada ser inválida parar. Esta é a função mais complexa.
*   No caso de ser uma jogada válida, simplesmente continuamos trocando o jogador e
*   incrementando as jogadas. Mas temos de verificar se ambos os reis existem.
*   Não vamos fazer esse trabalho aqui, utilizamos sim uma função para ir buscar a
*   posição dos reis, e depois valida-se. Tal como na estrutura de dados 2 pode-se
*   utilizar a função Letra para retornar a informação de acordo com a estrutura de
dados 1,
*   aqui pode-se fazer também o contrário, já que agora queremos saber a posição dos
*   reis (ou se não estão presentes). Daí as funções para obter a posição das
*   peças com base no tabuleiro:
*/

// localiza o rei e coloca em posicao
void CasaRei(char tabuleiro[6][5], char posicao[2][3]) {
    char *letraReis = "Rr";
    int i, j, jogador;
    for (jogador = 0; jogador < 2; jogador++) {
        posicao[jogador][0] = 0;
        for (i = 0; i<6; i++)
            for (j = 0; j<5; j++)
                if (tabuleiro[i][j] == letraReis[jogador]) {
                    posicao[jogador][0] = 'a' + i;
                    posicao[jogador][1] = '1' + j;
                    posicao[jogador][2] = 0;
                }
    }
}
// localiza as posições dos peões e coloca em posicoes
void CasaPeoes(char tabuleiro[6][5], char posicoes[2][6][3]) {
    char *letraPeoes = "Pp";
    int i, j, contador, jogador;
    for (jogador = 0; jogador < 2; jogador++) {
        contador = 0;
        for (i = 0; i<6; i++)
            for (j = 0; j<5; j++)
                if (tabuleiro[i][j] == letraPeoes[jogador]) {
                    posicoes[jogador][contador][0] = 'a' + i;
                    posicoes[jogador][contador][1] = '1' + j;
                    posicoes[jogador][contador++][2] = 0;
```

```
338                 }
339         // colocar os peões que faltam em casa nenhuma
340         while (contador < 6)
341             posicoes[jogador][contador++][0] = 0;
342     }
343 }
344 /*
345 *   Assim ambas as as estruturas de dados podem saber que peça está em
346 *   que posição, tal como podem saber em que posição está cada peça.
347 *   A estrutura de dados serve para auxiliar os algoritmos, não complicar.
348 *
349 *   Independentemente da estrutura de dados, algo é certo. Precisamos de
350 *   saber se um dado movimento é válido, para aceitá-lo ou não. Novamente
351 *   uma abstração de dados, que mantenha as regras, é essencial aqui.
352 */
353
354 int MovimentoValido(char *origem, char *destino, int jogador, char peca) {
355     if (!CasaValida(origem) || !CasaValida(destino))
356         return 0;
357     if (peca == 'R') {
358         return (abs(origem[0] - destino[0]) <= 1 && abs(origem[1] - destino[1]) <= 1
            &&
359             strcmp(origem, destino) != 0);
360     }
361     else if (peca == 'P') { // movimento de peão sem ser a tomar
362         return (destino[0] == origem[0] && destino[1] - origem[1] == 1 - 2 * jogador);
363     }
364     else if (peca == 'T') { // movimento de peão a tomar uma peça
365         return (abs(destino[0] - origem[0]) == 1 && destino[1] - origem[1] == 1 - 2
            * jogador);
366     }
367     return 0;
368 }
369
370 /*
371 *   Esta função verifica se a casa de origem/destino é válida, e se for um movimento
    de rei,
372 *   pode mover uma só casa, se for um movimento de peão, há dois movimentos, a
    tomada de
373 *   peça ou movimento normal sem tomar a peça. Portanto é fornecido um argumento para
374 *   cada uma das 3 situações.
375 *
376 *   Com esta abstração, a função para ler uma jogada é simples, aqui fica a versão
377 *   para a estrutura de dados 2 (a outra é idêntica):
378 */
379
380 int LerJogada(char reis[2][3], char peoes[2][6][3], int jogador)
381 {
382     char origem[80], destino[80], letraOrigem, letraDestino, *pecasJogador[] = {
        "RP", "rp" },
383         *letraReis = "Rr", *letraPeoes = "Pp";
384     // ler casa de origem
385     scanf("%s", origem);
386     if (CasaValida(origem)) {
387         letraOrigem = Letra(origem[1], origem[0], reis, peoes);
388         // o jogador apenas pode jogar de casas que tenham as suas peças
389         if (strchr(pecasJogador[jogador], letraOrigem) == NULL)
390             return 0;
391         // ler casa de destino
392         scanf("%s", destino);
393         if (CasaValida(destino)) {
394             letraDestino = Letra(destino[1], destino[0], reis, peoes);
395             // o jogador apenas pode jogar para casas vazias ou que tenham peças do
                adversário,
396             // e não para casas com as suas peças
397             if (strchr(pecasJogador[jogador], letraDestino) != NULL)
398                 return 0;
399             // caso seja o rei, pode mover uma casa para todas as direções
400             if (letraOrigem == letraReis[jogador]) {
401                 if (!MovimentoValido(origem, destino, jogador, 'R'))
402                     return 0;
```

```
403             }
404             // caso seja o peão, apenas pode mover para a frente e comer em diagonal
405             if (letraOrigem == letraPeoes[jogador]) {
406                 // se não toma nada
407                 if (letraDestino=='.') {
408                     if (!MovimentoValido(origem, destino, jogador, 'P'))
409                         return 0; // movimento inválido
410                 }
411                 else { // vai tomar uma peça
412                     if (!MovimentoValido(origem, destino, jogador, 'T'))
413                         return 0; // movimento inválido
414                 }
415             }
416             EfetuarMovimento(origem, destino, reis, peoes);
417             return 1;
418         }
419         return 0; // casa de destino inválida
420     }
421     if (strcmp(origem, "JA") == 0) // jogador artificial
422         return EfetuarJogada(reis, peoes, jogador);
423 
424     return 0; // casa de origem inválida
425 }
426 
427 /*
428 *   Notar a função EfetuarMovimento, após a jogada ser válida,
429 *   tem de se executar o movimento na estrutura de dados.
430 *   Outra função é EfetuarJogada, que entra no jogador automático.
431 */
432 
433 //----------------------------------------------------------------------------------
434 
435 /*
436 *   Alínea D - implementar uma estratégia para um jogador automático
437 *
438 *   Nesta alínea, claramente pelas regras definidas, precisamos de saber onde está
439 *   cada peça, pelo que a estrutura de dados 2 é preferivel.
440 *   Mostra-se aqui a função para a estrutura de dados 1:
441 */
442 
443 // jogador artificial
444 int EfetuarJogada(char tabuleiro[6][5], int jogador)
445 {
446     static char origens[40][3], destinos[40][3];
447     int i, jogadas = 0, melhor;
448     char posicaoReis[2][3];
449     char posicaoPeoes[2][6][3];
450     CasaRei(tabuleiro, posicaoReis);
451     CasaPeoes(tabuleiro, posicaoPeoes);
452     TomarRei(posicaoReis, posicaoPeoes, jogador, &jogadas, origens, destinos);
453     TomarPeoes(posicaoReis, posicaoPeoes, jogador, &jogadas, origens, destinos);
454     MoverPeoes(posicaoReis, posicaoPeoes, jogador, &jogadas, origens, destinos);
455     AvancarRei(posicaoReis, posicaoPeoes, jogador, &jogadas, origens, destinos);
456     MoverRei(posicaoReis, posicaoPeoes, jogador, &jogadas, origens, destinos);
457     if (jogadas > 0) {
458         // selecionar jogada mais para cima e mais para a esquerda (casa de destino),
459         // se igualdade, utilizar a mesma regra para a casa de origem para desempate
460         melhor = 0;
461         for (i = 1; i < jogadas; i++) {
462             if (jogador == 0 && destinos[i][1] > destinos[melhor][1] ||
463                 jogador == 1 && destinos[i][1] < destinos[melhor][1])
464                 melhor = i;
465             else if (destinos[i][1] == destinos[melhor][1]) {
466                 if (destinos[i][0] < destinos[melhor][0])
467                     melhor = i;
468                 else if (destinos[i][0] == destinos[melhor][0]) {
469                     if (jogador == 0 && origens[i][1] > origens[melhor][1] ||
470                         jogador == 1 && origens[i][1] < origens[melhor][1])
471                         melhor = i;
472                     else if (origens[i][1] == origens[melhor][1])
473                         if (origens[i][0] < origens[melhor][0])
```

```
                                melhor = i;
                    }
                }
            }
            EfetuarMovimento(origens[melhor], destinos[melhor], tabuleiro);
            return 1;
        }
        return 0;
    }

    /*
    *   Existem duas variáveis origens e destinos, com espaço para 40 movimentos.
    *   Estas variáveis são para conter as várias possíveis jogadas em cada nível.
    *   No entanto, quem analise por ordem, não precisa desta variáveis, já que o
    *   critério de escolha entre várias jogadas do mesmo nível é simples.
    *   No entanto se esse critério mudar, teriam sempre que ter uma estrutura de dados
    *   onde guardam as várias jogadas possíveis. Nesta função, após localizar a posição
    *   das peças, chamam-se as funções para cada tipo de jogada. Evidentemente que cada
    *   nível dá lugar a uma abstração funcional natural, já que as estratégias são
    *   distintas. No final, ficou o código que localiza a jogada com o critério de
    *   desempate mais alto, de entre as válidas.
    *
    *   A mensagem já vai longa, as funções específicas para cada tipo de jogada
    *   não são complicadas de implementar, mas naturalmente que quem procure implementar
    *   tudo numa só função, torna-se um desafio muito complicado.
    *
    *   Cumprimentos,
    *   José Coelho
    */
```