

Euro 2020, em 2021



UEFA
EURO2020

Introdução

Temos este ano, daqui a poucos meses, o Euro 2020. É verdade, por causa da pandemia, o campeonato, que estava marcado para o verão passado, só irá ter lugar este ano, entre 11 de junho e 11 de julho de 2021¹.

Enquanto esperamos pelas emoções desses grandes jogos recordemos o problema premonitório usado em 2016, também alguns meses antes do campeonato. Nessa altura, deu sorte. Quem sabe, este ano também dará.

Nota prévia

Este problema foi inventado em 2016, antes do Euro 2016, de boa memória. Eu teria atualizado o enunciado, com valores de 2021, mas para isso teria de obter dados atualizados sobre as internacionalizações dos jogadores. Ora o sítio Web onde eu fui buscar os dados em 2016 está em baixo ou já não existe e não consegui encontrar um outro sítio equivalente. Por isso, adaptei aqui o enunciado de 2016, com os dados de então.

¹ Calha bem, porque temos a nossa terceira festa no dia 9 de junho e as aulas acabam no dia 11 de junho. 😊

Na verdade, os dados existem no sítio da Federação Portuguesa de Futebol, mas não estão numa forma fácil de “sacar”.

Este problema foi usado em 2016 e novamente em 2020, em fevereiro, portanto antes da pandemia, quando inocentemente nem sonhávamos o que estava para acontecer. 😭

As três primeiras tarefas são de 2016; as duas seguintes são de 2020. A última é nova.

Texto de 2016, revisto



Teremos de planear cuidadosamente, para não perder nenhum dos jogos mais importantes. Por outro lado, temos de nos preparar desde já para aturar os comentadores, que nos bombardearão permanentemente com as estatísticas mais diversas e mais inúteis. Estou já a antecipar o alarido que vai acontecer quando o Cristiano Ronaldo ultrapassar o Figo em número de internacionalizações: o Figo tem 127 e o Cristiano tem 123. (Nota de 2020: há muito que o CR ultrapassou o Figo!) Como ainda há dois jogos internacionais particulares antes do

Euro, é mesmo provável que em junho o Cristiano se torne o jogador português mais internacional de todos os tempos. (Nota de 2020: todos sabemos que isso já aconteceu.)

Na verdade, qualquer um de nós, equipado com um computador ligado à Internet, pode obter variadas estatísticas, muito mais interessantes que aquelas com que os comentadores nos brindam.

O exercício de hoje é desse género: queremos calcular o *ranking* dos clubes pela contribuição que deram até agora à seleção nacional. Essa contribuição é medida pelo número total de internacionalizações dos jogadores de cada clube. Por exemplo, de acordo com os dados que consegui obter, e que correspondem aos anos a partir de 1990, os jogadores Marco Ferreira e Pedro Espinha, são os únicos jogadores do Vitória de Setúbal que jogaram na seleção. O primeiro jogou 3 vezes e o segundo 6 vezes. Logo, a contribuição do Vitória de Setúbal é 9.

Qual será o clube com maior contribuição? Melhor ainda, como é o *ranking* dos clubes, por contribuição?

Fui buscar os dados à página da Federação Portuguesa de Futebol, [aqui](#)² Depois de algum pré-processamento, produzi um ficheiro de texto, o ficheiro dos jogadores, onde em cada linha há seis elementos de informação: o número de internacionalizações do jogador, o nome, a posição a que joga (guarda-redes, defesa, médio ou avançado), o número de golos que marcou pela seleção, o ano em que nasceu e o clube que representa. Eis um exemplo de um troço desse ficheiro:

```
51 Jorge_Andrade Defesa 3 1978 JUVENTUS
51 Fábio_Coentrão Defesa 5 1988 MONACO
51 Paulo_Sousa Medio 0 1970 SPORTING
50 Jorge_Costa Defesa 2 1971 PORTO
45 Sá_Pinto Avancado 10 1972 SPORTING
45 Ricardo_Quaresma Avancado 4 1983 BESIKTAS
45 Rui_Jorge Defesa 1 1973 SPORTING
44 Dimas Defesa 0 1969 BENFICA
42 Rui_Patricio Guarda_Redes 0 1988 SPORTING
```

² Esta é a página que já não existe.

Na verdade, a informação sobre o clube é algo imprecisa, uma vez que os jogadores mudam de clube durante a carreira, e, portanto, o clube registado pode não ser o mais relevante, para efeitos da seleção. Por exemplo, o Fábio Coentrão jogou sobretudo no Benfica e no Real Madrid, mas agora está no Mónaco. Assim, as suas internacionalizações contribuem para o *ranking* do Mónaco e não para o do Benfica ou para o do Real Madrid.

A nossa meta hoje é produzir o *ranking*. Isso será feito no problema C. Antes, no problema A, montaremos a infraestrutura básica, com os tipos, leituras e escritas. Depois, no problema B, testaremos algumas consultas à tabela de jogadores, contida num array de estruturas. Finalmente, produziremos o *ranking* pretendido.

Não contei o número de linhas do ficheiro de jogadores, mas certamente não são mais de 1000.

Ler e escrever

Defina um tipo `Player` para representar os dados de cada jogador e programe o respetivo construtor. Depois, programe uma função para ler um array de `Player` a partir de um ficheiro, cuja referência é dada em argumento, e outra função para escrever num ficheiro, cuja referência é dada em argumento, o conteúdo do array, uma linha para cada elemento, com cada valor colocado entre parêntesis retos. Por exemplo, o troço do *output* correspondente ao troço do ficheiro de jogadores indicado acima seria assim:

```
[51] [Jorge_Andrade] [Defesa] [3] [1978] [JUVENTUS]
[51] [Fábio_Coentrão] [Defesa] [5] [1988] [MONACO]
[51] [Paulo_Sousa] [Medio] [0] [1970] [SPORTING]
[50] [Jorge_Costa] [Defesa] [2] [1971] [PORTO]
[45] [Sá_Pinto] [Avancado] [10] [1972] [SPORTING]
[45] [Ricardo_Quaresma] [Avancado] [4] [1983] [BESIKTAS]
[45] [Rui_Jorge] [Defesa] [1] [1973] [SPORTING]
[44] [Dimas] [Defesa] [0] [1969] [BENFICA]
[42] [Rui_Patricio] [Guarda_Redes] [0] [1988] [SPORTING]
[40] [João_Pereira] [Defesa] [0] [1984] [SPORTING]
```

Use a seguinte função de teste:

```
void test_players_read_write(const char *filename)
{
    FILE *f = fopen(filename, "r");
    assert(f != NULL);
    Player players[max_players];
    int n_players = players_read(f, players);
    players_write(stdout, players, n_players);
}
```

A variável `max_players` é uma constante global, assim definida:

```
const int max_players = 10000;
```

Na função `main`, chame a função de teste dando na linha de comando a letra ‘A’ como primeiro argumento e o nome do ficheiro de jogadores como segundo argumento. O *output* sai na consola.

Submeta no problema **A**.

Consultar por clube

Agora que já conseguimos ler do ficheiro de jogadores para o array, escrevamos uma função de teste para consultar o array. A função de teste lerá o ficheiro como no caso anterior e depois entra num ciclo iterativo, até ao fim dos dados. Em cada passo, a função aceitará o nome de um clube e escreverá em resposta uma sequência de linhas, cada uma com o nome de um jogador desse clube seguido do número de jogos (separados por um espaço), para cada um dos jogadores do clube, pela ordem porque vêm no array lido. Se não houver nenhum jogador do clube indicado, a função deve escrever apenas “`(void)`” (sem as aspas).

Na função `main`, chame esta função de teste dando na linha de comando a letra ‘B’ como primeiro argumento e o nome do ficheiro de jogadores como segundo argumento. O *output* sai na consola.

Submeta no problema **B**.

Ranking de clubes

Está na hora de escrever o programa para fazer o *ranking* dos clubes por contribuição para a seleção, de acordo com o significado de “contribuição”, explicado

acima. Em cada linha virá o nome do clube seguido do número de internacionalizações contribuídas, por ordem decrescente de contribuições, assim, em esquema:

```
OLHANENSE 134
FARENSE 82
PORTIMONENSE 19
LOULETANO 2
```

Em caso de empate, os nomes dos clubes devem vir por ordem alfabética.

Na função `main`, chame a função de teste dando na linha de comando a letra ‘C’ como primeiro argumento e o nome do ficheiro de jogadores como segundo argumento. O *output* sai na consola.

Submeta no problema **C**.

Indicações gerais

Resolva o problema à base de arrays. Quer dizer: por exemplo, na tarefa B, crie primeiro o array dos índices dos jogadores desse clube (será um array de `int`), e depois use uma função que, dado o array de todos os jogadores e um array de índices, escreva os jogadores cujos índices estão no array, pela ordem por que esses índices surgem no array.

Na tarefa C, crie primeiro um array só com clube e número de internacionalizações (com um `typedef` adequado para os elementos deste array), ordene, contabilize as internacionalizações de cada clube e ordene por número de internacionalizações, desempatando pelo nome do clube.

Para as ordenações, use sempre o *insertionsort*.

Ordenação por posição.

Queremos uma listagem de jogadores por posição: primeiro os guarda-redes, depois os defesas, depois os médios e depois os avançados. Desempata-se por número de golos marcados e pelo nome do jogador. A listagem deve vir no formato da tarefa A. Use o *insertionsort*, parametrizando a função de comparação.

Submeta no problema D.

Indicações

Terá de programar uma função de comparação de posição, tal que guarda-redes vem antes de defesa, que vem antes de médio, que vem antes de avançado. A ordem alfabética dos nomes das posições não dá, e mesmo que desse seria uma má escolha, pois ficaria agarrada aos nomes das posições usados na língua portuguesa. Sugiro uma função que atribua a cada posição um número (1 para guarda-redes, 2 para médio, etc.) e depois comparar esses números.

A função de comparação para a ordenação desejada usará esta outra que compara apenas por posição, e acrescenta-lhe os critérios de desempate.

Ordenação de arrays de apontadores para estruturas

Na ordenação com o *insertionsort*, trocamos elementos dentro do array. Ora neste caso, podemos reparar que cada elemento é volumoso. De facto, quantos bytes ocupa um valor de tipo `Player`? São três inteiros e três apontadores: $3 * 4 + 3 * 8 = 36$ bytes. Portanto, o tamanho será, *pelo menos*, 36 bytes. Na verdade, poderá ser um pouco mais, dependendo das regras de *preenchimento* (em inglês *padding*) usadas pelo compilador. Confira com uma função de teste que apenas faz `printf("%d\n", (int)sizeof(Player));` No meu computador deu 40.

Note que os caracteres do nome do jogador, da posição e do nome do clube estão em memória dinâmica e não são trocados. O que é trocado são apontadores para essas cadeias de caracteres, os quais residem na estrutura.

Para evitar o *overhead* de estar a trocar estruturas volumosas repetidamente, durante a ordenação, a técnica é usar arrays de apontadores para estruturas, em vez de arrays de estruturas. Assim, as estruturas ficarão quietinhas na memória dinâmica e quem troca são os apontadores.

Ou seja: em vez de trocar 40 bytes em cada *exchange*, trocaremos apenas 8 bytes!

O array de apontadores será declarado assim, na função de teste:

```
Player *players[MAX_PLAYERS];
```

Precisamos de um construtor que construa um objeto de tipo `Player` diretamente em memória dinâmica, devolvendo um apontador. Será assim, em esquema:

```
Player *player_ref(int caps, // other args...)
{
    Player *result = (Player *) malloc(sizeof(Player));
    result->caps = caps;
    // ...
    return result;
}
```

Repare que agora `result` é um apontador, de tipo `Player *`. No outro construtor, em que `result` é de tipo `Player`, escrevemos `result.caps = ...;`; agora, que `result` é um apontador para `Player` escrevemos `result->caps = ...;`. É uma regra geral: se `p` é um apontador para uma estrutura e `a` é um membro dessa estrutura, usamos `p->a`, para aceder a esse membro. Na verdade, poderíamos ter usado `(*p).a`, que vai dar ao mesmo, mas não dá jeito. Note que o operador ponto tem precedência em relação ao operador asterisco, pelo que escrever `*p.a`, seria equivalente a `*(p.a)` e não a `(*p).a`, que é o que queremos. Na prática, quando temos um apontador para estrutura usamos o operador seta (`->`) e não se fala mais nisso.

Precisamos de programar agora funções para arrays de apontadores para `Player`, análogas às que já temos. Será um bocado fastidioso, mas é a vida e isto é C. 😊

É claro que, na próxima vez, já mais avisados, programaremos logo tudo com arrays de apontadores e vez de arrays de objetos e despachar-nos-emos mais depressa.

Submeta no problema E.

Éder

Nas minhas investigações sobre os jogadores da seleção, descobri que no sítio da Federação Portuguesa de Futebol podemos realmente obter a informação desejada, para cada jogador: nome completo, clube da última convocatória, número de internacionalizações, número de golos, etc. Só que é uma página gerada au-

automaticamente, jogador a jogador, e não dá para sacar informação toda de uma vez, para um ficheiro csv, como eu queria.

Mais à frente, na cadeira de *Desenvolvimento de Aplicações para a Web*, aprenderemos a vasculhar *sites* automaticamente, mas isso ultrapassa os nossos conhecimentos neste momento.

Para já, o que conseguimos fazer é obter o código-fonte da página de cada jogador, uma a uma, e depois, extrair de lá os dados que nos fazem falta. Como as páginas têm a mesma estrutura para todos os jogadores (penso eu...), o que convém é um programa que leia o ficheiro código-fonte da página e automaticamente recolha os dados.

Para obter o código fonte de uma página, dado o URL, usamos o comando `curl` na linha de comando, assim, em esquema:

```
$ curl https://www.ualg.pt
```

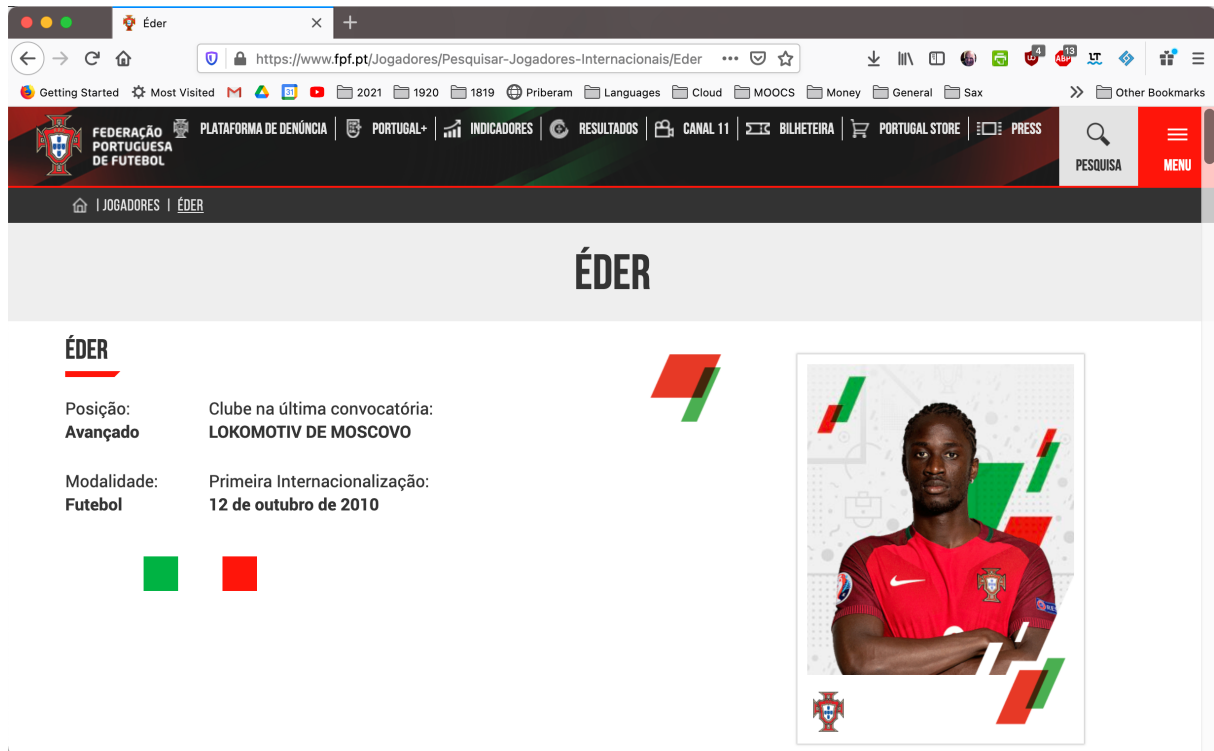
Em resposta, obteremos na consola a descrição codificada na página de entrada no sítio da Universidade do Algarve. Se quisermos estudar o código, o melhor é redirigir o output para um ficheiro, assim, por exemplo:

```
curl https://www.ualg.pt > out.txt
```

Nota de prudência: como medida de precaução é boa ideia redirigir o output na consola sempre para um ficheiro com o mesmo nome. Eu uso `out.txt`. Se o resultado estiver bom, e quisermos guardar o ficheiro, então a seguir mudamos o nome do ficheiro, de `out.txt` para o nome que for apropriado. Se, pelo contrário, decidirmos usar logo o nome apropriado de cada vez, corremos o risco de, inadvertidamente, escolher por engano um nome já usado para outro ficheiro na mesma diretoria. Se isso acontecer, o novo ficheiro de output destrói o ficheiro anterior e talvez não tivesse sido essa a nossa intenção...

Para obter o ficheiro para o Éder, por exemplo, começamos por aceder à página <https://www.fpf.pt/Jogadores>. Seleccionamos a “Futebol masculino, seleção A”. Como, os jogadores vêm ordenados por número de internacionalizações, o Éder não aparece na primeira página. Ainda assim, clicando no João Moutinho, va-

mos para a uma página cujo URL é <https://www.fpf.pt/Jogadores/Pesquisar-Jogadores-Internacionais/Joao-Moutinho>. Podemos arriscar que mudando de `Joao_Moutinho` para `Eder`, cairemos na página no Éder:



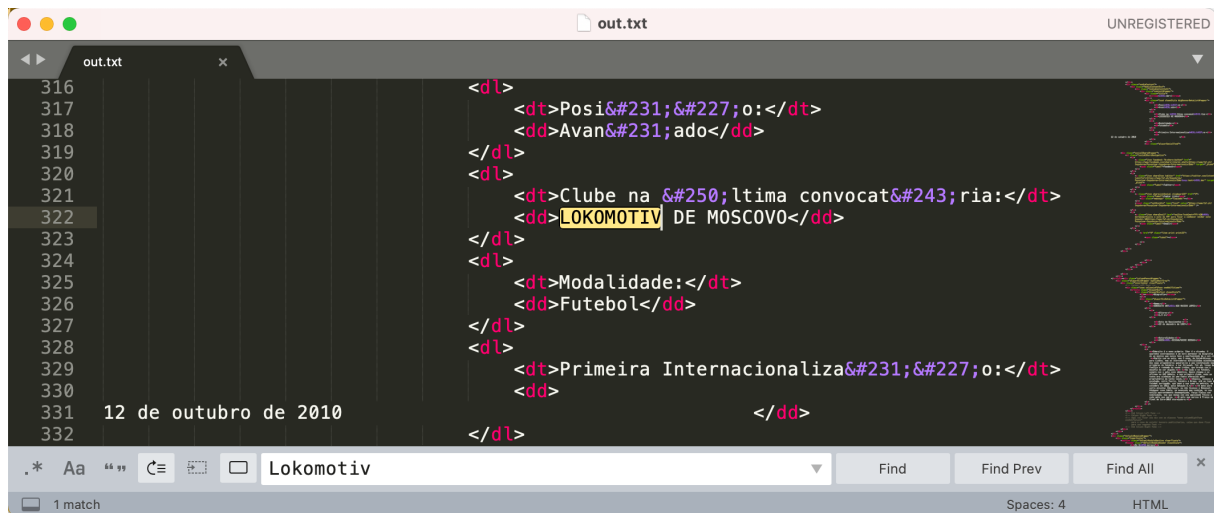
Portanto, para obter o código-fonte da página do Éder, basta dar o seguinte comando:

```
$ curl https://www.fpf.pt/Jogadores/Pesquisar-Jogadores-Internacionais/Eder > out.txt
```

O ficheiro obtido é enorme: cerca de 360 megabytes:

```
$ ls -al out.txt
-rw-r--r--@ 1 pedro  staff  361541 Mar 11 11:13 out.txt
```

Para conferir que conseguimos obter o que queríamos, abrimos o ficheiro no nosso editor de texto preferido e procuramos, por exemplo, “Lokomotiv”:



```
316 <dt>Posi&#231;&#227;o:</dt>
317 <dd>Avan&#231;ado</dd>
318 </dl>
319 <dl>
320 <dt>Clube na &#250;ltima convocat&#243;ria:</dt>
321 <dd>LOKOMOTIV DE MOSCOVO</dd>
322 </dl>
323 <dl>
324 <dt>Modalidade:</dt>
325 <dd>Futebol</dd>
326 </dl>
327 <dl>
328 <dt>Primeira Internacionaliza&#231;&#227;o:</dt>
329 <dd>
330 12 de outubro de 2010
331 </dd>
332 </dl>
```

Bingo, na linha 322!

Portanto, temos o que precisamos. Os restantes dados biográficos do Éder não estarão longe.

Admitimos que as todas as páginas de jogadores seguem o mesmo esquema. Portanto, inspirados pela página do Éder e de mais dois ou três jogadores, podemos imaginar um esquema para extrair automaticamente a informação que quisermos. Mas note que a técnica usada para ir buscar a página do Éder nem sempre resulta. Por exemplo, não dá para o Figo nem para o Eusébio. Por exemplo, o URL da página do Eusébio é <https://www.fpf.pt/Jogadores/Jogador/playerId/152729>, e não teríamos como adivinhar que o `playerId` do Eusébio é 152729. No entanto, *googlando* “player id 152729”, apanhamos logo a página do Eusébio na federação. Aliás, *googlando* mais simplesmente “fpf Eusébio”, também. Também, no caso do Éder, podíamos ter *googlado* “fpf Éder”.

As páginas dos jogadores são todas análogas, ao que percebi, ainda que para jogadores mais antigos falte alguma informação.

Tarefa

Escrever um programa que aceite por redireção do *standard input* um ficheiro com o código-fonte da página de um jogador da seleção, obtida no sítio da federação e escreva no *standard output* uma linha CSV com os valores obtidos relativos aos seguintes elementos de informação, por esta ordem:

1. Nome artístico
2. Nome completo
3. Posição
4. Clube na última convocatória
5. Data da primeira internacionalização
6. Data de nascimento
7. Naturalidade
8. Internacionalizações
9. Minutos jogados
10. Golos marcados
11. Cartões amarelos
12. Cartões vermelhos

Os sete primeiros campos são cadeias de caracteres. Os cinco últimos são números inteiros e referem-se a jogos da seleção A. (Alguns jogadores têm participação noutras seleções, por exemplo a seleção olímpica ou a seleção sub-19, mas isso não nos interessa aqui.) Os dados que não existirem na página ficarão em branco na linha CSV.

No caso do Éder, o programa deve escrever a seguinte linha CSV:

```
Éder,EDERZITO ANTÓNIO MACEDO LOPES,Avançado,LOKOMOTIV DE
MOSCÓVO,12 de outubro de 2010,GUINÉ-BISSAU/GUINÉ BISSAU,22 de
dezembro de 1987,35,1363,5,0,0
```

Submeta no problema F.

Anexo técnico

Os ficheiros HTML usam apenas os caracteres do bloco *Basic Latin* do Unicode. Estes são os caracteres “do costume”, que correspondem ao código ASCII de 7 bits, com valores numéricos até 127, inclusive.

Os restantes caracteres do Unicode são registados nos ficheiros HTML da forma que o seguinte troço ilustra:

```
Hist&#243;rico de Internacionaliza&#231;&#245;es
```

Percebemos que `Histórico` representa `Histórico` e `Internacionalizações` representa `Internacionalizações`. Na cadeia `Histórico` a

letra **ó** e representada internamente por dois bytes, usando a codificação UTF-8, e analogamente para as letras **ç** e **õ** em [Internacionalizações](#).

Quer dizer, ao ler o ficheiro HTML vamos ter de mudar da representação HTML para a codificação UTF-8.

No Unicode, cada caractere tem um número único, chamado *codepoint*. É esse número que aparece entre `&#` e `;`. Por exemplo, o *codepoint* do **ó** é 243. Portanto, o problema central é transformar automaticamente um *codepoint* na sequência de bytes correspondente na codificação UTF-8.

O processo é bastante detalhado, mas a nossa biblioteca tem uma função para realizar a operação:

```
//_Build the UTF-8 string corresponding to Unicode codepoint
_x
char *str_utf8(char *r, int x)
{
    // See table of powers of 2 at https://en.wikipedia.org/
    wiki/Power_of_two
    // See also https://en.wikipedia.org/wiki/Delete_character
    int n = 0;
    if (x <= 0)
        ; // these are not valid codepoints.
    else if (x < 127)
        r[n++] = (char)x;
    else if (x == 127)
        ; // this is the DEL character; it does not have a visual
    representation.
    else if (x < 128+64)
        ; // these are not valid codepoints.
    else if (x < 2048) // 2048 = 2^11
    {
        r[n++] = (char)(128+64+x/64);
        r[n++] = (char)(128+x%64);
    }
    else if (x < 65536) // 65536 = 2^16
    {
        r[n++] = (char)(128+64+32 + x/4096); // 4096 = 2^12
        r[n++] = (char)(128+(x%4096)/64);
        r[n++] = (char)(128+x%64);
    }
    else if (x < 2097152) // 2097152 = 2^21
    {
        r[n++] = (char)(128+64+32+16 + x/262144); // 262144 = 2^18
```

```

        r[n++] = (char)(128+(x%262144)/4096);
        r[n++] = (char)(128+(x%4096)/64);
        r[n++] = (char)(128+x%64);
    }
    else
        ; // these are not valid codepoints.
    r[n] = '\0';
    return r;
}

void unit_test_str_utf8(void)
{
    // See https://unicode-table.com
    char s[8];
    assert(str_equal(str_utf8(s, 65), "A"));
    assert(str_equal(str_utf8(s, 90), "Z"));
    assert(str_equal(str_utf8(s, 97), "a"));
    assert(str_equal(str_utf8(s, 122), "z"));

    assert(str_equal(str_utf8(s, 192), "À"));
    assert(str_equal(str_utf8(s, 201), "É"));
    assert(str_equal(str_utf8(s, 223), "ß"));
    assert(str_equal(str_utf8(s, 215), "×"));
    assert(str_equal(str_utf8(s, 247), "÷"));
    assert(str_equal(str_utf8(s, 213), "Ö"));

    assert(str_equal(str_utf8(s, 951), "η"));
    assert(str_equal(str_utf8(s, 8364), "€"));

    assert(str_equal(str_utf8(s, 128169), "🐛"));
}

```

A operação inversa, que calcula o codepoint de um dado caractere, não será necessária no nosso problema, mas é interessante:

```

// Compute the codepoint of the first character of `s`.
// Note: `s` uses UTF-8; the first character may occupy up to
// 4 bytes.
int str_codepoint(const char* s)
{
    // see https://en.wikipedia.org/wiki/UTF-8
    assert(s);
    int result;
    int x = (unsigned char)s[0];
    if (x < 128)
        result = x;
}

```

```

else if (x < 128+64)
    assert(0); // must not happen at the first byte.
else if (x < 128+64+32)
    result = x % 32 * 64 + (unsigned char)s[1] % 64;
else if (x < 128+64+32+16)
    result = (x % 16 * 64 + (unsigned char)s[1] % 64) * 64 +
              (unsigned char)s[2] % 64;
else
    result = ((x % 8 * 64 + (unsigned char)s[1] % 64) * 64 +
              (unsigned char)s[2] % 64) * 64 +
              (unsigned char)s[3] % 64;

return result;
}

void unit_test_str_codepoint(void)
{
    // See https://unicode-table.com
    assert(str_codepoint("A") == 65);
    assert(str_codepoint("Z") == 90);
    assert(str_codepoint("a") == 97);
    assert(str_codepoint("z") == 122);
    assert(str_codepoint("À") == 192);
    assert(str_codepoint("ß") == 223);

    assert(str_codepoint("x") == 215);
    assert(str_codepoint("÷") == 247);

    assert(str_codepoint("Õ") == 213);
    assert(str_codepoint("η") == 951);
    assert(str_codepoint("€") == 8364);
    assert(str_codepoint("💩") == 128169);

    assert(str_codepoint("À") == 192);
    assert(str_codepoint("Å") == 197);
    assert(str_codepoint("È") == 200);
    assert(str_codepoint("Ë") == 203);
    assert(str_codepoint("Ì") == 204);
    assert(str_codepoint("Ĩ") == 207);
    assert(str_codepoint("Ò") == 210);
    assert(str_codepoint("Ø") == 216);
    assert(str_codepoint("Ù") == 217);
    assert(str_codepoint("Ü") == 220);

    assert(str_codepoint("Ç") == 199);
    assert(str_codepoint("Ñ") == 209);
}

```


Com isto, programa-se a função que nos faz falta, para converter uma cadeia lida de um ficheiro HTML para a sua representação UTF-8:

```
// Convert html string `s` to UTF-8
char *str_from_html(char *r, const char *s)
{
    int i = 0;
    int n = 0;
    while (s[i])
    {
        if (s[i] == '&' && s[i+1] == '#')
        {
            char *p;
            int x = (int)strtol(s+i+2, &p, 10);
            assert(*p == ';');
            i = (int)(p-s)+1;
            str_utf8(r+n, x);
            n += (int)strlen(r+n);
        }
        else
            r[n++] = s[i++];
    }
    r[n] = '\0';
    return r;
}

void unit_test_str_from_html(void)
{
    // See https://unicode-table.com
    char s[32];
    assert(str_equal(str_from_html(s, "A"), "A"));
    assert(str_equal(str_from_html(s, "&#201;"), "É"));
    assert(str_equal(str_from_html(s, "GUIN&#201;"), "GUINÉ"));
    assert(str_equal(str_from_html(s, "&#951;&#8364;&#128169;"),
    "ñ€👉"),
    assert(str_equal(str_from_html(s, "Hist&#243;rico"),
    "Histórico"));
    assert(str_equal(str_from_html(s,
    "Internacionaliza&#231;&#245;es"),
    "Internacionalizações"));
    assert(str_equal(str_from_html(s,
    "Clube na &#250;ltima convocat&#243;ria"),
    "Clube na última convocatória"));
}
```

A função `strtol` calcula o número inteiro cuja representação é dada no primeiro argumento. O terceiro argumento representa a base usada na representação. No caso vertente é 10, pois usamos números decimais. O segundo argumento o endereço onde a conversão terminou, porque a cadeia chegou ao fim ou porque apareceu um caractere que não é um algarismo.

Nota pessoal: programei estas funções recentemente e ainda foram pouco usadas. Se houver azar, por favor avisem.