

Arrays de cadeias de caracteres

Pedro Guerreiro

Universidade do Algarve

1 de março de 2021

As cadeias de caracteres são arrays de caracteres. Tipicamente, cada cadeia representa uma palavra, ou um nome, ou uma frase, ou uma linha de texto. Assim, terá relativamente poucos caracteres, talvez algumas dezenas, raramente alguns milhares. Mas nada impede que surjam cadeias com milhões de caracteres.

Consultando as estrofes de *Os Lusíadas*

Por exemplo, *Os Lusíadas* têm 8816 versos e cada verso terá, em média, cerca de 36 caracteres. Logo, uma única cadeia de caracteres com $8816 \times 36 = 317376$ caracteres chegaria para conter todo o texto. Alguns destes caracteres serão caracteres de mudança de linha, bem entendido, no fim de cada verso. Os caracteres de mudança de linha são os que nos programas em C representamos por `'\n'`.

Para concretizar, escrevamos um programa que repetidamente aceita um número da consola e responde, mostrando na consola para cada número a estrofe d'Os Lusíadas com esse número de ordem.

Sabemos que *Os Lusíadas* têm 1102 estrofes de oito versos, distribuídas em dez cantos. Dispomos à partida de um ficheiro com o texto completo. Este ficheiro tem 8816 linhas, uma para cada verso.

Convém guardar o texto todo em memória, no início das operações para facilitar o processamento. Alternativamente, poderíamos ler o ficheiro de novo, para cada número aceite, mas isso é pouco prático.

Para guardar o texto em memória, podemos usar uma única cadeia, como dissemos há pouco, mas é mais conveniente usar um array de cadeias, uma cadeia para cada verso. Tendo *Os Lusíadas* 8816 versos, precisamos de um array de 8816 cadeias. E qual deve ser a capacidade das cadeias? Admitindo que a capacidade

tem de ser a mesma para todas as cadeias, temos de dimensionar por excesso. Não tendo ainda analisado a obra em pormenor, não sabemos com exatidão, mas podemos estimar que, tendo cada verso dez sílabas, quase de certeza não haverá versos com mais de 50 caracteres. Arredondamos para uma potência de 2 e declaramos o array de cadeias de caracteres assim:

```
const int max_verses = 8816;
const int max_stanzas = max_verses / 8;
const int max_verse_length = 64;

char poem[max_verses][max_verse_length];
```

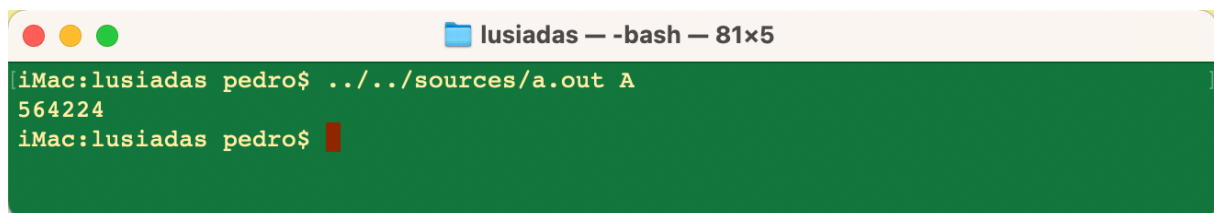
As constantes serão declaradas globalmente e o array será declarado na função de teste.

Desta forma, a variável `poem` é um array com capacidade para 8816 cadeias, cada uma das quais tem capacidade para 64 caracteres (dos quais um será o terminador). Este array ocupa $8816 \times 64 = 564224$ bytes.

Confirmemos que é assim, com a seguinte função de teste:

```
void test_array_size(void)
{
    char poem[max_verses][max_verse_length];
    printf("%ld\n", sizeof poem);
}
```

Eis o resultado do teste:

A screenshot of a macOS terminal window titled "lusiadas — -bash — 81x5". The prompt is "iMac:lusiadas pedro\$". The user enters the command "../sources/a.out A". The output is "564224". The prompt then changes to "iMac:lusiadas pedro\$".

```
iMac:lusiadas pedro$ ../sources/a.out A
564224
iMac:lusiadas pedro$
```

Repare, no `printf`, o especificador de formato `%ld`. Tem de ser `%ld`, e não `%d` como de costume, porque, tecnicamente, o operador `sizeof` retorna um valor de tipo `long int`, e com `long int` deve usar-se `%ld`.

Repare também que `sizeof` é um operador unário e não uma função. Por isso, podemos escrever `sizeof poem`, em vez de `sizeof(poem)`. Esta segunda expres-

são também é válida, pois os parênteses são redundantes. Também seria válido escrever `sizeof((poem))` ou `sizeof((((poem))))`.

Escrevamos então uma função de teste para ler um ficheiro contendo o texto d’*Os Lusíadas*, carregando-o para um array, e despejando depois o array na consola, para controlo. Lemos linha a linha, usando a função `str_readline` da nossa biblioteca. Por hipótese, o ficheiro reside numa determinada diretoria e conhecemos o *pathname*, que registamos numa constante global:

```
const char *poem_filename =  
    "/Users/pedro/Dropbox/LP_2021/work/lusiadas/lusiadas.txt";
```

Eis a função de teste:

```
void test_read_lusiadas(void)  
{  
    FILE *f = fopen(poem_filename, "r");  
    assert(f);  
    char poem[max_verses][max_verse_length];  
    int n = 0;  
    char line[max_line_length];  
    while (str_readline(f, line) != EOF)  
    {  
        assert(str_len(line) < max_verse_length);  
        str_cpy(poem[n++], line);  
    }  
    assert(n == max_verses);  
    for (int i = 0; i < n; i++)  
        printf("%s\n", poem[i]);  
}
```

A constante global `max_line_length` terá sido declarada anteriormente, com um valor generoso:

```
const int max_line_length = 10000;
```

Note que se alguma linha lida tiver comprimento maior ou igual a `max_verse_length` a instrução `assert(str_len(line) < max_verse_length);` fará o programa terminar em erro.

As funções `str_len`, `str_readline` e `str_cpy` fazem parte da nossa biblioteca:

```

// Length of the string, measured in memory bytes used
int str_len(const char *s)
{
    int result = 0;
    while (s[result] != '\0')
        result++;
    return result;
}

// Copy `s` to `r`, return `r`.
// `r` should not overlap `s` to the right.
char* str_cpy(char *r, const char *s)
{
    int n = 0;
    for (int i = 0; s[i]; i++)
        r[n++] = s[i];
    r[n] = '\0';
    return r;
}

// Read a line from `f` to `s`.
// Does not control buffer overflow.
int str_readline(FILE *f, char *s)
{
    int result = EOF;
    char *p = fgets(s, INT_MAX, f);
    if (p != NULL)
    {
        result = str_len(s);
        if (result > 0 && s[result-1] == '\n')
            s[--result] = '\0';
    }
    return result;
}

```

Corramos a função `test_read_lusiadas`, com o seguinte comando:

```
$ ../../sources/a.out B
```

O programa escreverá os 8816 versos instantaneamente e apenas ficarão visíveis os últimos:

```
lusiadas — -bash — 80x24
De vós não conhecido nem sonhado?
Da boca dos pequenos sei, contudo,
Que o louvor sai às vezes acabado.
Nem me falta na vida honesto estudo,
Com longa experiência misturado,
Nem engenho, que aqui vereis presente,
Cousas que juntas se acham raramente.
Pera servir-vos, braço às armas feito,
Pera cantar-vos, mente às Musas dada;
Só me falece ser a vós aceito,
De quem virtude deve ser prezada.
Se me isto o Céu concede, e o vosso peito
Dina empresa tomar de ser cantada,
Como a pres[s]aga mente vaticina
Olhando a vossa inclinação divina,
Ou fazendo que, mais que a de Medusa,
A vista vossa tema o monte Atlante,
Ou rompendo nos campos de Ampelusa
Os muros de Marrocos e Trudante,
A minha já estimada e leda Musa
Fico que em todo o mundo de vós cante,
De sorte que Alexandre em vós se veja,
Sem à dita de Aquiles ter enveja.
iMac:lusiadas pedro$
```

Se quisermos controlar, para verificar que as 8816 linhas foram realmente escritas, o melhor é redirigir o *standard output* para um ficheiro, na linha de comando, assim, por exemplo:

```
$ ../../sources/a.out B > out.txt
```

Este programa resolve o problema de ler e escrever o texto d'*Os Lusíadas*, mas fá-lo de maneira sofrível. De facto, deveria haver uma função capaz de ler linha a linha ficheiros de texto quaisquer, guardando as linhas lidas num dado array de cadeias de caracteres. E, reciprocamente, deveria haver uma função capaz de escrever em ficheiro de texto quaisquer, linha a linha, as cadeias presentes num dado array de cadeias. Essas funções deveriam ter os seguintes protótipos:

```
int strings_read(FILE *f, const char **a);
void strings_fprint_basic(FILE *f, const char **a, int n);
```

Na verdade, estas funções existem na nossa biblioteca:

```
int strings_read(FILE *f, const char **a)
{
    int result = 0;
    char line[max_line_length];
```

```

while (str_readline(f, line) != EOF)
    a[result++] = str_dup(line);
return result;
}

void strings_fprint_basic(FILE *f, const char **a, int n)
{
    for (int i = 0; i < n; i++)
        fprintf(f, "%s\n", a[i]);
}

```

Desta forma, somos levados reprogramar a função `test_read_lusiadas` assim:

```

void test_read_lusiadas_does_not_compile(void)
{
    FILE *f = fopen(poem_filename, "r");
    assert(f);
    char poem[max_verses][max_verse_length];
    int n = strings_read(f, poem);
    assert(n == max_verses);
    strings_fprint_basic(stdout, poem, n);
}

```

A ideia é boa, mas não compila. Nas duas funções, `strings_read` e `strings_fprint_basic` o compilador queixa-se de que o argumento `poem` é de tipo `char [8812][64]` e que o argumento é de tipo `const char **`, sendo os dois tipos *incompatíveis*.

Esta é uma questão fundamental. Podemos tentar torneá-la de várias maneiras, mas o mais simples é declarar a variável `poem` assim:

```
const char *poem[max_verses];
```

As duas declarações, `char poem[max_verses][max_verse_length]` e `const char *poem[max_verses]`, não são equivalentes. A primeira corresponde a um array de 8816 cadeias de caracteres. Ocupa, como já vimos, 564224 bytes. A segunda indica uma array de 8816 *apontadores*. Como cada apontador ocupa 8 bytes, o array ocupará $8816 \times 8 = 70528$ bytes.

Os apontadores são variáveis que contêm o endereço de outras variáveis. Neste caso, tratando-se de apontadores de tipo `char *`, as variáveis cujo endereço está contido no apontador serão de tipo `char`. Com este arranjo, o que nos convém é

guardar em cada apontador do array o endereço na posição de memória do primeiro caractere do verso correspondente.

É precisamente assim que funciona a função `strings_read`. Cada cadeia lida é copiada para uma posição de memória atribuída *dinamicamente* pelo sistema operativo, por meio da função `malloc`, a qual é usada na função `str_dup`:

```
const char *str_dup(const char *s)
{
    char *result = (char *) malloc(strlen(s) + 1);
    strcpy(result, s);
    return result;
}
```

Vimos a função `string_read` mais acima. Recordemos o seu protótipo

```
int strings_read(FILE *f, const char **a);
```

Aqui declaramos que o parâmetro `a` é de tipo `const char **`, significando que o argumento dever ser um array de apontadores de tipo `const char *`. O atributo `const` neste contexto estabelece que o compilador não aceitará operações que modifiquem a memória usando apontadores provenientes do array `a`. Mas o valor de cada um desses apontadores pode mudar. De certa forma o que é “constante” é a memória apontada, não cada apontador.

Nota técnica: os nossos programas usam abundantemente `const char *`. Não confunda com `char const *` que é o tipo dos apontadores “constantes”, isto é, cujo valor não pode mudar, mas em que a memória apontada pode. E não confunda também com `const char * const`, que é o tipo dos apontadores constantes através do qual não se pode modificar a memória. Nos nossos programas, não temos uso para estes tipos `char const *` e `const char * const`. Nas funções com cadeias de caracteres usaremos `const char *` para representar parâmetros de entrada (que não vão ser modificados pela função) e `char *`, para parâmetros de saída, que sim, vão ser modificados pela função. Nas funções com arrays de cadeias de caracteres o tipo dos arrays será sempre `const char **`. Isto significa que as funções sobre arrays de caracteres não modificam as cadeias

em memória, mas podem modificar os apontadores, efetivamente substituindo cadeias por outras cadeias.

Por conseguinte, a anterior função que não compilava deve ser reprogramada assim:

```
void test_lusiadas_fixed(void)
{
    FILE *f = fopen(poem_filename, "r");
    assert(f);
    const char *poem[max_verses];
    int n = strings_read(f, poem);
    assert(n == max_verses);
    strings_fprint_basic(stdout, poem, n);
}
```

Podemos agora escrever o programa que escreve as estrofes, proposto de início.

```
void test_stanzas(void)
{
    FILE *f = fopen(poem_filename, "r");
    assert(f);
    const char *poem[max_verses];
    int n = strings_read(f, poem);
    assert(n == max_verses);
    int x;
    while (scanf("%d", &x) != EOF)
    {
        int y = int_constrain(x, 1, max_stanzas) - 1;
        strings_fprint_basic(stdout, poem + 8*y, 8);
    }
}
```

Esta função de teste recorre à função `int_constrain` para lidar com o caso que o utilizador indicar um número defetivo ou excessivo:

```
int int_constrain(int x, int min, int max)
{
    int result = x;
    if (x < min)
        result = min;
    else if (x > max)
        result = max;
    return result;
}
```



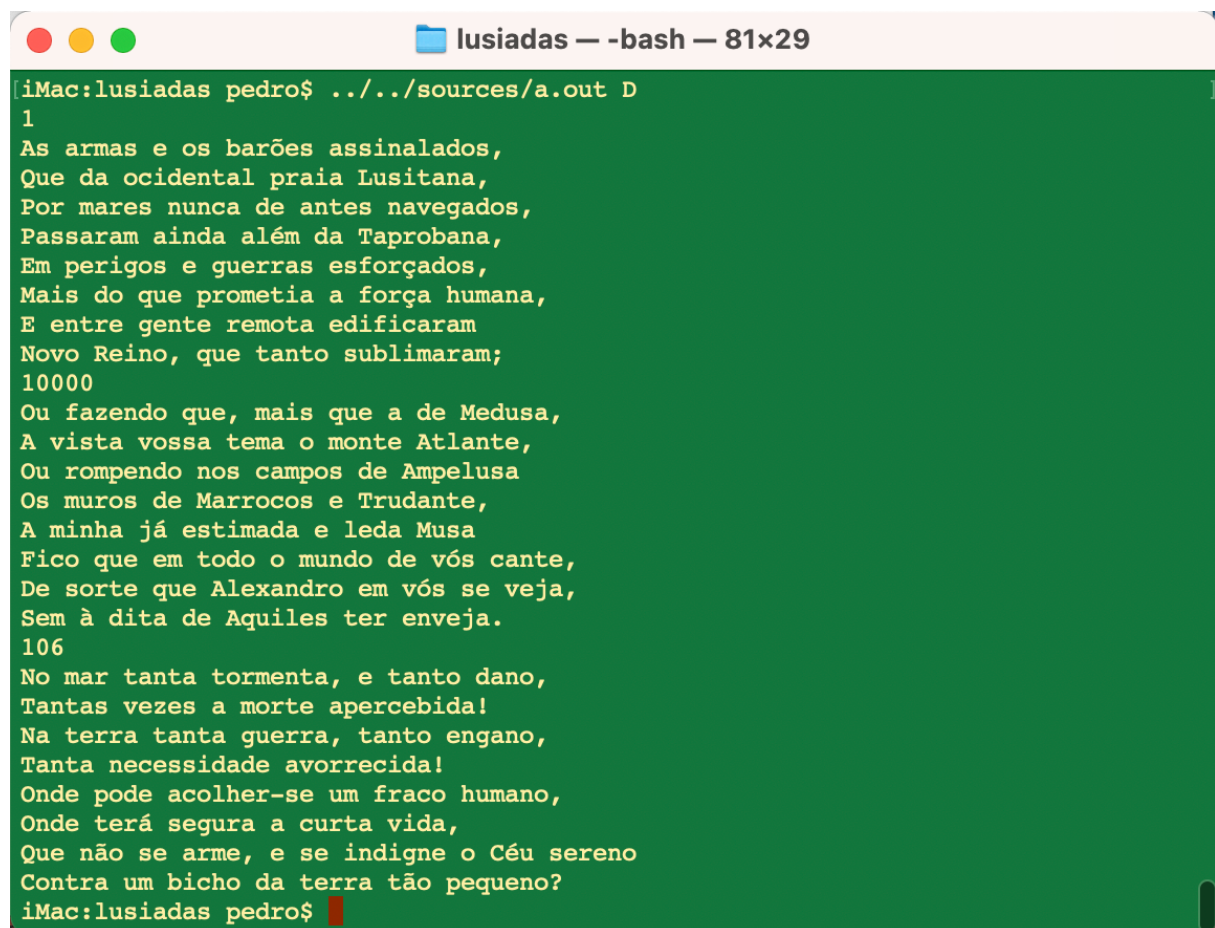
```

}

void unit_test_int_constrain(void)
{
    assert(int_constrain(48, 0, 100) == 48);
    assert(int_constrain(24, 0, 20) == 20);
    assert(int_constrain(32, 65, 90) == 65);
    assert(int_constrain(20, 32, 32) == 32);
    assert(int_constrain(1000, 100, 100) == 100);
    assert(int_constrain(10, 10, 20) == 10);
    assert(int_constrain(20, 10, 20) == 20);
    assert(int_constrain(-15, -20, -10) == -15);
    assert(int_constrain(-100, -20, -10) == -20);
    assert(int_constrain(-5, -20, -10) == -10);
}

```

Eis o resultado de uma sessão de teste com a função `test_stanzas`:



```

iMac:lusiadas pedro$ ../../sources/a.out D
1
As armas e os barões assinalados,
Que da ocidental praia Lusitana,
Por mares nunca de antes navegados,
Passaram ainda além da Taprobana,
Em perigos e guerras esforçados,
Mais do que prometia a força humana,
E entre gente remota edificaram
Novo Reino, que tanto sublimaram;
10000
Ou fazendo que, mais que a de Medusa,
A vista vossa tema o monte Atlante,
Ou rompendo nos campos de Ampelusa
Os muros de Marrocos e Trudante,
A minha já estimada e leda Musa
Fico que em todo o mundo de vós cante,
De sorte que Alexandro em vós se veja,
Sem à dita de Aquiles ter enveja.
106
No mar tanta tormenta, e tanto dano,
Tantas vezes a morte apercebida!
Na terra tanta guerra, tanto engano,
Tanta necessidade avorrecida!
Onde pode acolher-se um fraco humano,
Onde terá segura a curta vida,
Que não se arme, e se indigne o Céu sereno
Contra um bicho da terra tão pequeno?
iMac:lusiadas pedro$

```

Variações de leitura

Estudemos agora algumas variações à maneira de resolver o problema de ler o ficheiro com o texto d'Os Lusíadas, ou de um texto qualquer, que reside num ficheiro.

Na secção anterior, lemos o texto para uma array de cadeias. Será que não podemos ler o ficheiro para uma cadeia única?

As funções de leitura que de dispomos não servem, pois leem uma linha de cada vez. Isso, aliás, existe até uma limitação escondida, pois a leitura é feita para a cadeia declarada na função, com uma capacidade grande, mas fixa: `max_line_length = 10000`. Se houver uma linha com mais caracteres do que este valor, o programa funcionará mal e provavelmente estoirará.

É simples ler um ficheiro todo para uma cadeia única. Observe:

```
char *str_from_file(char *r, FILE *f)
{
    int n = 0;
    char x;
    while (fscanf(f, "%c", &x) != EOF)
        r[n++] = x;
    r[n] = '\0';
    return r;
}
```

Note o especificador de conversão `%c` usado na função `scanf`, indicando que a leitura deve ser feita byte a byte. Informalmente, dizemos que lemos caractere a caractere, mas isso não é exato, pois, como já sabemos certos caracteres ocupam mais do que um byte.

Eis uma função de teste, que lê para uma cadeia um ficheiro obtido por redireção do input:

```
void test_str_from_file(void)
{
    char s[1000];
    str_from_file(s, stdin);
    printf("%d\n", str_len(s));
    printf("<%s\n", s);
}
```

}

Para controlo, a função mostra o número de bytes lidos, que pode ser maior que o número de caracteres lidos (no caso de alguns caracteres ocuparem mais de um byte). Além disso, a cadeia é escrita entre < e >, para ser claro onde começa e termina.

Eis o registo de uma sessão de teste:



```
lusiadas — -bash — 81x24
[iMac:lusiadas pedro$ ../../sources/a.out E
aaa bbbbbbb ccc ddd
eeeeeee f gggg
hhhh iii jjjj
49
<aaa bbbbbbb ccc ddd
eeeeeee f gggg
hhhh iii jjjj
>
[iMac:lusiadas pedro$ ../../sources/a.out E
água vinho café
18
<água vinho café
>
[iMac:lusiadas pedro$ ../../sources/a.out E
🇪🇺
🇵🇹
🇧🇷
27
<🇪🇺
🇵🇹
🇧🇷
>
iMac:lusiadas pedro$
```

Observamos que a cadeia `água vinho café` é indicada como tendo 18 bytes. É assim porque a letra ‘á’ ocupa dois bytes, a letra ‘é’ também, e o caractere de mudança de linha foi lido para a cadeia.

As três linhas com bandeiras ocupam 27 bytes. Confere, pois cada bandeira ocupa 8 bytes e cada uma é seguida por um caractere de mudança de linha.

Podemos testar, lendo *Os Lusíadas*, desde que ajustemos a capacidade da cadeia:

```
void test_str_from_file(void)
{
    // char s[1000];
    char s[400000];
    str_from_file(s, stdin);
    printf("%d\n", str_len(s));
}
```

```
    printf("<%s>\n", s);
}
```

Os Lusíadas são ecoados para a consola, de uma vez, mas usando as 8816 linhas, pois há 8816 caracteres `\n` na cadeia.

Portanto, “vemos” as linhas na consola, mas elas não existem autonomamente no programa.

No entanto, podemos, a partir da cadeia que contém o texto todo, construir um array de apontadores para o início de cada linha. Se, ao fazer isso, substituírmos os caracteres `\n` pelo terminador `\0`, ficaremos com as linhas acessíveis individualmente. É isso que faz a função `str_lines`:

```
int str_lines(char *s, const char **a)
{
    int result = 0;
    int i = 0;
    while (s[i])
    {
        a[result++] = s+i;
        int z = str_count_while_not(s+i, '\n');
        i += z;
        if (s[i] != '\0')
            // if not at the end of string, replace '\n' by '\0'
            s[i++] = '\0';
    }
    return result;
}
```

Esta função usa a técnica do *count-while* para cadeias:

```
int str_count_while(const char *s, char x)
{
    int result = 0;
    while (s[result] != '\0' && s[result] == x)
        result++;
    return result;
}

int str_count_while_not(const char *s, char x)
{
    int result = 0;
    while (s[result] != '\0' && s[result] != x)
```

```

    result++;
    return result;
}

```

Note que na função `str_lines` o parâmetro `s` não é `const`. Se fosse `const`, não poderíamos modificar o argumento, na instrução `s[i++] = '\0';`.

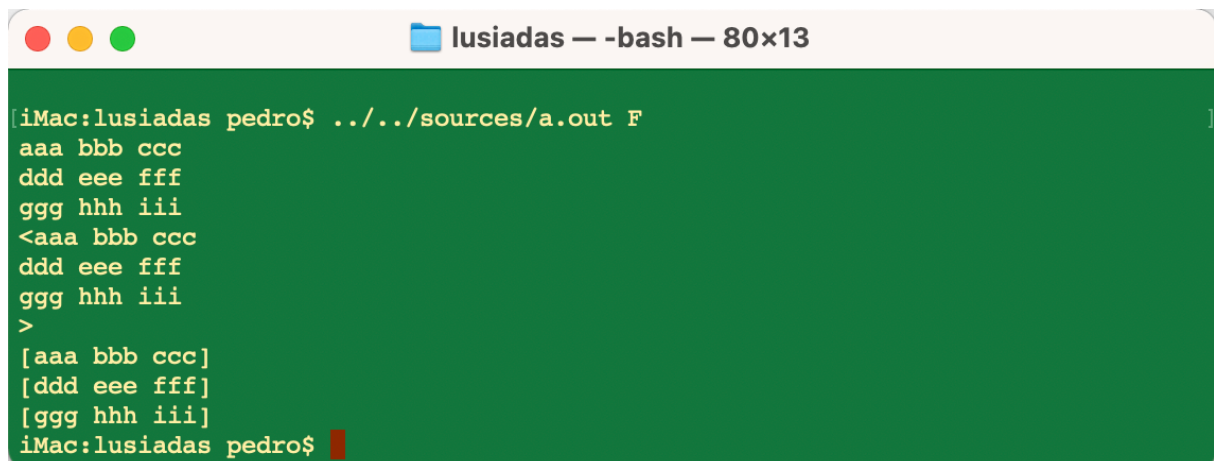
Eis uma função de teste:

```

void test_str_lines(void)
{
    char s[400000];
    str_from_file(s, stdin);
    printf("<%s>\n", s);
    const char *a[10000];
    int n = str_lines(s, a);
    strings_fprintf(stdout, a, n, "[%s]\n");
}

```

A função lê o input para a cadeia `s`, mostra a cadeia entre `<` e `>`, “separa” a cadeia `s` em linhas, guardando o endereço do início de cada linha no array `s` e, finalmente, mostras as linhas, cada uma entre `[` e `]`. Eis o registo de uma sessão de teste:



```

lusiadas - -bash - 80x13
[Mac:lusiadas pedro$ ../../sources/a.out F
aaa bbb ccc
ddd eee fff
ggg hhh iii
<aaa bbb ccc
ddd eee fff
ggg hhh iii
>
[aaa bbb ccc]
[ddd eee fff]
[ggg hhh iii]
iMac:lusiadas pedro$

```

A função `strings_fprintf` escreve num ficheiro um array de cadeias, usando para cada uma a cadeia de formato especificada:

```

void strings_fprintf(FILE *f, const char **s, int n, const
char *fmt)
{
    for (int i = 0; i < n; i++)
        fprintf(f, fmt, s[i]);
}

```

}

Análise das palavras

Um problema clássico de programação é contar o número de palavras num texto. Estudemo-lo.

A primeira questão é perceber o que é uma “palavra”, para efeitos da contagem. Abreviando, usamos a “definição” habitual em programação, neste contexto: uma palavra é uma sequência de caracteres que não inclui espaços e que está situada entre espaços. Em rigor, a primeira palavra de um texto pode não ter nenhum espaço antes e a última pode não ter nenhum espaço depois.

Por “espaços”, na frase anterior, entendemos os caracteres que não se “veem”, dos quais nos servimos para separar cada palavra da seguinte. São, designadamente, o caractere espaço, de código numérico 32, o caractere de mudança de linha `\n` e o caractere de tabulação, ou “tab”, de código numérico 9. Na prática, não temos de nos preocupar com os códigos e usamos a função de biblioteca `isspace`, que dá 1 se o argumento for um caractere que deve ser considerado “espaço” e 0, se não.

Nota técnica: para usar a função `isspace` e outras funções sobre caracteres individuais, é preciso fazer `#include <ctype.h>`.

Para contar as palavras, usaremos de novo a técnica do *count-while*:

```
int str_count_while_is_not_space(const char *s)
{
    int result = 0;
    while (s[result] != '\0' && !isspace(s[result]))
        result++;
    return result;
}

int str_count_words(const char *s)
{
    int result = 0;
    int i = 0;
    while (s[i])
        if (isspace(s[i]))
            i++;
}
```

```

        else
        {
            int z = str_count_while_is_not_space(s+i);
            result++;
            i += z;
        }
    return result;
}

void unit_test_count_words(void)
{
    assert(str_count_words("aaa bbb ccc") == 3);
    assert(str_count_words("") == 0);
    assert(str_count_words(" ") == 0);
    assert(str_count_words("aaa") == 1);
    assert(str_count_words("   aaa ") == 1);
5) assert(str_count_words("  a      aaa  a      aaaa  aa  ") ==
    assert(str_count_words(" a b c d e f g h i j ") == 10);
    assert(str_count_words("aaa\naaa\naaa\naaa") == 4);
6) assert(str_count_words("água café limão Évora São João") ==
}

```

Agora que já sabemos contar, com um pouco mais de esforço, construiremos o array das palavras. Na verdade, a técnica é semelhante à usada para partir uma cadeia em linhas: guardamos um apontador para o início de cada palavra e colocamos um terminador logo após o fim de cada palavra, onde antes havia um espaço. Observe:

```

int str_words(char *s, const char **a)
{
    int result = 0;
    int i = 0;
    while (s[i])
        if (isspace(s[i]))
            i++;
        else
        {
            a[result++] = s+i;
            int z = str_count_while_is_not_space(s+i);
            i += z;
            if (s[i] != '\0')
                s[i++] = '\0';
        }
}

```

```

    return result;
}

```

Eis uma função de teste, que lê de um ficheiro para uma cadeia, depois separa a cadeia em palavras e finalmente lista as palavras, cada uma entre { e }:

```

void test_str_words(void)
{
    char s[400000];
    str_from_file(s, stdin);
    const char *a[100000];
    int n = str_words(s, a);
    strings_printf(a, n, "%s\n");
}

```



```

iMac:lusiadas pedro$ ../../sources/a.out G
aaa bbb ccc
ddd eee fff
{aaa}
{bbb}
{ccc}
{ddd}
{eee}
{fff}
iMac:lusiadas pedro$

```

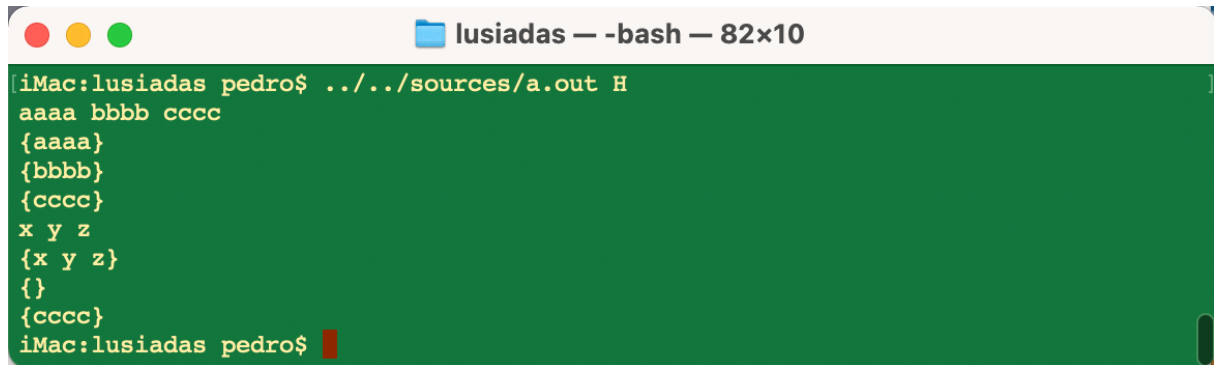
Apesar de funcionar corretamente neste exemplo, a ideia de usar apontadores para uma cadeia pré-existente é pouco recomendável. De facto, se essa cadeia mudar, depois de as palavras terem sido calculadas, o arrays das palavras deixa de fazer sentido. Observe a seguinte função de teste, que ilustra a questão:

```

void test_str_words_again(void)
{
    char s[1000];
    str_readline(stdin, s);
    const char *a[100];
    int n = str_words(s, a);
    strings_printf(a, n, "%s\n");
    str_readline(stdin, s);
    strings_printf(a, n, "%s\n");
}

```


A função lê uma linha para a variável `s`, separa as palavras e guarda os apontadores no array `a`, mostra o array `a`, lê outra linha para `s`, e mostra o array `a`. Vejamos uma experiência:



```
iMac:lusiadas pedro$ ../../sources/a.out H
aaaa bbbb cccc
{aaaa}
{bbbb}
{cccc}
x y z
{x y z}
{}
{cccc}
iMac:lusiadas pedro$
```

Constatamos que após a segunda leitura, as palavras no array estão “estradas”. Não admira, percebendo nós a implementação. No entanto, este comportamento não é desejável.

É preferível, bem entendido, guardar no array endereços não de posições no meio da cadeia que contem o texto lido, mas sim endereços de cópias dinâmicas de cada uma das palavras identificadas, assim:

```
// This is a better technique
int str_words(const char *s, const char **a)
{
    int result = 0;
    int i = 0;
    while (s[i])
        if (isspace(s[i]))
            i++;
        else
        {
            int z = str_count_while_not_func(s+i, isspace);
            a[result++] = str_ndup(s+i, z);
            i += z;
        }
    return result;
}
```

A função `str_ndup` é análoga à função `str_dup` mas só copia para memória dinâmica um dado número de caracteres (e não necessariamente a cadeia toda):

```

const char *str_ndup(const char *s, int n)
{
    char *result = (char *) malloc(n + 1);
    str_ncpy(result, s, n);
    result[n] = '\0';
    return result;
}

```

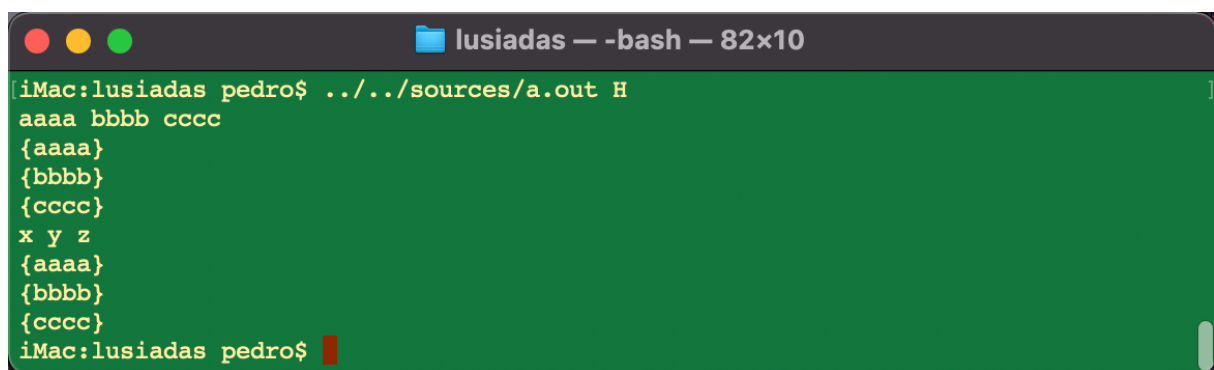
Por sua vez, a função `str_ncpy` é análoga à função `str_cpy`, mas só copia de uma cadeia para outra um dado número de caracteres (e não necessariamente a cadeia toda):

```

// Copy at most `x` bytes from `s` to `r`, return `r`.
char* str_ncpy(char *r, const char *s, int x)
{
    int n = 0;
    for (int i = 0; i < x && s[i]; i++)
        r[n++] = s[i];
    r[n] = '\0';
    return r;
}

```

Experimentemos de novo a função de teste `test_str_words_again`, após ter substituído a anterior versão da função `str_words` pela nova:



A terminal window titled "lusiadas — -bash — 82x10" showing the execution of a program. The prompt is "iMac:lusiadas pedro\$../sources/a.out H". The output is as follows:

```

aaaa bbbb cccc
{aaaa}
{bbbb}
{cccc}
x y z
{aaaa}
{bbbb}
{cccc}
iMac:lusiadas pedro$

```

Assim é melhor!

Para o que der e vier, é melhor substituir também a função `str_lines` por forma a usar memória dinâmica:

```

int str_lines(const char *s, const char **a)
{
    int result = 0;
    int i = 0;

```

```

while (s[i])
{
    int z = str_count_while_not(s+i, '\n');
    a[result++] = str_ndup(s+i, z);
    i += z;
    if (s[i]) // if not at the end of strings, skip the
        i++;
}
return result;
}

```

Note que agora o parâmetro `s` é `const`. Antes não era.


Resta coligir as palavras d’*Os Lusíadas* e contá-las:

```

void test_count_lusiadas(void)
{
    FILE *f = fopen(poem_filename, "r");
    assert(f);
    char text[400000];
    str_from_file(text, f);
    const char *words[100000];
    int n_words = str_words(text, words);
    printf("%d\n", n_words);
    strings_fprintf(stdout, words, n_words, "[%s]\n");
}

```

Correndo na linha de comando, só conseguimos observar as últimas palavras:



```

iMac:lusiadas pedro$ ./test_count_lusiadas
[De]
[sorte]
[que]
[Alexandro]
[em]
[vós]
[se]
[veja,]
[Sem]
[à]
[dita]
[de]
[Aquiles]
[ter]
[enveja.]
iMac:lusiadas pedro$

```

Constatamos que os sinais de pontuação aparecem colados às palavras propriamente ditas, o que era de esperar, mas podemos querer evitar. Para estudar o resultado com mais comodidade, o melhor é guardar num ficheiro:

```
iMac:lusiadas pedro$ ../../sources/a.out I > out.txt
```

A primeira linha do ficheiro `out.txt` revela que foram detetadas 55324 palavras.

E assim termina o nosso exercício.

Falta só indicar, para referência, a função `unit_tests` e a função `main`:

```
void unit_tests(void)
{
    unit_test_int_constrain();
    unit_test_str_equal();
    unit_test_str_len();
    unit_test_str_find();
    unit_test_str_find_last();
    unit_test_str_prefix();
    unit_test_count_words();
}

int main(int argc, const char **argv)
{
    unit_tests();
    int x = 'U';
    if (argc > 1)
        x = *argv[1];
    if (x == 'A')
        test_array_size();
    else if (x == 'B')
        test_read_lusiadas();
    else if (x == 'C')
        test_read_lusiadas_fixed();
    else if (x == 'D')
        test_stanzas();
    else if (x == 'E')
        test_str_from_file();
    else if (x == 'F')
        test_str_lines();
    else if (x == 'G')
        test_str_words();
    else if (x == 'H')
        test_str_words_again();
}
```

```
else if (x == 'I')
    test_count_lusiadas();

else if (x == 'U')
    printf("All unit tests PASSED\n");
else
    printf("%s: Invalid option.\n", argv[1]);
return 0;
}
```