UAlg FCT

UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Laboratório de Programação

Lição n.º 1

Arrays de cadeias de carateres

Técnicas com arrays de cadeias de carateres

- Leitura linha a linha.
- Leitura de arrays de cadeias.
- Leitura de cadeias dinâmicas.
- Escrita de arrays de cadeias.

Arrays de cadeias de carateres

- Normalmente, usaremos apenas arrays de cadeias dinâmicas, e, ocasionalmente, arrays de cadeias estáticas.
- As cadeias dinâmicas são criadas com malloc.
- Cada valor no array de cadeias dinâmicas contém o endereço de uma cadeia, a qual reside na memória dinâmica.
- O valor da cadeia na memória dinâmica terá sido copiado a partir de uma cadeia na pilha.
- Tecnicamente, um array de cadeias dinâmicas é um array de "apontadores", isto é, um array cujos valores são endereços.
- Nos usos mais comuns, as cadeias dinâmicas são referenciadas por variáveis que estão na pilha.

Note bem: dizemos "cadeias dinâmicas" porque elas residem na memória dinâmica (ou seja, no heap), não porque elas tenham algum tipo de "dinamismo".

Exemplo: Hello para muitos

- Um programa que aceita uma sequência de nomes, até ao fim dos dados, e depois diz "Hello" com cada um dos nomes lidos.

```
void test_hello_many(void)
{
  char *names[10];
  int n = 0;
  char word[16];
  while (scanf("%s", word) != EOF)
  {
    names[n] = (char *) malloc(strlen(word) + 1);
    strcpy(names[n++], word);
  }
  for (int i = 0; i < n; i++)
    hello(names[i]);
}
```

```
$ ./a.out
Cristiano
Rui
Ricardo
William
Rafael
Hello Cristiano
Hello Rui
Hello Ricardo
Hello William
Hello Rafael
$
```

strlen

- A função strlen dá o comprimento da cadeia passada em argumento:

```
void unit_test_strlen(void)
{
  char *s1 = "guatemala";
  assert(strlen(s1) == 9);
  assert(strlen(s1+4) == 5);
  assert(strlen(s1+9) == 0);
  char *s2 = "brasil";
  assert(strlen(s2) == 6);
  assert(strlen(s2+1) == 5);
}
```

Note que s1+4, por exemplo, é a subcadeia de s1 que começa em s[4].

- Podia programar-se assim:

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

Na verdade, o tipo do resultado da função de biblioteca strlen não é int, mas sim size_t, o que por vezes causa algumas surpresas.

Note bem: representamos por '\0' o caráter cujo valor numérico é 0 (o tal que é usado como terminador nas cadeias).

strcpy

- Para copiar os bytes que constituem uma cadeia para outra posição de memória, usamos a função strcpy.
- Podia programar-se assim:

```
char *str_cpy(char *r, const char *s)
{
  int i = 0;
  while (s[i] != 0)
  {
    r[i] = s[i];
    i++;
  }
  r[i] = '\0';
  return r;
}
```

Na verdade, a função de biblioteca strcpy retorna o valor do primeiro argumento, que representa o endereço para onde o segundo argumento terá sido copiado. Quase sempre ignoramos o valor de retorno.

Note que todos os carateres de s são copiados para posições sucessivas, a partir da primeira posição r, e depois, no fim, acrescenta-se o terminador.

Ler linha a linha vs. ler palavra a palavra

- Ao ler cadeias de carateres com scanf ("%s", ...), primeiro a função salta os carateres *brancos* que existam no input e depois recolhe para a cadeia passada em argumento os carateres não brancos, até surgir um caráter branco (que não é recolhido) ou até ao fim dos dados.
- Os carateres brancos são o espaço, o tab ('\t') e o caráter de mudança de linha ('\n').
- No final, o scanf acrescenta o terminador.
- Sendo assim, o scanf("%s", ...) é prático para ler o input palavra a palavra, mas não para ler linhas inteiras que tenham mais que uma palavra.

Note bem: o scanf não controla o *buffer overflow*.

Ler uma linha inteira

- Por razões técnicas deveras subtis, o C não tem uma função de biblioteca para ler uma linha inteira (ou o resto da linha corrente).
- À falta de uma função de biblioteca, usaremos as seguintes:

```
int str_readline(FILE *f, char *s)
{
  int result = EOF;
  char *p = fgets(s, INT_MAX, f);
  if (p != NULL)
  {
    result = (int) strlen(s);
    if (result > 0 && s[result-1] == '\n')
      s[--result] = '\0';
  }
  return result;
}

int str_getline(char *s)
{
  return str_readline(stdin, s);
}
```

Explicação: str_readline lê uma linha com fgets, para a cadeia s, a partir do ficheiro f, sem controlar *buffer overflow*. Se a linha lida terminar com mudança de linha (o que acontece sempre exceto porventura na última linha do ficheiro), o último caráter de s será '\n'. Então, a função elimina o '\n', substituindo-o pelo terminador '\0'. Em caso de fim de ficheiro, o fgets devolve NULL e a função str_readline devolve EOF (por analogia com scanf).

A função str_getline faz a leitura a partir da consola, representada por stdin.

Digressão: ++ e --

- Quanto vale x++? Vale x.
- Quanto vale ++x? Vale x+1.
- Em ambos os casos, depois da avaliação da expressão, a variável x fica a valer x+1.
- Quanto vale x--? Vale x.
- Quanto vale --x? Vale x-1.
- Em ambos os casos, depois da avaliação da expressão, a variável x fica a valer x-1.

Não confunda: uma coisa é o valor da expressão x++; outra coisa é o valor da variável x.

Evidência

- A seguinte função de teste unitário ilustra o significado dos operadores **++** e **--**:

```
void unit_test_plus_plus_minus_minus(void)
{
  int x = 5;
  assert(x++ == 5);
  assert(x == 6);
  int y = 9;
  assert(++y == 10);
  assert(y == 10);
  int z = 3;
  assert(z-- == 3);
  assert(z == 2);
  int w = 8;
  assert(--w == 7);
  assert(w == 7);
}
```

Ler linhas para a memória dinâmica

- Eis uma função de teste que lê linhas, da consola para a memória dinâmica, até ao fim dos dados e que depois despeja a memória para a consola, cada linha entre parêntesis retos.

```
#define MAX_LINES 10000
#define MAX_LINE_LENGTH 10000

void test_get_many_lines_basic(void)
{
  char *s[MAX_LINES];
  int n = 0;
  char line[MAX_LINE_LENGTH];
  while (str_getline(line) != EOF)
  {
    s[n] = (char *) malloc(strlen(line) + 1);
    strcpy(s[n++], line);
  }
  for (int i = 0; i < n; i++)
    printf("[%s]\n", s[i]);
}
```

```
$ ./a.out
viana do castelo
ponte de lima
porto
vila franca de xira
setubal
sao bras de alportel
[viana do castelo]
[ponte de lima]
[porto]
[vila franca de xira]
[setubal]
[sao bras de alportel]
$
```

Cada uma das operações assinadas com uma caixa merece ser autonomizada numa função.

Duplicar uma cadeia, str_dup

- Duplicar uma cadeia significa alocar espaço para uma cópia dessa cadeia na memória dinâmica e copiar para lá os carateres da cadeia original, devolvendo o endereço da cópia recém-criada:

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

Atenção: esta é uma operação fundamental. Usamo-la a toda a hora!

Nota: esta função não existe na biblioteca standard do C, mas existe em certas extensões, com o nome strdup.

Ler cadeias, linha a linha

- Normalmente, queremos ler de um ficheiro, linha a linha, para um array de cadeias dinâmicas:

```
int strings_read(FILE *f, char **a)
{
  int result = 0;
  char line[MAX_LINE_LENGTH];
  while (str_readline(f, line) != EOF)
    a[result++] = str_dup(line);
  return result;
}
```

- Para ler da consola, usamos a seguinte variante:

```
int strings_get(char **a)
{
  return strings_read(stdin, a);
}
```

Ler cadeias, palavra a palavra

- Para ler palavra a palavra, confiamos no scanf:

```
int strings_readwords(FILE *f, char **a)
{
  int result = 0;
  char word[MAX_LINE_LENGTH];
  while (fscanf(f, "%s", word) != EOF)
    a[result++] = str_dup(word);
  return result;
}
```

- Para ler da consola, usamos a seguinte variante:

```
int strings_getwords(char **a)
{
  return strings_readwords(stdin, a);
}
```

Escrever cadeias

- Em geral, ao escrever um array de cadeias num ficheiro, queremos ser capazes de especificar o formato de escrita:

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

- Para escrever na consola, usamos a seguinte variante:

```
void strings_printf(char **s, int n, const char *fmt)
{
  strings_fprintf(stdout, s, n, fmt);
}
```

Escrever cadeias, com separador

- Por vezes, queremos apenas indicar o separador:

```
void strings_fprint(FILE *f, char **s, int n, const char *separator)
{
  if (n > 0)
  {
    fprintf(f, "%s", s[0]);
    for (int i = 1; i < n; i++)  // i = 1
      fprintf(f, "%s%s", separator, s[i]);
  }
}
```

- Para escrever na consola, com separador, usamos a seguinte variante:

```
void strings_print(char **s, int n, const char *separator)
{
  strings_write(stdout, s, n, separator);
}
```

Escrever cadeias e mudar de linha

- Frequentemente, depois de escrever queremos mudar de linha automaticamente:

```
void strings_fprintln(FILE *f, char **s, int n, const char *separator)
{
  strings_write(f, s, n, separator);
  fprintf(f, "\n");
}

void strings_println(char **s, int n, const char *separator)
{
  strings_writeln(stdout, s, n, separator);
}

void strings_fprintfln(FILE *f, char **s, int n, const char *fmt)
{
  strings_fprintf(f, s, n, fmt);
  fprintf(f, "\n");
}

void strings_printfln(char **s, int n, const char *fmt)
{
  strings_fprintfln(stdout, s, n, fmt);
}
```

Função test_strings_get

- Eis uma função que testa simultaneamente algumas das funções que descrevemos, lendo de uma vez um ficheiro para uma array de cadeias, linha a linha, e despejando as linhas para a consola, entre chavetas:

```
void test_strings_get(void)
{
  char *s[MAX_WORDS];
  int n = strings_get(s);
  strings_printf(s, n, "{%s}\n");
}
```

```
$ ./a.out
a cidade e as serras
memorial do convento
o que diz molero
{a cidade e as serras}
{memorial do convento}
{o que diz molero}
$
```