Programação Estruturada

Aula 16 - Strings

Yuri Malheiros (yuri@ci.ufpb.br)

Strings Literais

- Um string literal é uma sequência de caracteres entre aspas duplas:
  - "Alô mundo"
- Nós já usamos strings literais no `printf` e no `scanf`

Strings Literais

- Se você precisar escrever um string literal muito grande que não caiba em uma linha, podemos usar `\` para continuá-lo em outra linha

```
printf("Este é um string literal muito grande \
que ocupa mais de uma linha");
```

- A continuação do string após a `\` precisa começar no início da linha

Strings Literais

- Para strings grandes você também pode usar a alternativa:

```
printf("Este é um string literal muito grande"
       "que ocupa mais de uma linha");
```

Strings Literais

- Quando passamos um string literal para o `printf` estamos passando ele como argumento
- O que realmente estamos passando?

Strings Literais

- C trata um string literal como um array de caracteres
- Um string de `n` caracteres é armazenado em um array de `n+1` posições
    - 1 posição para cada caractere e 1 posição no final para o caractere nulo que marca o fim do string
    - O caractere nulo é representado por `\0`, ele ocupa um byte e todos os seus bits são 0

Strings Literais

- O string literal `"abc"` é armazenado em um array de 4 caracteres:
  - `{'a', 'b', 'c', '\0'}`
- Um string literal vazio ocupa apenas um caractere
  - `{ \0 }`

Strings Literais

- Como um string literal é armazenado como um array, o compilador o trata como um ponteiro `char *`
- `printf` e `scanf` esperam um valor do tipo `char *` como argumento.
- Quando `printf("abc")` é chamado, o endereço para `"abc"` é passado para função.

Strings Literais

- No geral, podemos usar strings literais onde for permitido um `char *`

```
char *p;
p = "abc";
```

Strings Literais

- É possível usar indexação em strings literais

```
char c;
ch = "abc"[1];
```

- `c` vai receber o caractere `b`

Variável String

- Qualquer array de caracteres pode ser utilizado para armazenar um string

```
#define STR_TAM 80

char str[STR_TAM+1]
```

- `str` pode armazenar strings de **até** 80 caracteres
- O tamanho do string depende da posição do `\0`
- Então, `str` pode armazenar strings de tamanhos variados, mas que tenham no máximo 80 caracteres

Variável String

- Uma variável que armazena um string pode ser inicializada assim:

```
char data[12] = "14 de junho"
```

- O compilador vai armazenar:
  - `{'1', '4', ' ', 'd', 'e', ' ', 'j', 'u', 'n', 'h', 'o', '\0'}`

Variável String

- Se o tamanho do array for maior que o string, o compilador adiciona `\0` até o fim do array

```
char data[13] = "14 de junho"
```

- `{'1', '4', ' ', 'd', 'e', ' ', 'j', 'u', 'n', 'h', 'o', '\0', '\0'}`

Variável String

- Podemos omitir o tamanho do array que o compilador infere o tamanho

```
char data[] = "14 de junho"
```

- O compilador vai armazenar:
  - `{'1', '4', ' ', 'd', 'e', ' ', 'j', 'u', 'n', 'h', 'o', '\0'}`

Variável String

- `char date[] = "14 de junho"`
- `char *date = "14 de junho"`
- As duas declarações são semelhantes, mas não iguais
- Na versão usando array, os caracteres podem ser modificados, no ponteiro não
- Na versão usando ponteiros, `date` pode ser usado para apontar para outros strings, no array não

Escrevendo Strings

- Usando a conversão `%s` no `printf` conseguimos exibir strings

```
char str[] = "alo mundo";

printf("%s\n", str);
```

Escrevendo Strings

- Usando a conversão `%.ps`, onde `p` é um número, apenas os `p` primeiros caracteres são exibidos

```
char str[] = "alo mundo";

printf("%.3s\n", str);
```

Escrevendo Strings

- Usando a conversão `%ms`, onde `m` é um número, se o string for menor que `m` então o string é justificado a direita considerando um campo de `m` caracteres

```
char str[] = "alo mundo";

printf("%10s\n", str);
```

Escrevendo Strings

- C ainda traz a função `puts` que exibe um string
- `puts` tem apenas um argumento, o string a ser exibido
- `puts` coloca uma quebra de linha no final automaticamente

```
char str[] = "alo mundo";

puts(str);
```

Lendo Strings

- A conversão `%s` permite o `scanf` ler strings e guardá-los num array de caracteres
  - `scanf("%s", str);`
- Não precisamos colocar o `&` pois `str` é um ponteiro
- O `scanf` lê o string até encontrar um espaço, tab ou quebra de linha
- Para ler strings com espaços em branco usamos o `gets`
  - A leitura é feita até encontrar uma quebra de linha

Lendo strings

- Se você precisar de mais flexibilidade ou controle é possível ler um string caractere por caractere
- Antes de escrever uma função própria para leitura de strings é importante levar em consideração:
    - Quais caracteres vão fazer a função terminar a leitura? Esse caractere vai ser guardado no string?
    - A função vai descartar ou considerar espaços em branco à esquerda?
    - O que a função vai fazer se o string for maior que a variável usada para armazená-lo?

Lendo strings

- A função a seguir não descarta caracteres em branco à esquerda, termina ao ler uma quebra de linha e descarta caracteres se o string for maior que a variável usada para armazená-lo

```
int ler_linha(char str[], int n);
```

- `str` é a variável que receberá o string, `n` o tamanho do array e a função retorna um inteiro representando a quantidade de caracteres lidos

Lendo strings

```
int ler_linha(char str[], int n) {
  int ch, i=0;

  while ((ch = getchar()) != '\n')
    if (i<n-1)
      str[i++] = ch;

  str[i] = `\0`;

  return i;
}
```

- `ch` é um `int` pois `getchar` retorna o caractere lido como um `int`

Acessando os caracteres de um string

- Como strings são armazenados como arrays, então podemos usar indexação para acessar os seus caracteres
- Vamos fazer uma função que conta espaços em branco em um string

Acessando os caracteres de um string

```
int contar_espacos(char s[]) {
  int cont=0;

  for (int i=0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      cont++;

  return cont;
}
```

Acessando os caracteres de um string

- Usando ponteiros, temos:

```
int contar_espacos(char *s) {
  int cont=0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      cont++;

  return cont;
}
```

Biblioteca C para Strings

- C fornece uma biblioteca com diversas funções para realizarmos operações em strings
- Para isso devemos usar `#include <string.h>`

Biblioteca C para Strings

- `strcpy` - string copy
- `char *strcpy(char *s1, const char *s2)`
- `strcpy` copia `s2` para `s1`
- `strcpy` retorna `s1`

Biblioteca C para Strings

- `strcpy` - string copy
- Como você não pode usar a atribuição diretamente para copiar strings, então o `strcpy` passa a ser fundamental

```
str2 = "abcd"           // Erro
strcpy(str2, "abcd")  // str2 agora contém "abcd"
strcpy(str1, str2)    // str1 agora contém "abcd"
```

Biblioteca C para Strings

- `strlen` - string length
- `size_t strlen(const char *s)`
- `size_t` é um `typedef` interno que representa um tipo `unsigned int`
- `strlen` retorna o tamanho do string `s` (não inclui o caractere nulo)

Biblioteca C para Strings

- `strlen` - string length

```c
int len;

len = strlen("abc");  // len é 3
len = strlen("");     // len é 0
```

Biblioteca C para Strings

- `strcat` - string concatenation
- `char *strcat(char *s1, const char *s2)`
- `strcat` concatena o conteúdo de `s2` no final de `s1`
- `strcat` retorna `s1` que contém os dois strings concatenados

Biblioteca C para Strings

- `strcat` - string concatenation

```
strcpy(str1, "abc");
strcat(str1, "def");  // str1 agora contém "abcdef"
```

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);  // str1 agora contém "abcdef"
```

Biblioteca C para Strings

- `strcmp` - string comparison
- `int strcmp(const char *s1, const char *s2)`
- `strcmp` compara os strings `s1` e `s2`
- `strcmp` retorna um valor maior, igual ou menor que 0 dependendo se `s1` é maior, igual ou menor que `s2`

Biblioteca C para Strings

- `strcmp` compara os strings de acordo com sua ordem lexicográfica
- Lembra a ordem alfabética, mas...
- As letras maiúsculas são menores que as minúsculas
    - O código ASCII nos ajuda a entender isso
- Dígitos são menores que letras
- Espaços são menores que qualquer outro caractere