Strings

Representação de strings

Prof. Edson Alves - UnB/FGA

Sumário

Strings em C

Strings na linguagem C

- Em C, uma string é implementada como um *array* de caracteres terminado em zero (`'\0'`)

Strings na linguagem C

- Em C, uma string é implementada como um *array* de caracteres terminado em zero (`'\0'`)
- Esta implementação é a mais sintética possível em termos de memória: é reservado espaço apenas para armazenar os caracteres da string mais o terminador `'\0'`

Strings na linguagem C

- Em C, uma string é implementada como um *array* de caracteres terminado em zero (`'\0'`)
- Esta implementação é a mais sintética possível em termos de memória: é reservado espaço apenas para armazenar os caracteres da string mais o terminador `'\0'`
- Em contrapartida, a ausência deste marcador pode levar a execução errônea de várias das funções que manipulam strings

Strings na linguagem C

- Em C, uma string é implementada como um *array* de caracteres terminado em zero (`'\0'`)
- Esta implementação é a mais sintética possível em termos de memória: é reservado espaço apenas para armazenar os caracteres da string mais o terminador `'\0'`
- Em contrapartida, a ausência deste marcador pode levar a execução errônea de várias das funções que manipulam strings
- Além disso, rotinas simples, como determinar o tamanho de uma string s, passam a ter complexidade $O(|s|)$, contrastando com as implementações que utilizam memória adicional e que podem retornar o tamanho em $O(1)$

Declaração e inicialização de strings em C

- Uma string pode ser declarada e inicializada em C conforme os exemplos abaixo

```c
char s1[101];           // Declaração da string s1
char s2[] = "Test";     // Inicialização da string s2
```

Declaração e inicialização de strings em C

- Uma string pode ser declarada e inicializada em C conforme os exemplos abaixo

```c
char s1[101];           // Declaração da string s1
char s2[] = "Test";     // Inicialização da string s2
```

- A strings s1, não inicializada, comporta até 100 caracteres (e o terminador `'\0'`)

Declaração e inicialização de strings em C

- Uma string pode ser declarada e inicializada em C conforme os exemplos abaixo

```c
char s1[101];           // Declaração da string s1
char s2[] = "Test";     // Inicialização da string s2
```

- A strings s1, não inicializada, comporta até 100 caracteres (e o terminador `'\0'`)
- A string s2, inicializada com o valor `"Test"`, não exige que seja informado o número de caracteres e nem o terminador (o compilador completa tais informações automaticamente)

Declaração e inicialização de strings em C

- Uma string pode ser declarada e inicializada em C conforme os exemplos abaixo

```c
char s1[101];           // Declaração da string s1
char s2[] = "Test";     // Inicialização da string s2
```

- A strings s1, não inicializada, comporta até 100 caracteres (e o terminador `'\0'`)
- A string s2, inicializada com o valor `"Test"`, não exige que seja informado o número de caracteres e nem o terminador (o compilador completa tais informações automaticamente)
- Importante notar que, devido a aritmética de ponteiros da linguagem, as strings em C tem como primeiro elemento indexado em zero, não em um

Declaração e inicialização de strings em C

- Uma string pode ser declarada e inicializada em C conforme os exemplos abaixo

```c
char s1[101];           // Declaração da string s1
char s2[] = "Test";     // Inicialização da string s2
```

- A strings s1, não inicializada, comporta até 100 caracteres (e o terminador `'\0'`)
- A string s2, inicializada com o valor `"Test"`, não exige que seja informado o número de caracteres e nem o terminador (o compilador completa tais informações automaticamente)
- Importante notar que, devido a aritmética de ponteiros da linguagem, as strings em C tem como primeiro elemento indexado em zero, não em um
- Assim, `s[2]` representa o terceiro, e não o segundo, elemento da string

Bibliotecas para manipulação de strings

- A biblioteca padrão do C oferece o *header* `string.h`, onde são declaradas várias funções para a manipulação de strings

Bibliotecas para manipulação de strings

- A biblioteca padrão do C oferece o *header* `string.h`, onde são declaradas várias funções para a manipulação de strings
- O *header* `stdlib.h` traz funções para conversão de strings para valores numéricos

Bibliotecas para manipulação de strings

- A biblioteca padrão do C oferece o *header* `string.h`, onde são declaradas várias funções para a manipulação de strings
- O *header* `stdlib.h` traz funções para conversão de strings para valores numéricos
- Ele também define três funções que permitem manipular a memória (`memcmp()`, `memset()`, `memcpy()`), através de comparações, atribuições e cópia, respectivamente, as quais são úteis para trabalhar com strings

Bibliotecas para manipulação de strings

- A biblioteca padrão do C oferece o *header* `string.h`, onde são declaradas várias funções para a manipulação de strings
- O *header* `stdlib.h` traz funções para conversão de strings para valores numéricos
- Ele também define três funções que permitem manipular a memória (`memcmp()`, `memset()`, `memcpy()`), através de comparações, atribuições e cópia, respectivamente, as quais são úteis para trabalhar com strings
- Outro arquivo útil para a manipulação de strings em C é o `ctype.h`, onde são definidas funções para a manipulação de caracteres

Exemplo de uso dos arquivos string.h e stdlib.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char a[50] = "Test", b[] = "TEP";

    // Tamanho da string
    int s = strlen(a);      // s = 4, o zero terminador não é computado
    s = strlen(b);          // s = 3

    // Comparação
    s = strcmp(a, b);       // s > 0, "Test" sucede "TEP" na ordem lexicográfica
    s = strcmp(b, a);       // s < 0, "TEP" precede "Test" na ordem lexicográfica

    s = strncmp(a, b, 1);   // s = 0, as strings são iguais no primeiro caractere
    s = strncmp(a, b, 2);   // s > 0, "Te" sucede "TE" na ordem lexicográfica

```

Exemplo de uso dos arquivos string.h e stdlib.h

```
20    // Cópia
21    strcpy(a, b);           // a = "TEP"
22    strncpy(a, "SA", 2);    // a = "SAP"
23
24    // Acesso aos elementos individuais
25    a[2] = 'T';             // a = "SAT"
26    a[0] = b[2];            // a = "PAT"
27
28    // Conversão de string para valores numéricos
29    strcpy(a, "123.45");
30    double d = atof(a);             // d = 123.45
31    int i = atoi(a);                // i = 123
32
33    strcpy(a, "10000000000000000000000000000000000000000");
34    long long ll = strtoll(a, NULL, 2);   // ll = 1099511627776, conversão em base binária
35
36    // Concatenação
37    strcpy(b, "xyz");       // b = "xyz"
38    strcpy(a, "abcba");     // a = "abcba"
```

Exemplo de uso dos arquivos string.h e stdlib.h

```
40      strcat(a, b);           // a = "abcbaxyz"
41      strncat(a, a, 3);       // a = "abcbaxyzabc"
42
43      // Busca de caracteres
44      char *p;
45
46      p = strchr(a, 'b');     // p - a = 1, índice da primeira ocorrência de 'b'
47      p = strrchr(a, 'b');    // p - a = 9, índice da última ocorrência de 'b'
48
49      p = strstr(a, "cba");   // p - a = 2, índice da primeira coorrência de "cba"
50      p = strstr(a, "dd");    // p = NULL, "dd" não é substring de a
51
52      i = strspn(a, "abc");   // i = 5, a[0..4] contém apenas caracteres em "abc"
53      i = strcspn(a, "z");    // i = 7, a[0..6] contém caracteres diferentes de "z"
54
55      return 0;
56  }
```

Exemplo de uso do arquivo ctype.h

```c
#include <stdio.h>
#include <ctype.h>

int main()
{
    char a[] = "Test with numbers: 0x1234";
    int r;

    a[0] = tolower(a[0]);       // a = "test with numbers: 0x1234"
    a[10] = toupper(a[10]);     // a = "test with Numbers: 0x1234"

    r = isalpha(a[0]);          // r != 0, 't' é alfabético
    r = isalpha(a[19]);         // r = 0, '0' não é alfabético
    r = isalnum(a[19]);         // r != 0, '0' é alfanumérico
    r = isblank(a[18]);         // r != 0, ' ' é um espaço em branco
    r = isdigit(a[1]);          // r = 0, 'e' não é um dígito decimal
    r = isxdigit(a[1]);         // r != 0, 'e' é um dígito hexadecimal

    return 0;
}
```

Strings em C++

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$
- Por outro lado, esta representação demanda mais memória do que a representação em C

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$
- Por outro lado, esta representação demanda mais memória do que a representação em C
- Existem técnicas para tentar reduzir a memória utilizada, como a *small string optimization*

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$
- Por outro lado, esta representação demanda mais memória do que a representação em C
- Existem técnicas para tentar reduzir a memória utilizada, como a *small string optimization*
- A classe fundamental dentre as várias classes que representam strings em C++ é a `std::string`.

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$
- Por outro lado, esta representação demanda mais memória do que a representação em C
- Existem técnicas para tentar reduzir a memória utilizada, como a *small string optimization*
- A classe fundamental dentre as várias classes que representam strings em C++ é a `std::string`.
- O método `c_str()` permite obter, a partir de uma string C++, uma representação compatível com a utilizada em C

Representação de strings em C++

- Embora seja possível utilizar a abordagem e as bibliotecas da linguagem C em C++, existe uma representação em C++ de strings baseada em classes
- O uso de classes para representar strings traz a vantagem de poder manter outras informações sobre a string sempre atualizadas e com acesso em $O(1)$
- Por outro lado, esta representação demanda mais memória do que a representação em C
- Existem técnicas para tentar reduzir a memória utilizada, como a *small string optimization*
- A classe fundamental dentre as várias classes que representam strings em C++ é a `std::string`.
- O método `c_str()` permite obter, a partir de uma string C++, uma representação compatível com a utilizada em C
- Deste modo, é possível utilizar funções escritas para strings em C a partir de instâncias da classe da linguagem C++

Exemplo de uso da classe string em C++

```cpp
#include <iostream>

using namespace std;

int main()
{
    string a { "Test" }, b { "TEP" };

    // Tamanho da string
    int s = a.size();       // s = 4
    s = b.size();           // s = 3

    // Comparação
    bool r;

    r = (a == b);           // r = false, "Test" e "TEP" são distintas
    r = (a != b);           // r = true, "Test" e "TEP" são distintas
    r = (a < b);            // r = false, "Test" sucede "TEP" na ordem lexicográfica
    r = (a > b);            // r = true, "Test" sucede "TEP" na ordem lexicográfica
```

Exemplo de uso da classe string em C++

```
21    s = a.compare(0, 1, b); // s = 0, as strings são iguais no primeiro caractere
22    s = a.compare(0, 2, b); // s > 0, "Te" sucede "TE" na ordem lexicográfica
23
24    // Cópia
25    a = b;                  // a = "TEP"
26
27    // Acesso aos elementos individuais
28    a[1] = 'A';             // a = "TAP"
29    auto c = a.front();     // c = 'T', primeiro elemento
30    c = a.back();           // c = 'P', último elemento
31
32    // Conversão de string para valores numéricos
33    a = "123.45";
34    auto d = stod(a);       // d = 123.45;
35    auto i = stoi(a);       // i = 123
36
37    a = "10000000000000000000000000000000000000000";
38    auto ll = stoll(a, NULL, 2);   // ll = 1099511627776, conversão em base binária
```

Exemplo de uso da classe string em C++

```
40    // Conversão de valores numéricos para strings
41    a = to_string(999);      // a = "999"
42    a = to_string(9.99);     // a = "9.99"
43
44    // Concatenação
45    b = "xyz";               // b = "xyz"
46    a = "abcba";             // a = "abcba"
47
48    a += b;                  // a = "abcbaxyz"
49    a += a.substr(0, 3);     // a = "abcbaxyzabc"
50
51    // Busca de caracteres
52    auto p = a.find('b');               // p = 1, índice da primeira ocorrência de 'b'
53    p = a.rfind('b');                   // p = 9, índice da última ocorrência de 'b'
54    p = a.find("cba");                  // p = 2, índice da primeira ocorrência de "cba"
55    p = a.find("dd");                   // p = string::npos, "dd" não é substring de a
56    p = a.find_first_not_of("abc");     // i = 5, a[0..4] contém apenas caracteres em "abc"
57    p = a.find_first_of("z");           // i = 7, a[0..6] contém caracteres diferentes de "z"
```

Exemplo de uso da classe string em C++

```
59      // Exemplo de uso do método c_str()
60      a = "Test";
61      printf("%s\n", a.c_str());
62
63      return 0;
64  }
```

Strings em Python

Representação de strings em Python

- Embora as strings em Python também sejam implementadas através de classes, elas podem ser vistas informalmente como listas de caracteres

Representação de strings em Python

- Embora as strings em Python também sejam implementadas através de classes, elas podem ser vistas informalmente como listas de caracteres
- Em Python, constantes do tipo string podem ser representados usando-se aspas simples ou duplas, ou mesmo triplas (para strings com múltiplas linhas)

Representação de strings em Python

- Embora as strings em Python também sejam implementadas através de classes, elas podem ser vistas informalmente como listas de caracteres
- Em Python, constantes do tipo string podem ser representados usando-se aspas simples ou duplas, ou mesmo triplas (para strings com múltiplas linhas)
- Para maratonas de programação, o módulo `string` da linguagem Python traz constantes bastantes úteis, como listagens de caracteres comuns:

```
import string

a = string.ascii_lowercase        # a = 'abcdefghijklmnopqrstuvwxyz'
b = string.ascii_uppercase        # b = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
c = string.ascii_letters          # c = a + b
d = string.digits                 # d = '0123456789'
x = string.hexdigits              # x = '0123456789abcdefABCDEF'
```

Representação de strings em Python

- Mesmo que a solução proposta pela equipe seja escrita em outra linguagem, estas constantes podem ser facilmente acessadas via terminal, importando o módulo e usando o comando (ou função, no Python 3) `print`

Representação de strings em Python

- Mesmo que a solução proposta pela equipe seja escrita em outra linguagem, estas constantes podem ser facilmente acessadas via terminal, importando o módulo e usando o comando (ou função, no Python 3) `print`
- Outra particularidade do Python é que, ao contrário das linguagens C e C++, ele suporta índices negativos para strings

Representação de strings em Python

- Mesmo que a solução proposta pela equipe seja escrita em outra linguagem, estas constantes podem ser facilmente acessadas via terminal, importando o módulo e usando o comando (ou função, no Python 3) `print`
- Outra particularidade do Python é que, ao contrário das linguagens C e C++, ele suporta índices negativos para strings
- Por exemplo, `s[-1]` se refere ao último caractere, `s[-2]` ao penúltimo, e assim por diante

Representação de strings em Python

- Mesmo que a solução proposta pela equipe seja escrita em outra linguagem, estas constantes podem ser facilmente acessadas via terminal, importando o módulo e usando o comando (ou função, no Python 3) `print`
- Outra particularidade do Python é que, ao contrário das linguagens C e C++, ele suporta índices negativos para strings
- Por exemplo, `s[-1]` se refere ao último caractere, `s[-2]` ao penúltimo, e assim por diante
- Outra notação útil é `s[::-1]`, que indica o reverso da string s (isto é, s lida do fim para o começo)

Representação de strings em Python

- Mesmo que a solução proposta pela equipe seja escrita em outra linguagem, estas constantes podem ser facilmente acessadas via terminal, importando o módulo e usando o comando (ou função, no Python 3) `print`
- Outra particularidade do Python é que, ao contrário das linguagens C e C++, ele suporta índices negativos para strings
- Por exemplo, `s[-1]` se refere ao último caractere, `s[-2]` ao penúltimo, e assim por diante
- Outra notação útil é `s[::-1]`, que indica o reverso da string s (isto é, s lida do fim para o começo)
- A API para strings em Python contempla ainda muitas outras funções úteis, como `strip()`, `join()` e `split()`

Exemplo de uso de strings em Python

```python
# -*- coding: utf-8 -*-

if __name__ == '__main__':

    a = "Test"
    b = "TEP"

    # Tamanho da string
    s = len(a)          # s = 4
    s = len(b)          # s = 3

    # Comparação
    r = (a == b)        # r = False, "Test" e "TEP" são distintas
    r = (a != b)        # r = True, "Test" e "TEP" são distintas
    r = (a < b)         # r = False, "Test" sucede "TEP" na ordem lexicográfica
    r = (a > b)         # r = True, "Test" sucede "TEP" na ordem lexicográfica

    s = a[:1] == b[:1]  # s = True, as strings são iguais no primeiro caractere
    s = a[:2] == b[:2]  # s = False, "Te" sucede "TE" na ordem lexicográfica
```

Exemplo de uso de strings em Python

```
21    # Cópia
22    a = b                # a = "TEP"
23
24    # Acesso aos elementos individuais
25    #a[1] = 'A'          # Erro! Strings em Python são imutáveis
26    c = a[0]             # c = 'T', primeiro elemento
27    c = a[-1]            # c = 'P', último elemento
28
29    # Conversão de string para valores numéricos
30    a = "123.45"
31    d = float(a)         # d = 123.45
32
33    a = "10000000000000000000000000000000000000000";
34    ll = int(a, 2)       # ll = 1099511627776, conversão de base binária
35
36    # Conversão de valores numéricos para strings
37    a = str(999)         # a = "999"
38    a = str(9.99)        # a = "9.99"
```

Exemplo de uso de strings em Python

```
40    # Concatenação
41    b = "xyz"            # b = "xyz"
42    a = "abcba"          # a = "abcba"
43    a += b               # a = "abcbaxyz"
44    a += a[:3]           # a = "abcbaxyzabc"
45
46    # Busca de caracteres
47    p = a.find('b')      # p = 1, índice da primeira ocorrência de 'b'
48    p = a.rfind('b')     # p = 9, índice da última ocorrência de 'b'
49    p = a.find("cba")    # p = 2, índice da primeira ocorrência de "cba"
50    p = a.find("dd")     # p = -1, "dd" não é substring de a
51
52    # Exemplo de uso do método strip()
53    a = "  Espaços antes e depois    "
54    b = a.strip()        # b = "Espaços antes e depois"
55
56    # Exemplo de uso do método join()
57    xs = ["1", "2", "3", "4", "5"]
58    a = ', '.join(xs)    # a = "1, 2, 3, 4, 5"
```

Exemplo de uso de strings em Python

```
60    t = ['22', '05', '43']
61    b = ':'.join(t)     # b = "22:05:43
62
63    # Exemplo de uso do método split()
64    a = "Frase com quatro palavras"
65    b = a.split()       # b = ['Frase', 'com', 'quatro', 'palavras']
66
67    a = "abacad"
68    b = a.split('a')    # b = ['', 'b', 'c', 'd']
69
70    # Exemplo de uso de métodos de alteram o case dos caracteres
71    a = "Tep"
72    b = a.lower()       # b = "tep"
73    c = a.upper()       # c = "TEP"
74    d = a.swapcase()    # d = "tEP"
```

Entrada e saída de strings em console

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`
- O marcador utilizado para o tipo string é o `%s`

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`
- O marcador utilizado para o tipo string é o `%s`
- A função `scanf()` fará a leitura da entrada até encontrar um caractere de espaço (quebra de linha, tabulações, espaço em branco, etc)

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`
- O marcador utilizado para o tipo string é o `%s`
- A função `scanf()` fará a leitura da entrada até encontrar um caractere de espaço (quebra de linha, tabulações, espaço em branco, etc)
- Se a intenção é ler uma linha na íntegra, deve ser utilizada a função `fgets()`

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`
- O marcador utilizado para o tipo string é o `%s`
- A função `scanf()` fará a leitura da entrada até encontrar um caractere de espaço (quebra de linha, tabulações, espaço em branco, etc)
- Se a intenção é ler uma linha na íntegra, deve ser utilizada a função `fgets()`
- A função `fgets()` é mais segura que a `scanf()`, pois utiliza o segundo parâmetro como limite máximo de caracteres (incluindo o zero terminador) a serem lidos e escritos no primeiro parâmetro

I/O de strings em C

- Cada linguagem tem mecanismos apropriados para a leitura e escritas de strings a partir do terminal
- Em C, são utilizadas as funções `printf()` e `scanf()`
- O marcador utilizado para o tipo string é o `%s`
- A função `scanf()` fará a leitura da entrada até encontrar um caractere de espaço (quebra de linha, tabulações, espaço em branco, etc)
- Se a intenção é ler uma linha na íntegra, deve ser utilizada a função `fgets()`
- A função `fgets()` é mais segura que a `scanf()`, pois utiliza o segundo parâmetro como limite máximo de caracteres (incluindo o zero terminador) a serem lidos e escritos no primeiro parâmetro
- Vale notar que a função `fgets()` insere, no primeiro parâmetro, o caractere de nova linha, se o encontrar (a função também termina se for encontrado o caractere `EOF`, que indica o fim do arquivo)

Exemplo de I/O de strings em C

```c
#include <stdio.h>
#include <string.h>

// Assuma que será inserida em uma linha, via console, a mensagem
// "Teste de I/O em C"
int main()
{
    char s[1024], line[1024];

    scanf("%s", s);
    printf("s = [%s]\n", s);            // s = [Teste]

    fgets(line, 1024, stdin);
    line[strlen(line) - 1] = 0;

    printf("line = [%s]\n", line);      // line = [ de I/O em C]

    return 0;
}
```

I/O de strings em C++

- Em C++, strings podem ser lidas e escritas com os operadores << e >> das classes `cin` e `cout`, respectivamente

I/O de strings em C++

- Em C++, strings podem ser lidas e escritas com os operadores << e >> das classes `cin` e `cout`, respectivamente
- A classe `cin` se comporta de forma semelhante à função `scanf()`, lendo a entrada até encontrar um caractere que indique um espaço em branco

I/O de strings em C++

- Em C++, strings podem ser lidas e escritas com os operadores << e >> das classes `cin` e `cout`, respectivamente
- A classe `cin` se comporta de forma semelhante à função `scanf()`, lendo a entrada até encontrar um caractere que indique um espaço em branco
- Para ler linhas inteiras, de forma semelhante à `fgets()`, basta usar a função `getline()`

I/O de strings em C++

- Em C++, strings podem ser lidas e escritas com os operadores << e >> das classes `cin` e `cout`, respectivamente
- A classe `cin` se comporta de forma semelhante à função `scanf()`, lendo a entrada até encontrar um caractere que indique um espaço em branco
- Para ler linhas inteiras, de forma semelhante à `fgets()`, basta usar a função `getline()`
- Porém, diferentemente da função `fgets()`, a função `getline()` despreza o caractere de fim de linha, e não o insere na string apontada pelo segundo parâmetro

Exemplo de I/O de strings em C++

```cpp
#include <iostream>

using namespace std;

// Assuma que será inserida em uma linha, via console, a mensagem
// "Teste de I/O em C++"
int main()
{
    string s, line;

    cin >> s;
    cout << "s = [" << s << "]\n";              // s = [Teste]

    getline(cin, line);
    cout << "line = [" << line << "]\n";        // line = [ de I/O em C++]

    return 0;
}
```

I/O de strings em Python

- Em Python 2, strings podem ser lidas e escritas por meio da função `raw_input()` e pelo comando `print`

I/O de strings em Python

- Em Python 2, strings podem ser lidas e escritas por meio da função `raw_input()` e pelo comando `print`
- A função `raw_input()` se comporta de maneira semelhante à função `getline()` do C++

I/O de strings em Python

- Em Python 2, strings podem ser lidas e escritas por meio da função `raw_input()` e pelo comando `print`
- A função `raw_input()` se comporta de maneira semelhante à função `getline()` do C++
- O comando `print` insere, automaticamente, uma quebra de linha após a impressão de sua mensagem

I/O de strings em Python

- Em Python 2, strings podem ser lidas e escritas por meio da função `raw_input()` e pelo comando `print`
- A função `raw_input()` se comporta de maneira semelhante à função `getline()` do C++
- O comando `print` insere, automaticamente, uma quebra de linha após a impressão de sua mensagem
- Para suprimir este comportamento, deve-se usar uma vírgula ao final da mensagem, a qual substitui a quebra de linha por um espaço em branco

I/O de strings em Python

- Em Python 2, strings podem ser lidas e escritas por meio da função `raw_input()` e pelo comando `print`
- A função `raw_input()` se comporta de maneira semelhante à função `getline()` do C++
- O comando `print` insere, automaticamente, uma quebra de linha após a impressão de sua mensagem
- Para suprimir este comportamento, deve-se usar uma vírgula ao final da mensagem, a qual substitui a quebra de linha por um espaço em branco
- Em Python 3, a função `raw_input()` foi renomeada para `input()`, e o comando `print` foi substituído pela função `print()`

Exemplo de I/O de strings em Python

```
1  # -*- coding: utf-8 -*-
2
3  # Assuma que será inserida em uma linha, via console, a mensagem
4  # "Teste de I/O em Python"
5
6  s = input()
7  print(f's = [{s}]')          # s = [Teste de I/O em Python]
8
9  s = s.split()[0]
10 print(f's = {s}')            # s = Teste
```

Referências

1. **AHLGREEN**, John. Small String Optimization and Move Operators, acesso em 16/12/2016.
2. CppReference. Null-terminated byte strings, acesso em 21/12/2016.
3. CppReference. std::basic_string, acesso em 21/12/2016.
   DAVID. A look at std::string implementations in C++, acesso em 22/12/2016.
4. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
5. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
6. Python Documentation. Common string operations, acesso em 26/12/2016.