# Strings

Dr. Mariano Méndez, Juan Pablo Capurro

Facultad De Ingeniería. Universidad de Buenos Aires

2 de octubre de 2018

## 1. Introducción

En el contexto de un lenguaje de programación, un string es una cadena de caracteres usada para representar texto. Fueron usados desde el comienzo de la cursada, por ejemplo, en el programa de *Hola mundo*

```c
#include<stdio.h>
int main(){
    printf("Hello, World!");
    return 0;
}
```

`Hello, World!` es un string literal[**?**], de la misma forma que `0` es un int literal.

Sin embargo, todos los strings que tratamos hasta ahora fueron siempre literales, es decir, siempre *hardcodeados* en los fuentes del programa y nunca asignados a una variable de ningún tipo. El mismo programa podría ser reescrito de la siguiente manera:

```c
#include<stdio.h>
int main() {
  char* cadena_salida = "Hello, world!
    ";
  printf("%s", cadena_salida);
  return 0;
}
```

Ahora `Hello, World!` está siendo asignado a una variable, y el valor de dicha variable es usado.
El modificador `%s` indica a `printf` que el argumento a mostrar es un string.

### 1.1. Implementación en C

Se vio en la sección anterior que puede asignarse un string a un `char *`. Esto es debido a que, en C, la representación interna de los strings en memoria está expuesta al programador (cosa que no es así en todos los lenguajes).
En C, los strings son simples arreglos de caracteres (`char`s), por esto es que se usa un `char *` para guardarlos: es un puntero a la primer posición de dicho vector.
De tal manera, ambos strings serían equivalentes:

```c
char* cadena_salida = "Hello, world!";
```

```c
char* cadena_salida = {'H','e','l','l','o',',',' ','w','o','r','l','d','!','\0'};
```

Una pregunta no muy evidente que puede surgir de esto es ¿Cómo sabe, por ejemplo, `printf` dónde termina el string? Es decir, debe haber una forma de distinguir, por ejemplo, entre los strings `'Hello'` y `'Hello, World!'`.
C usa los llamados *null-terminated strings*, designando al valor `0` (`0xb00000000` en binario, `'\0'` en la tabla ASCII) como el fin de string.

### Ejercicio 1

Escribir un programa que muestre un string por pantalla, caracter a caracter, es decir, sin hacer uso de `puts`, `printf` con modificador `%s` o similares.
**Análisis:** como los strings son un vector de caracteres con un caracter especial que marca su finalización, lo lógico es iterar sobre el vector, con un índice, y dejar de iterar al llegar a ese caracter especial.
**Hint:** La función `putchar`, de `stdio.h`, cumple las necesidades de entrada/salida de este programa.

1. ¿Cuál es el tipo de iteración que correspondería implementar: definida o indefinida? ¿Bucle `for` o `while`? ¿Por qué?

### Ejercicio 2

Escribir una función que devuelva el largo de un string pasado por parámetro.
**Hint:** Al igual que en el ejercicio anterior, se deberá iterar sobre todo el string, pero en vez de mostrar cada uno de sus caracteres, contarlos.

1. ¿Qué tipo de dato debería devolver esta función?
2. ¿Qué valor debería devolver la función para el siguiente vector?

```
1 const char  string_definido_raro[15] = {'r', '4', 'r', '0', '\0'};
```

   ¿Por qué?
3. Proveer datos de ejemplo y resultados con los que probar la función. En la implementación, puede usarse la función `assert()`, de `assert.h`
4. ¿Cuál es el tipo de iteración que correspondería implementar: definida o indefinida? ¿Bucle `for` o `while`? ¿Por qué?
5. ¿Qué pasaría si se pasa un string sin el marcador de finalización?

## 2. Biblioteca estándar de C y strings

Por muy formativo que resulte, como programadores, reinventar la rueda es algo que debería ser evitado. Por esto es que la biblioteca estándar de C nos provee con una serie de funciones, declaradas en `string.h` que implementan, entre otras cosas, los ejercicios anteriores, además de proveer otras facilidades.

### 2.1. Definiciones

#### 2.1.1. `size_t strlen(const char* str)`

Devuelve el largo de un string hasta el caracter delimitador `'\0'`, sin contarlo.

#### 2.1.2. `int strcmp(const char* str_left, const char* str_right)`

Analiza los dos strings pasados por parámetro para chequear su igualdad. Si son iguales, devuelve `0`. Si no son iguales, devuelve un número menor a `0` si `str_left` es lexicográficamente[**?**] menor a `str_right`, y un número mayor a `0` en el caso de que `str_left` sea mayor a `str_right`

#### 2.1.3. `char* strcpy(char* dest, const char* src)`

Copia el string `src` a `dest`. Asume que `src` está bien formado y que se alojó en `dest` memoria suficiente. Copia el caracter finalizador.

## 2.2. Consideraciones

Todas las funciones anteriores asumen que el string está bien formado, es decir, que cuentan con el caracter finalizador. En el caso de que esto no sea cierto, todas ellas tienen comportamiento indefinido [**?**], probablemente terminando por segfault. Existen versiones más seguras de las últimas dos funciones, `int strncmp(const char* str1, const char* str2, size_t n)` y `char* strncpy(char* dest, const char* src, size_t n)`, que cumplen la misma función, pero con un tope `n`, lo que evita acceder a memoria inválida en el caso de que algún string esté mal formado. El parámetro `n` debería ser seteado a la cantidad de memoria que se sabe que se reservó para ese string.

### Ejercicio 3

1. Probar la implementación de la función realizada en el ejercicio **??** contra la implementación de la biblioteca estándar `strlen`, haciendo uso `assert`.
2. De la misma forma, codificar y probar contra `strcmp` una función que devuelva la diferencia lexicográfica entre dos strings. Esto es útil para tener bien claro qué es lo que devuelve `strcmp`.

#### Solución del ejercicio ??

1. Correspondería usar una iteración indefinida, con un bucle `while`, porque al comenzar a iterar no se sabe cuántas iteraciones se realizarán.
   **Implementación:**

```c
#include<stdio.h>

const char* string_vacio             = "";
const char* string_largo             = "comandante, we all miss you";
const char  string_definido_raro[15] = {'r', '4', 'r', '0', '\0'};

void diy_puts(const char* string){
    int i = 0;
    while(string[i] != '\0'){
        putchar(string[i]);
        i++;
    }
}

int main(){
    diy_puts(string_largo);
    diy_puts("\n");
    diy_puts(string_vacio);
    diy_puts("\n");
    diy_puts(string_definido_raro);
    return 0;
}
```

#### Solución del ejercicio ??

1. La función debería devolver un `size_t`, por ser el tipo de dato que usa la librería estándar de C para representar tamaños.
   Sin embargo, cualquier tipo de dato entero que pueda almacenar el largo de un string sería suficiente. Un `char`, por ejemplo, no sería suficiente.
2. Corresponde que el largo evalúe a `4`, ya que hay cuatro caracteres hasta el señalizador de fin de string. El `15` se corresponde con la cantidad de `char`s reservados en memoria para el string.
3. Datos de ejemplo proveídos en la implementación.
4. Correspondería usar una iteración indefinida, con un bucle `while`, porque al comenzar a iterar no se sabe cuántas iteraciones se realizarán.

5. Si se omitiera el marcador de fin de string, el programa no tendría forma de enterarse dónde termina el string, por lo que seguiría leyendo hasta intentar leer memoria inválida y terminaría por segfault.
**Implementación:**

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

const char* string_vacio             = "";
const char* string_largo_10          = "comandante";
const char  string_definido_raro[15] = {'r', '4', 'r', '0', '\0'};

size_t largo_string(const char* string){
    size_t largo = 0, i = 0;
    while(string[i] != '\0'){
        i++;
        largo++;
    }
    return largo;
}

int main(){
    assert(largo_string(string_vacio)         == 0);
    assert(largo_string(string_largo_10)      == 10);
    assert(largo_string(string_definido_raro) == 4);
    return 0;
}
```

## Solución del ejercicio ??

**Implementación:**

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

const char* string_vacio             = "";
const char* string_largo_10          = "comandante";
const char  string_definido_raro[15] = {'r', '4', 'r', '0', '\0'};

size_t largo_string(const char* string){
    size_t largo = 0, i = 0;
    while(string[i] != '\0'){
        i++;
        largo++;
    }
    return largo;
}
int diy_strcmp(const char* un_string, const char* otro_string){
    size_t i=0;
    while(un_string[i] == otro_string[i]){
        if(!un_string[i]){
            return 0;
        }
        i++;
    }
    return un_string[i] - otro_string[i];
}

int main(){
    puts("Pruebas de largo_string()");
    assert(largo_string(string_vacio)         == strlen(string_vacio));
    assert(largo_string(string_largo_10)      == strlen(string_largo_10));
    assert(largo_string(string_definido_raro) == strlen(string_definido_raro));
    puts("Pruebas de largo_string(): OK");
    puts("Pruebas de diy_strcmp()");
```

```
35      assert(strcmp(string_largo_10, "comandante") == diy_strcmp(string_largo_10, "
        comandante"));
36      assert(strcmp(string_largo_10, string_definido_raro) == diy_strcmp(
        string_largo_10, string_definido_raro));
37      puts("Pruebas de diy_strcmp(): OK");
38      return 0;
39  }
```

## Referencias

[1] webopedia define un literal como un valor que será interpretado tal como fue escrito, a diferencia de una variable, que almacena un valor que puede variar a lo largo de la ejecución de un programa.

[2] wikipedia define formalmente el orden lexicográfico. A nosotros nos importa saber que es el orden alfabético, como en la guía telefónica.

[3] En C, se considera comportamiento indefinido al hecho de que no esté especificado qué sucederá frente a unas condiciones dadas. Esto no quiere decir que no pueda saberse, sino que quien desarrolló esa función en particular no se preocupa en documentarlo porque está fuera de como esta debería ser usada. Un estudiante curioso, o un usuario con malas intenciones podría, a veces fácilmente, saber qué sucederá en tal caso. A menudo el resultado depende del entorno, como el sistema operativo sobre que está corriendo el programa.