# Cadenas

- Lenguaje C no maneja cadenas (strings) como un elemento del lenguaje, sino que lo asimila a secuencia de caracteres con un “centinela” (`'\0'`) que indica el final de la secuencia.
- Por supuesto que reservar memoria para las cadenas es tarea del programador. Los métodos habituales son declarar un vector de caracteres o usar la función `malloc()`.
- La lógica para manipular cadenas es recorrerlas y operarlas carácter a carácter.
- La biblioteca estándar `string.h` provee varias funciones para operar con cadenas.

# Definir como vector

- Si declaro como un vector
  - `char palabra[10];`
    - Permite guardar cadenas de hasta 9 caracteres, ya que debemos contemplar el '\0' final.
  - `char frase[] = "Hola mundo";`
    - El vector frase se dimensiona automáticamente de 11 posiciones (10 caracteres + '\0' final)
  - `char frase[100] = "Hola mundo";`
    - Similar al caso anterior, con las posiciones no inicializadas explícitamente en cero, como cualquier vector.
    - **OJO**: Si pongo un literal más largo que la dimensión del vector, recorta pero **NO** pone el '\0' final

# Definir como vector

- El uso de literal cadena es equivalente a haber inicializado cada posición :

```
char palabra[5] = "Hola";
char palabra[5] = {'H', 'o', 'l', 'a', '\0'};
```

- Como con cualquier vector **solo sirve para inicializar**, la siguiente secuencia es **incorrecta**
  - `char palabra[10];`
  - `palabra = "Hola";`
- El modo **correcto** sería
  - `char palabra[10];`
  - `strcpy(palabra, "Hola");`

# Definir con punteros

- Puedo simplemente definir un puntero a char, pero eso no reserva memoria para los datos, debo hacerlo aparte

```
char *cadena;
cadena = malloc(20);
strcpy(cadena, "Hola");
```

- Si la zona de memoria ya está reservada puede apuntarla directamente

```
char buffer[512];
char *cadena = buffer;
char *saludo = "Hola";
```

- El tipo de dato de un literal cadena es arreglo de caracteres, lo que a efectos prácticos es `char*`. Puede que no sea modificable

# Funciones de string.h

- `size_t strlen(const char *s);`
  - Devuelve la cantidad de caracteres en la cadena s (sin incluir el '\0')
- Ejemplo

```
char pais[20] = "Argentina";
char *p = malloc(strlen(pais) + 1);
strcpy(p, pais);
```

- Más simple:

```
char pais[20] = "Argentina";
char *p = strdup(pais);
//strdup NO estandar de C (es posix)
```

# Duplicar cadena

- `char *strdup(const char *s);`
  - Duplica la cadena s, pide memoria con malloc y copia la cadena. Devuelve puntero a la cadena duplicada o NULL si no hay suficiente memoria
- `char *strndup(const char *s, size_t n);`
  - Similar a la anterior pero copia a lo sumo n caracteres y luego agrega `'\0'`
- Usar `malloc` y `strcpy` si `strdup` no está disponible (p.ej: en mingw)

# Copiar cadena

- `char *strcpy(char * restrict s1, const char * restrict s2);`
  - Copia la cadena apuntada por s2 en la memoria apuntada por s1. Si al copiar hay solapamiento el resultado es indefinido. Devuelve el valor de s1
- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`
  - Similar a la anterior pero copiando a lo sumo n caracteres. Esto implica que puede **NO** copiar el `'\0'` final!!!. Si largo de la cadena copiada es menor a n, el resto de las posiciones se llenan con `'\0'`.

# Concatenar cadena

- ```
  char *strcat(char * restrict s1,
               const char * restrict s2);
  ```
  - Agrega una copia de la cadena apuntada por s2 al final de la cadena s1. Si hay solapamiento resultado es indefinido. Devuelve el valor de s1
- ```
  char *strncat(char * restrict s1,
                const char * restrict s2,
                size_t n);
  ```
  - Similar a la anterior pero agrega a los sumo n caracteres de s2. Si al copiar n caracteres el último NO es un `'\0'`, lo agrega. Significa que la cadena resultante puede llegar a ser de largo strlen(s1)+n+1

# Comparar cadenas

- `int strcmp(const char *s1, const char *s2);`
  - Compara las cadenas apuntadas por s1 y s2. El entero devuelto será:
    - `> 0 si s1 > s2`
    - `< 0 si s1 < s2`
    - `= 0 si s1 = s2`
- `int strncmp(const char *s1, const char *s2, size_t n);`
  - Similar al anterior pero compara solo hasta el carácter n, o menos si alguna de las dos cadenas termina antes.

# Ejemplo a

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

int main()
{
        char s1[30] = "Hola";
        char *s2 =", que tal!";
        if (strlen(s1) + strlen(s2) < sizeof(s1))
                strcat(s1, s2);
        printf("%s\n", s1);
        //Imprime: Hola, que tal!
        return EXIT_SUCCESS;
}
```

# Ejemplo b

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

char *strmin(const char* str1, const char*str2)
{
        if (strcmp(str1, str2) < 0) {
                //cuando está disponible posix
                return strdup(str1);
        } else {
                // Cuando posix no está disponible
                char *temp = malloc(strlen(str2) + 1);
                strcpy(temp, str2);
                return temp;
        }
}
```

# Ejemplo b (continuación)

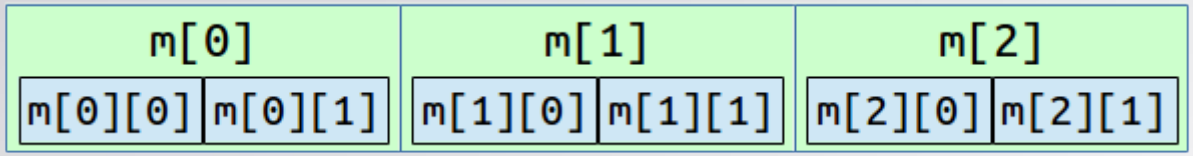
```c
int main()
{
        char *smin;
        char cand1[50];
        char cand2[50];
        printf("Nombre del primer candidato: ");
        scanf("%[^\n]%*c",cand1);
        printf("Nombre del segundo candidato: ");
        scanf("%[^\n]",cand2);

        smin = strmin(cand1, cand2);
        printf("Debe listarse primero a: %s\n", smin);
        free(smin);
        return EXIT_SUCCESS;
}
```

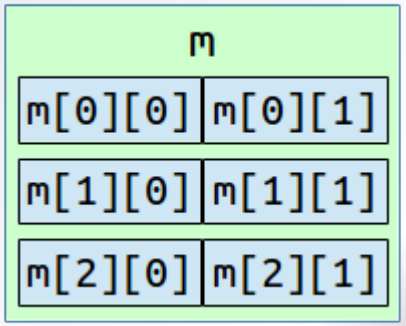
# Matrices

- No hay matrices como tales, solo vector de vector. Por ejemplo si declaramos

```
double mat[3][2];
```

- Entonces `mat` es un vector de 3 elementos, donde cada uno de ellos es a su vez un vector de 2 doubles.
- Como cada elemento es un vector de 2 doubles implica que en memoria se guardará la “matriz” por filas (cada fila es un elemento del “vector” `mat`)



# Matrices

- Si tengo un **vector de tipo T** al pasarlo a una función se degrada a **puntero a tipo T**. Por tanto si paso mat a una función degradará a: puntero a vector de 2 doubles: `(double(*)[2])`
- Una función con parámetro: **`T param[]`**se degrada a **puntero a T** y con eso le basta para saber como pasar de un elemento al siguiente.
- Con una matriz puedo hacer **`T param[][DIM]`** con **`DIM`** una expresión entera conocida en tiempo de compilación, ya que es un vector de vector. Esto le basta para saber como pasar al elemento siguiente.
- En resumen, solo la dimensión "más interna" puede no especificarse, el resto debe ser conocida en tiempo de compilación

# Licencia