Tema 4: Cadenas en profundidad

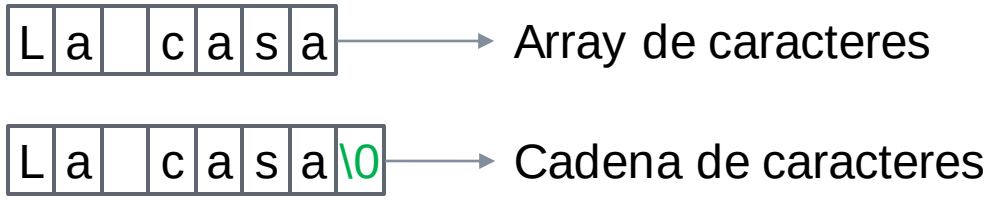
Introducción a la programación II

Marcos Novalbos
Alfonso Castro Escudero
Maximiliano Alfredo Fernández

Índice

1. Cadenas en C: Concepto

- El lenguaje C no tiene datos predefinidos tipo cadena.
- En su lugar C, manipula cadenas mediante arrays de caracteres que terminan con el carácter ASCII nulo (‘\0’).
- Una cadena se considera como un array unidimensional de tipo char o unsigned char.



El número total de caracteres de una cadena en C es siempre igual a la longitud del array más 1.

1. Cadenas y arrays en C

- ¿Qué diferencias y analogías existen entre las variables `c1`, `c2` y `c3`?.

```
char  **c1;
char  *c2 [10];
char  *c3 [10][21];
```

- La variable `c1` es un puntero que puede apuntar a un puntero a caracteres, pero no está inicializado con una dirección válida
- La variable `c2` es un array de 10 punteros a caracteres, pero estos 10 punteros no apuntan a ningún dato válido.
- La variable `c3` es una matriz con espacio para 210 punteros a caracteres no inicializados, accesibles según un array de 10 filas de 21 elementos cada uno de ellos.

2. Cadenas en C. Inicialización

- Una cadena no se puede inicializar fuera de la definición.
- La razón es que un identificador de cadena, como cualquier identificador de array, se trata como un valor de dirección, como un puntero constante.
- Puede inicializar un array de caracteres con un literal de cadena

```
char letras[] = "abc";
```

- Se inicializa letras como un array de caracteres de cuatro elementos. El cuarto elemento es el carácter null, ('\0') que finaliza todos los literales de cadena
- Esta inicialización es equivalente a las siguientes:

```
char letras[] = {'a','b','c','\0'};

char letras[4] = "abc";
```

2. Cadenas en C. Inicialización

- Una forma de dar valor a una cadena es asignándole el valor de otra. Para asignar una cadena a otra, se utiliza la función `strcpy ()`.

  Su prototipo es:

```
char *strcpy ( char *destino, const char *origen );
```

- La función strcpy() copia los caracteres de la cadena fuente a la cadena destino. Incluyendo el '\0'.

- La función supone que la cadena destino tiene espacio suficiente para contener toda la cadena fuente.

2. Cadenas en C. Inicialización

- Un ejemplo de strcpy.

```
/* Se inicializa la variable origen con el valor deseado */
char *origen = "Hello";

/* El puntero destino debe contener suficiente espacio para
copiar sobre la zona de memoria a la que apunta, la cadena
apuntada por origen*/
Char* destino =(char*)malloc(strlen(origen)+1);

strcpy (destino, origen);
```

NOTA: La función `strlen(char*)` devuelve la longitud de la cadena que se pasa como parámetro excluyendo el carácter '\0'.

2. Cadenas en C. Repaso sobre arrays y punteros

- Existe una equivalencia entre arrays y punteros. Cuando se define un array se están haciendo varias cosas a la vez:
  - Se define un puntero del mismo tipo que los elementos del array.
  - Se reserva memoria para todos los elementos del array. Los elementos de un array se almacenan internamente en la memoria del ordenador en posiciones consecutivas.
  - Se inicializa el puntero de modo que apunte al primer elemento del array.

- Las diferencias entre un array y un puntero son dos:
  - Que el identificador de un array se comporta como un puntero constante, es decir, no se puede hacer que apunte a otra dirección de memoria.
  - Que el compilador asocia, de forma automática, una zona de memoria para los elementos del array, cosa que no hace para los elementos apuntados por un puntero corriente.

2. Cadenas en C. Repaso sobre arrays y punteros

Ejemplo 1:

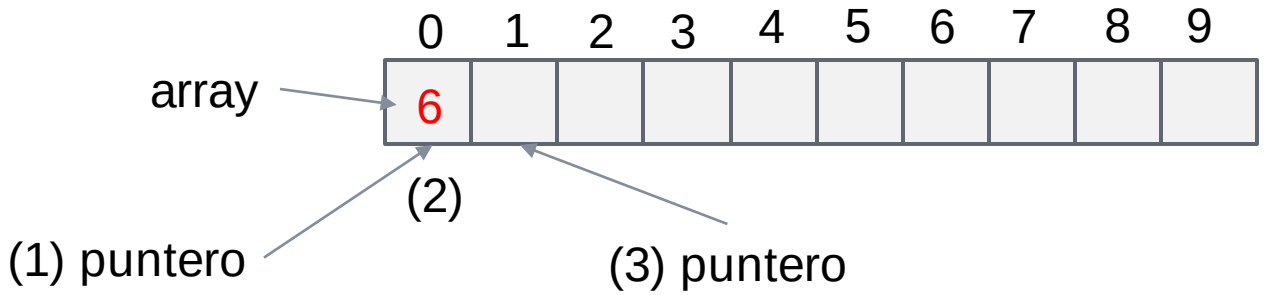
```
int array[10];
int *puntero = NULL;

array[0]=5;
puntero = array;   /* (1) Equivale a puntero = &array[0];
                      esto se lee como "dirección del primer
                      elemento de array" */
(*puntero)++;      /* (2) Equivale a array[0]++; (2) */
puntero++;         /* (3) puntero equivale a asignar a puntero el
                      valor &array[1] */
*(puntero +2)=5;   /* (4) lo apuntado por puntero + 2 es igual a
                      5, es decir array[3] = 5; */
```

(*puntero) es lo apuntado por la variable puntero, es decir, es el valor apuntado por dicho puntero

El nombre de un array es la dirección del primer elemento del array
x [i] es equivalente a *(x+i)

2. Cadenas en C. Repaso sobre arrays y punteros

- ¿Qué hace cada una de estas instrucciones?:
  - En (1): `puntero = array;`
    se asigna a puntero la dirección del array, o más exactamente, la dirección del primer elemento del array vector. `
  - En (2): `(*puntero)++;`
    se incrementa el contenido de la memoria apuntada por puntero, que es array[0].
  - En (3): `puntero++;`
    se incrementa el puntero, esto significa que apuntará a la posición de memoria del siguiente elemento int, y no a la siguiente posición de memoria. Es decir, el puntero no se incrementará en una unidad, como tal vez sería lógico esperar, sino en la longitud de un int, ya que puntero apunta a un objeto de tipo int.



2. Cadenas en C. Repaso sobre arrays y punteros

Ejemplo 2:

```
int array[10]; // --> se encuentra en la dirección 0x002, apunta a 0x016
int* puntero;  // --> se encuentra en la dirección 0x006

puntero=array; // --> la dirección 0x006 contiene 0x016
puntero++;     // --> la dirección 0x016 contiene 0x01A (avanza en sizeof(int)
               //   bytes)
*puntero=5;    // -->  la dirección 0x1A contiene 5
```

Probar a ejecutar el código anterior con los siguientes "printf". ¿Qué se mostrará en cada caso?

```
printf("%p", puntero)
printf("%d", *puntero)
printf("%p", &puntero)
```

```
printf("%p", array)
printf("%d", array[0])
printf("%p", &array)
printf("%p", &(array[1]))
```

2. Cadenas en C. Repaso sobre arrays y punteros

Este es el código:

```
int array[10]; // --> se encuentra en la dirección 0x002, apunta a 0x016
int* puntero;  // --> se encuentra en la dirección 0x006

puntero=array; // --> la dirección 0x006 contiene 0x016
puntero++;     // --> la dirección 0x016 contiene 0x01A (avanza en sizeof(int)
               //   bytes)
*puntero=5;    // -->  la dirección 0x1A contiene 5
```

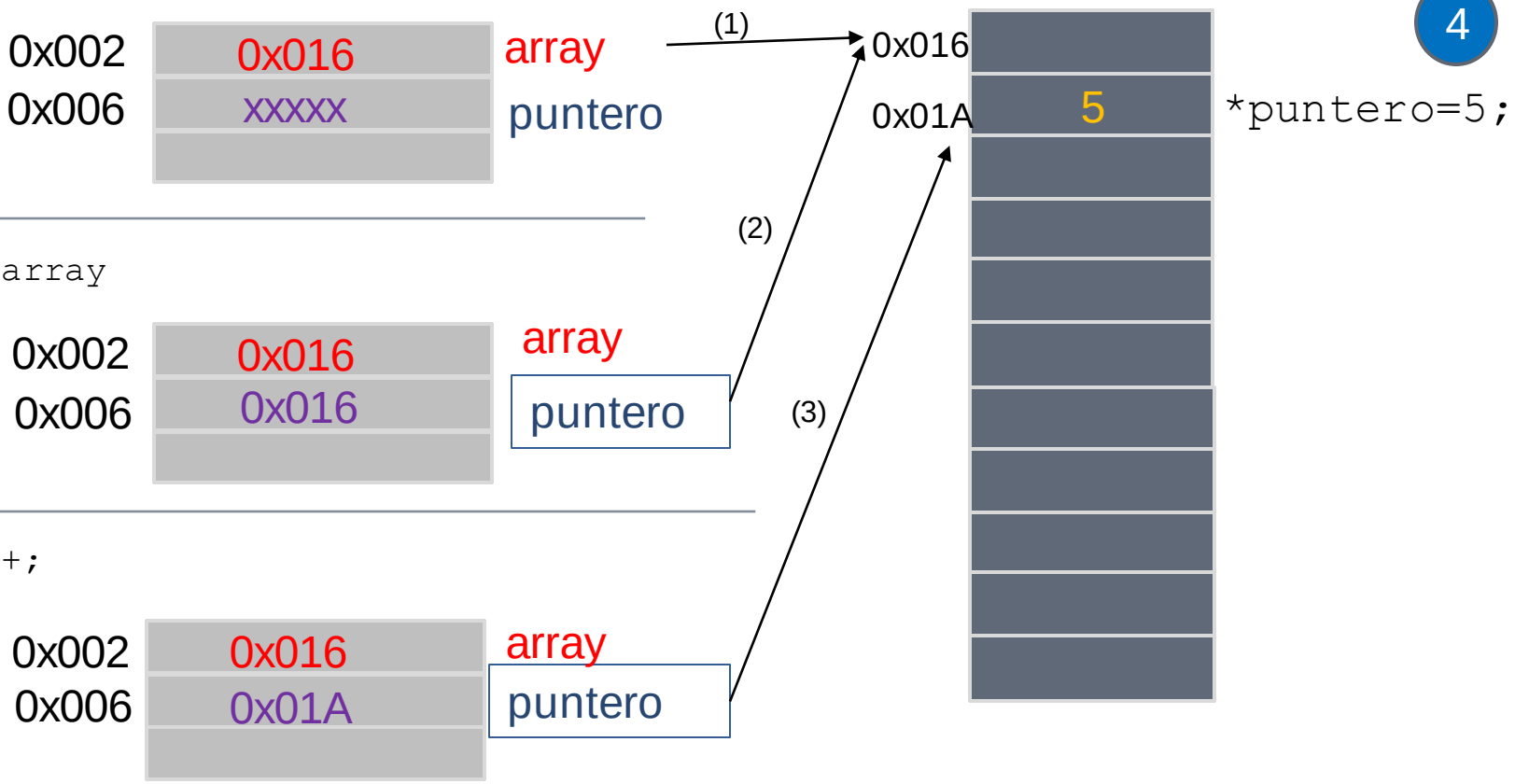
Este será el resultado:

```
printf("%h", puntero)  => 0x01A
printf("%h", *puntero) => 0x005
printf("%h", &puntero) => 0x006
```

```
printf("%h", array) => 0x016
printf("%h", array[1]) => 0x005
printf("%h", &array)   => 0x002
printf("%h",&(array[1])) => 0x01A
```

2. Cadenas en C. Repaso sobre arrays y punteros

1
```
int array[10];
int *puntero
```



2. Cadenas en C. Errores comunes al inicializar un char* con cadenas estáticas

Ejemplo 3:

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

const char* inicializaCadenaHolaMundo()
{
    return "Hola mundo"; // return de un puntero constante (se
devolverá                                   // siempre el mismo puntero!)

}
int main(int argc, char** argv)
{
    char* cadena =(char*)malloc(sizeof(char)*11);
    cadena [0]='\0';
    cadena=strcpy(cadena,"hola mundo\0"); //Inicialización correcta
    cadena = inicializaCadenaHolaMundo(); //Inicialización incorrecta
    printf("%s\n",cadena);
    return 0;
}
```

3. Lectura de cadenas

¿Qué opciones conocéis para realizar una lectura de cadenas?

3.1 Lectura de cadenas `scanf()`

- Cuando se utiliza la consola, el programa no lee directamente el texto tal cual lo introduce el usuario, sino que éste se almacena en una tabla intermedia que llamaremos **buffer**. Cada vez que el usuario pulsa el retorno de carro, se llena el buffer con la línea introducida (incluyendo el carácter '\n') y el texto introducido es leido por el programa.

- La lectura usual de datos se realiza con la función scanf()

- Cuando se aplica a datos cadena el código **de formato es %s (string).**

- **La función da por terminada la cadena cuando encuentra un espacio en blanco y el fin de línea. Los caracteres leídos se guardan en el array añadiendo el carácter nulo al final automáticamente.**

- Sin embargo "%s" **plantea dos problemas**:
  - Leerá caracteres hasta encontrarse con el primer espacio en blanco y ahí se detendrá (dejando el resto de los caracteres que hubiera tecleado el usuario sin leer, a la espera del siguiente scanf()). Esto no es normalmente lo que se desea, ya que sólo permitiría leer una palabra si el usuario escribe varias separadas por espacios.

  - Es posible que el usuario escriba más caracteres de los que podemos guardar en la variable texto. scanf() no comprueba los límites de esa variable (en realidad no puede hacerlo, aunque quisiera, porque todo lo que recibe es la dirección donde comienza el array texto, pero no su tamaño). Si el usuario escribiera más caracteres (sin espacios) de los que caben, el resto sobre escribirían otras partes de la memoria del programa, con el riesgo de seguridad que ello conlleva (ataques por buffer overrun).

3.1 Lectura de cadenas `scanf()`

- La solución pasa por utilizar scanf con formato de cadena:

```
char texto[10000];
scanf("%10000[^\n]s", texto);
getchar();
```

- Esta cadena de formato sigue esperando un string, por la “s” del final, pero:
  - El número (10000) sería el máximo de caracteres a leer. Eso evita el posible buffer overrun.
  - El [^\n] indica la categoría de caracteres a admitir, y es una expresión regular que significa "todo lo que no sea el carácter \n".

3.1 Lectura de cadenas `scanf()`

- Por tanto, esa lectura de cadena con formato leería una línea completa, con espacios y todo, deteniéndose en cuanto encuentre un \n, o cuando haya leído 10000 caracteres (lo que ocurra antes).

- **El \n queda sin leer**, a la espera de la próxima instrucción que lea algo de la entrada estándar.

- **Es por ese \n que se debe hacer luego un getchar(),** para "consumirlo", pues de lo contrario sería encontrado por el próximo scanf() que se ejecutara, lo que le confundiría y consideraría que la entrada es una línea en blanco.

3.2 Lectura de cadenas `getchar()`

- La función `getchar()` se utiliza para leer carácter a carácter.
- La llamada a `getchar()` devuelve el carácter siguiente del flujo de entrada de la entrada estándar `stdin`.
- Se encuentra en la librería <stdio.h> y su prototipo es:

```
int getchar(void);
```

- En caso de error, o de encontrar el fin de archivo devuelve `EOF` (macro definida en `stdio.h`). Su valor es típicamente -1 y representa que no hay más datos de entrada para leer del stream indicado.

```
#include <stdio.h>

int main () {
   char c;

   printf("Enter character: ");
   c = getchar();

   printf("Character entered: ");
   putchar(c);

   return(0);
}
```

Pese a que `getchar()` y `getc()` devuelven un int, el resultado puede almacenarse en un char. Esto es posible porque, internamente, las variables de tipo char se almacenan con su correspondencia numérica.

3.3 Lectura de cadenas `getc()`

- La función `getc()` proporciona el siguiente carácter del flujo (stream) especificado y avanza una posición.
- Se encuentra en la librería <stdio.h> y su prototipo es:

```
int getc(FILE *stream);
```

- Retorna el caracter leído como un unsigned char convertido (cast) a un int o EOF si es fin de fichero o error.

```
#include<stdio.h>
int main () {
   char c;

   printf("Enter character: ");
   c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);
   return(0);
}
```

La diferencia entre getc() and getchar() es que getc() puede leer de cualquier stream de entrada, mientras que getchar() lee de la entrada estándar. Por tanto, getchar() es equivalente a getc(stdin).

3.4 Escritura de cadenas `putchar()` `putc()`

- La función `putchar()` se utiliza para escribir en la salida stdout carácter a carácter.

```
int putchar(int char);
```

- El carácter que se escribe es el trasmitido como argumento.

- La función `putc()` escribe un carácter trasmitido como argumento char en el flujo (stream) especificado y avanza la posición del indicador para el flujo (stream).

```
int putc(int char, FILE *stream);
```

putchar(c) es equivalente a putc(c, stdout).

3.5 Lectura de cadenas `gets()` y `fgets()`

- Mientras que scanf() puede leer distintos tipos de datos según el formato especificado, `gets ()` y `fgets()` solo leen cadenas de caracteres introducidas por el usuario

- La función `gets()`, **permite leer una cadena completa incluyendo cualquier espacio en blanco, hasta el carácter fin de línea,** es decir, lee todo lo que haya en el buffer hasta que se encuentra un \n o el EOF.

- La función `gets()` asigna la cadena leída al argumento pasado a la función, que será un array de caracteres o un puntero (char *) con un número de elementos suficiente para guardar la cadena leída. **La función añade el carácter de terminación de cadena ('\0')**

- Si todo va bien devuelve la cadena y si ha habido un error en la lectura de la cadena, devuelve NULL.

- `gets()` es una función insegura ya que no se puede controlar el número de caracteres que introduce el usuario pudiendo ocurrir que se copien en la cadena más caracteres que los permitidos por su tamaño máximo. La alternativa segura a `gets()` es `fgets()`.

3.5 Lectura de cadenas `gets()` y `fgets()`

- El prototipo de la función es:

```
char *gets(char *cadena);
```

- Esta función lee caracteres desde el stream apuntado por **stream stdin**, y los almacena en el array apuntado por **cadena**, hasta que se encuentre un final de fichero (EOF) o un carácter de línea nueva.

- La función devuelve NULL si se produce algún fallo.

- Cualquier carácter de nueva línea es descartado, y en su lugar se escribe un carácter nulo inmediatamente después del último carácter leído en el array.

```
#include <stdio.h>

int main () {
   char str[50];

   printf("Enter a string : ");
   gets(str);

   printf("You entered: %s", str);

   return(0);
}
```

3.5 Lectura de cadenas `gets()` y `fgets()`

- La alternativa segura de gets() es **`fgets()`** que sí permite establecer el máximo de caracteres que pueden leerse.

- El prototipo de la función es:

```
char *fgets(char *cadena, int n, FILE *stream);
```

- Esta función lee como máximo un carácter menos que el número de caracteres indicado por el parámetro "n", es decir, lee n-1 caracteres desde el stream apuntado por stream y deja el resultado en el array apuntado por cadena. El carácter nulo se escribe inmediatamente después del último carácter leído en el array.

- Si no ha llegado a n-1 pero llega un carácter de retorno de línea (\n) para de leer y devuelve la cadena con dicho carácter incluido.

- La función fgets retorna el puntero "cadena" si se ejecuta con éxito. Si encuentra el final del fichero o se produce un error, devuelve NULL.

`fgets` es más segura que `gets`, pero tiene un par de inconvenientes;

- Si se han introducido más caracteres de los que se han podido leer, dichos caracteres se quedan en el buffer de entrada (en stdin), por lo que deben ser eliminados para que no sean leídos en el próximo `fgets` (siempre que no se desean leer).
- `Si fgets` ha leido el carácter \n, éste se incluye al final, así que será necesario eliminarlo.

Ejemplo lectura de cadenas: scanf() y gets()

Ejemplo 4:

- Supongamos que el usuario introduce un número y retorno de carro, después otro número y retorno de carro y por último un nombre y retorno de carro.
  - "33\n55\nJuan\n"
- La secuencia de lectura sería la siguiente:

Entrada	Buffer antes	Instrucción	Buffer después
33\n	33\n	`scanf("%d", &i1);`	\n
55\n	\n55\n	`scanf("%d", &i2);`	\n
	\n	`gets(s1);`	
Juan\n	Juan\n	`gets(s2);`	

```
int i1, i2;
char s1[30], s2[30];

scanf("%d", &i1);
scanf("%d", &i2);

gets(s1);
gets(s2);
```

Recordad: Cuando se utiliza la consola, el programa no lee directamente el texto tal cual lo introduce el usuario si no que éste se almacena en una tabla intermedia que llamaremos buffer. Cada vez que el usuario pulsa el retorno de carro, se llena el buffer con la línea introducida (incluyendo el carácter '\n').

Ejemplo lectura de cadenas: scanf() y gets()

Ejemplo 4

- El primer scanf tiene que leer del buffer hasta que encuentra un número (%d).
- En cuanto encuentra el 33 termina de leer, y deja en el buffer un '\n'.
- El segundo scanf, tras la entrada del usuario, se encuentra con \n55\n, y tiene que realizar la misma tarea que el anterior, leer un número.
- Se salta el primer '\n', y lee 55, dejando de nuevo un '\n' en el buffer.
- La función gets es más simple, y lo único que hace es leer todo lo que haya en el buffer hasta que encuentre un '\n', y lo copia en la variable correspondiente.
  - Así, el primer gets se encuentra un \n, lo consume, pero no copia nada en s1.
  - El segundo gets se encuentra Juan\n, así que lee todo lo que hay en el buffer y lo guarda en s2.
- Para permitir que Juan se copie en s1, una posibilidad es incluir una llamada a getchar() antes de emplear gets para leer s1.
  - Esta función lee un carácter del buffer, consumiendo así el '\n' que impedía rellenar s1 con Juan

Ejercicio1: cadenas_malloc

- Realizar un programa que cree un usuario con memoria dinámica y pida por pantalla los datos para rellenar los campos de dicho usuario. Los datos datos son nombre y apellidos y debe utilizarse el tamaño justo de memoria para su almacenamiento.
  - El usuario será una estructura del tipo:

```
typedef struct usuario_t{
    char* nombre;
    char* ape1;
    char* ape2;
  }usuario_t;
```

- A continutación se imprimirá dicho usuario con su nombre y apellidos en la misma línea.
- A continuación, el usuario creado se copiará sobre otra estructura del mismo tipo y se imprimirá dicha copia.
- No olvidar liberar la memoria solicitada.

4. Las funciones estándar de string.h

- La biblioteca estándar de C contiene las funciones de manipulación de cadenas utilizadas más frecuentemente.
- Cuando se utiliza una función, se puede usar un puntero a una cadena o se puede especificar el nombre de una variable array de char.

- La llamada a `strcpy()` copia la cadena `fuente` en la cadena `destino`.

```
char *strcpy (char* destina, const char* fuente);
```

- La llamada a `strcat()` añade la cadena `fuente` al final de `destino`. Concatena.

```
char* strcat (char* destino, const char* fuente);
```

Ejercicio 2: strcpy_strcat

- Se tiene un nombre y apellido por separado de un alumno. Se pide unir ambos, sin perder los datos por separado (crear una nueva variable con el resultado), y escribirlos por pantalla.

Ejercicio 2: strcpy_strcat

- Se tiene un nombre y apellido por separado de un alumno. Se pide unir ambos, sin perder los datos por separado, y escribirlos por pantalla.

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

int main () {
   char nombre[10] = "Cristina";
   char apellido[20] = "Alonso";
   char fullName[30];

   //strcpy, copia una cadena en otra
   strcpy (fullName, nombre); //lo que hay en nombre se copia a fullName
   printf("La cadena copiada) es: %s\n", fullName);

   strcat (fullName, " "); //concatena con un espacio en blanco
   strcat (fullName, apellido); //le aniado la cadena apellido
   printf("El nombre y apellido (cadena concatenada) es: %s\n", fullName);

   return(0);
}
```

4. Las funciones estándar de string.h

- La llamada a **`strchr()`** devuelve un puntero a la primera ocurrencia del carácter " ch" en la cadena "s1".
- Devuelve NULL si ch no está en s1.

```
char* strchr (char *s1, int ch);
```

- La llamada a **`strcmp()`** compara alfabéticamente la cadena `s1` con `s2` y devuelve:
  - `0 si s1 = s2`
  - `<0 si s1 < s2`
  - `>0 si s1 > s2`

```
int strcmp (const char* s1, const char* s2);
```

Ejercicio 3: strcmp

- Ordena por orden alfabético las siguientes 4 frases:
  - “La educación ayuda a la persona a aprender a ser lo que es capaz de ser”
  - “Dime y lo olvido, enséñame y lo recuerdo, involúcrame y lo aprendo”
  - “En cuestiones de cultura y de saber, sólo se pierde lo que se guarda; sólo se gana lo que se da”
  - “Para viajar lejos no hay mejor nave que un libro”
- Almacenar en un array de punteros a char cada una de las frases.
- Imprimir las 4 frases desordenadas.
- Imprimir las 4 frases ordenadas.

Ejercicio 3: strcmp

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

#define ELEMENTOS 4

int main(int argc, char *argv[])
{
            char *citas[ELEMENTOS] = {
                        "La educación ayuda a la persona a aprender a ser lo que es capaz de ser",
                        "Dime y lo olvido, enséñame y lo recuerdo, involúcrame y lo aprendo",
                        "En cuestiones de cultura y de saber, sólo se pierde lo que se guarda; sólo se gana lo que se da",
                        "Para viajar lejos no hay mejor nave que un libro"
                        };
            char *temp;
            int i, j;

            printf( "Lista desordenada:\n" );
            for( i=0; i<ELEMENTOS; i++ )
                        printf( "  %s.\n", citas[i] );

            for( i=0; i<ELEMENTOS-1; i++ )
                        for( j=i+1; j<ELEMENTOS; j++ )
                                    if ( strcmp(citas[i], citas[j]) > 0 ) {
                                                temp = citas[i];
                                                citas[i] = citas[j];
                                                citas[j] = temp;
                                    }

            printf( "Lista ordenada:\n" );
            for( i=0; i<ELEMENTOS; i++ )
                        printf( "  %s.\n", citas[i] );
}
```

4. Las funciones de string.h

- La llamada a **`strcasecmp()`** misma función que `strcmp()`, pero sin distinguir entre mayúsculas y minúsculas. Prototipo:

```
int strcasecmp (const char* s1, const char* s2);
```

- La llamada a **`memcpy()`** reemplaza los primeros n bytes de `*s1` con los primeros `n` bytes de `*s2`. Devuelve `s1`. Prototipo:

```
void* memcpy(void* s1, const void* s2, size_t n);
```

Si se produce un error devuelve NULL. Útil para copias de arrays grandes o arrayes de estructuras.

Ejercicio 4: memcpy

- Copiar los 5 primeros caracteres de una cadena (array de caracteres) en un bloque (en otro array) a partir de una dirección de memoria definida por un puntero, esta dirección será una dirección elegida arbitrariamente

Ejercicio 4: memcpy

- Copiar los 5 primeros caracteres de una cadena en un bloque a partir de una dirección de memoria definida por un puntero

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

int main()
{
   char a[8] = "abcdefg";
   char *ptr;
   int i;
   // Hay que reservar espacio para el bloque apuntado por ptr
   // Si no habria core
   ptr = (char *) malloc (strlen(a)+1);

  memcpy( ptr, a, 5 );

  for( i=0; i<8; i++ )
     printf( "a[%d]=%c ", i, a[i] );

  printf( "\n" );

  for( i=0; i<5; i++ )
     printf( "ptr[%d]=%c ", i, ptr[i] );
   printf( "\n" );

   return 0;
}
```

4. Las funciones de string.h

- La llamada a **`strstr()`** busca la cadena `s2` en la cadena `s1` y devuelve un puntero a los caracteres donde se encuentra `s2`. Es decir, busca un substring dentro de otro. Si no lo encuentra devuelve NULL. Util para trocear cadenas.

```
char *strstr (const char* s1, const char* s2);
```

- La llamada a **`strtok()`** analiza o parsea la cadena `s1` en tokens (componentes léxicos), dichos tokens delimitados por cadena `s2` .
- **La llamada inicial a `strtok(s1,s2)` devuelve la dirección del primer token** y sitúa el puntero al final del token. Si queremos seguir obtenindo los tokens invocaremos strtok sustituyendo s1 por NULL. Es decir:
  - **Después de la llamada inicial, cada llamada sucesiva se invoca `strtok(NULL,s2)` devuelve un puntero al siguiente token encontrado en `s1`**. Si se volviera a hacer la misma llamada utilizando s1 volvería a tomar el primer token.
  - **Estas llamadas cambian la cadena `s1` reemplazando el carácter `s2` por '\0'.**

```
char *strtok (char* s1, const char* s2);
```

Ejercicio 5: strtok

- Se ha recogido en una única cadena una línea de un fichero csv con 3 campos (nombre, apellido y edad). Separar los tres campos en cadenas diferentes

Ejercicio 5: strtok

- Se ha recogido en una cadena una línea de un fichero csv con 3 campos (nombre, apellido y edad). Separar los tres campos en cadenas diferentes

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

#define SEPARADOR ";"

int main () {
   char linea_csv[50];
   char nombre[20];
   char apellido[20];
   int  edad;
   char* p_token;

   //strcpy, copia una cadena en otra
   strcpy (linea_csv, "Cristina;Alonso;23"); //lo que hay en nombre se copia a fullName
   printf("La linea del csv) es: %s\n", lineacsv);

   p_token = strtok(linea_csv, SEPARADOR); //devuelve el primer token
   strcpy (nombre, p_token);

   p_token = strtok(NULL, SEPARADOR); //devuelve el segundo token
   strcpy (apellido, p_token);

   p_token = strtok(NULL, SEPARADOR); //devuelve el tercer token
   edad = atoi (p_token);

   printf("El nombre, apellido y la edad son: %s, %s, %d\n", nombre, apellido, &edad);

   return(0);
}
```

4. Las funciones de string.h

- La llamada a `strncpy()` copia los `n` caracteres de la cadena `fuente` en la cadena `destino` y devuelve el valor de la cadena destino.

```
char *strncpy (char* destino, const char* fuente, size_t num);
```

- La llamada a `strcspn()` devuelve la longitud de la subcadena más larga de `s1` comenzando desde el principio (s1[0]) y que no contiene ninguno de los caracteres de la cadena `s2`.

```
size_t strcspn (const char* s1, const char* s2);
```

Ejercicio 6: Cadenas y punteros

- Definir un array de cadenas de caracteres para poder leer un texto, compuesto por un máximo de 80 líneas. Escribir una función para leer dicho texto. Las líneas de dicho texto serán introducidas por el usuario.
- Una de las ventajas de trabajar con punteros es poder reservar memoria dinámicamente, es decir, en tiempo de ejecución, para las variables necesarias. En este caso si se reservase un array entero de 60 posiciones por línea, se desperdiciaría todo el espacio sobrante de las cadenas cuya longitud no llegase a 59 caracteres.
  - Manejando punteros y reserva dinámica de memoria hacer:
    - leer cada línea en un buffer temporal, para averiguar su longitud final
    - luego reservar tanta memoria como sea precisa para la nueva línea
    - hacer que el puntero de la línea apunte a la primera posición de la memoria recién asignada y
    - Por último, copiar la línea de texto desde el buffer a la nueva zona apuntada por el puntero de la línea.

Ejercicio 6: Cadenas y Punteros

Utilizar como prototipo de la función que debe invcarse desde el main:

```
char** leerTexto (int nlineas);
```

El main debe invocar esa función para leer nlineas y debe imprmit todas las líneas del texto leido.

Ejercicio 7: Cadenas y punteros

- Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.
  - Como se pide que la función devuelva tres valores y una función en C sólo puede devolver un valor, se opta por hacer que la función tenga tres parámetros de salida. Con la semántica del paso por referencia, se pasan tres punteros a las variables que la función puede modificar para dejar los tres resultados que se piden, que no son más que tres contadores.
  - Para averiguar el tipo de carácter, recorrer la cadena por medio de un puntero auxiliar y comparar su código ASCII con el de los números y las letras

Ejercicio 7: Cadenas y punteros

```
void cuentaLetras (const char* cadena, int *vocales, int *consonantes, int *digitos)
{
     // Hacer la función
}


int main ()
{

    int vocales=0;
    int consonantes=0;
    int digitos=0;

   const char* cadena="Hola123";

    cuentaLetras(cadena, &vocales, &consonantes, &digitos);

    printf ("Número de vocales: %d, consonantes: %d y digitos: %d\n", vocales, consonantes, digitos);


    return 0;
}
```