

(Versión de fecha 30 de noviembre de 2024)

La teoría de este documento sirve para entender la primera parte de la práctica 4

1 TIPOS DE DATOS PERSONALIZADOS PARA CADENAS DE CARACTERES

Es habitual declarar tipos de datos personalizados para cadenas de caracteres de determinado tamaño. El tamaño se suele declarar como una constante con la cláusula *#define* y el tipo de datos con *typedef*.

Observa el siguiente ejemplo:

Ejemplo 1 Tipo personalizado para cadenas de 40 caracteres (39+1)

```
#include <stdio.h>

#define MAX_TXT 40

typedef char tTexto[MAX_TXT];

int main() {
    tTexto cadena_1 = "Prueba de cadena";
    printf("%s\n", cadena_1);
    return 0;
}
```

Primero se declara una constante de nombre *MAX_TXT*, de valor 40. Luego se define un tipo de datos llamado *tTexto* para las cadenas de caracteres de tamaño 40: 39 caracteres efectivos más el '\0'.

En el programa se declara la variable *cadena_1* del tipo de datos *tTexto*, se le asigna un valor y se imprime en pantalla.

A todos los efectos, el nuevo tipo de datos es una cadena de caracteres de tamaño 40. Se puede pasar como parámetro de funciones que esperen recibir una cadena de caracteres y, en ese sentido, se puede usar en las funciones habituales para manejar cadenas de la librería *string.h*. El nombre del nuevo tipo de datos es simplemente un *alias* o un *seudónimo* para referirse a las cadenas de caracteres de tamaño 40.

El código del Ejemplo 2 utiliza el mismo tipo de datos del ejemplo anterior. En esta ocasión, le pide la cadena al usuario con *fgets()*, concatena la cadena "ABC" utilizando la función *strcat()* de la librería *string.h* y muestra el resultado por pantalla.

Observa la salida del programa en la figura de la derecha del Ejemplo 2: las letras ABC aparecen en distinta línea que la cadena que tecleó el usuario. ¿Qué ha pasado? Bueno, como hemos visto en otras ocasiones, al leer una cadena del terminal con la función *fgets()*, la última tecla es *INTRO*, el '\n', y ese cambio de línea se queda como último carácter de la cadena leída.

Por tanto, siempre que leamos una cadena de caracteres del terminal con la función *fgets()*, tenemos que *limpiar* el '\n' residual.

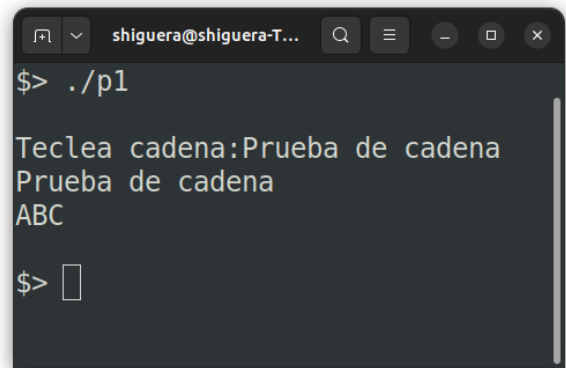
Ejemplo 2 Entrada de cadenas: '\n' residual

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

#define MAX_TXT 40

typedef char tTexto[MAX_TXT];

int main() {
    tTexto cadena_1;
    printf("Teclea cadena:");
    fgets(cadena_1, MAX_TXT, stdin);
    strcat(cadena_1, "ABC");
    printf("%s\n", cadena_1);
    return 0;
}
```



En el Apartado 5 del documento *P10_Pizarras_2024_11_19.pdf* mostrábamos una función llamada *limpia()* que recibía como parámetro una cadena de caracteres y, si el último carácter era un '\n', lo sustituía por un '\0'. El código de la función era el siguiente:

```
void limpia(char cad[]) {
    int ultimo = strlen(cad)-1;
    if(cad[ultimo] == '\n') {
        cad[ultimo]='\0';
    }
}
```

Podemos incorporar esa función a nuestro programa del Ejemplo 2 y, tras leer la cadena que teclea el usuario, llamar a la función *limpia()* para eliminar el '\n' residual. El código resultante y la salida por pantalla se muestran en el Ejemplo 3.

Ejemplo 3 Lectura y limpieza de cadena desde el terminal

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

#define MAX_TXT 40

typedef char tTexto[MAX_TXT];

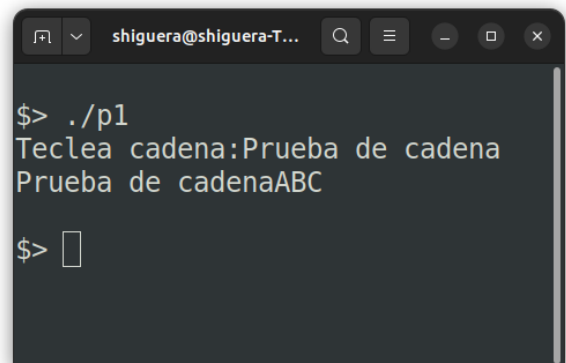
void limpia(char cad[]);

int main() {
    tTexto cadena_1;
    printf("Teclea cadena:");
    fgets(cadena_1, MAX_TXT, stdin);

    limpia(cadena_1);

    strcat(cadena_1, "ABC");
    printf("%s\n", cadena_1);
    return 0;
}

void limpia(char cad[]) {
    int ultimo = strlen(cad)-1;
    if(cad[ultimo] == '\n') {
        cad[ultimo]='\0';
    }
}
```



Observa en la salida por pantalla que el '\n' se ha eliminado correctamente.

2 INICIALIZACIÓN DE ARRAYS

Los arrays se pueden inicializar de varias maneras:

- **Con basura:** si solo se hace la declaración del array, sin asignar ningún valor a sus elementos, se reserva sitio en memoria para sus valores. El valor inicial será el que hubiera en ese momento en esas posiciones de memoria, que se le suele llamar *basura*. Es obligado indicar la dimensión del array en la declaración. Ejemplo:

```
int numeros[3]; // Se inicializa el array con basura
```

- **Con ceros:** si al declarar el array se le asignan unas llaves vacías o unas llaves con un cero, el array se inicializa con valores cero de sus elementos. Es obligado indicar la dimensión del array en la declaración. El tipo de elementos del array puede ser cualquiera, no solo tipos primitivos como *int* o *double*, podrían ser otros arrays o estructuras. Ejemplos:

```
int numeros[3] = {}; // Se inicializa el array con ceros
```

```
int numeros[3] = {0}; // Se inicializa el array con ceros
```

- **Con valores concretos:** en la misma línea de la declaración, se pueden asignar valores concretos entre llaves. En este caso, no es necesario indicar la dimensión del array, se pueden dejar los corchetes vacíos. Ejemplo:

```
int numeros[] = {1, 2, 3}; // Se inicializa el array con valores
```

Como se ha puesto de manifiesto, al declarar un array, queda inicializado, de una forma u otra. El lenguaje C no permite inicializar un array dos veces. Por ejemplo, el siguiente código daría error:

```
int numeros[3]; // Se inicializa el array
numeros = {1, 2, 3}; // ¡ERROR!
```

Una vez que se ha inicializado un array, si se quiere asignar valores a los elementos, hay que hacerlo accediendo individualmente a los mismos. El ejemplo siguiente sí que sería correcto:

```
int numeros[3]; // Se inicializa el array
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
```

3 INICIALIZACIÓN DE ESTRUCTURAS

Las variables del tipo estructura también se pueden inicializar de diferentes maneras:

- **Con basura:** si se declara una variable de tipo estructura y no se asigna ningún valor, los campos tendrán inicialmente basura.
- **Con ceros:** al igual que sucedía con los arrays, si se asignan unas llaves vacías o con un cero entre las llaves, los campos de la estructura se inicializan a cero.
- **Con valores entre llaves:** de manera similar a la de los arrays, se puede dar valor a los campos, poniendo los valores entre llaves, separados por comas y en el mismo orden en el que están definidos en la estructura.
- **Con copia:** en las estructuras se puede utilizar el operador de asignación, cosa que no se puede hacer con los arrays. Por ello, se puede crear una variable de un tipo estructura como copia de otra ya existente.

La forma general de inicializar una estructura es hacerlo en dos pasos: primero declarar la variable y luego asignar valores concretos a los campos. Se puede hacer inicializando con basura o inicializando con ceros. Este procedimiento permite declarar la variable y luego, en un momento posterior, asignar valor a los campos. Es útil cuando el valor de los campos no se conoce en tiempo de compilación, sino en tiempo de ejecución.

En el siguiente ejemplo se inicializa la estructura con basura y luego se asignan valores a los campos:

Ejemplo 4 Inicialización de una estructura

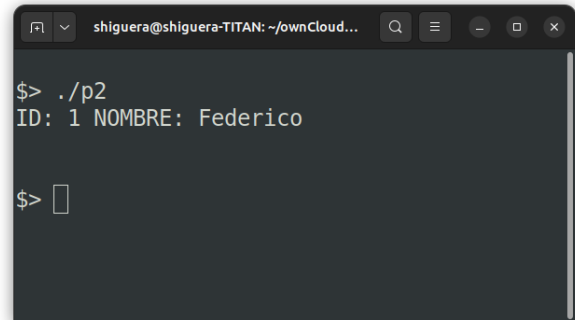
```
#include <stdio.h>
#include <string.h>

typedef struct {
    int id;
    char nombre[40];
} Autor;

int main() {
    Autor autor1;

    autor1.id = 1;
    strcpy(author1.nombre, "Federico");

    printf("ID: %d NOMBRE: %s\n\n",
        autor1.id, autor1.nombre);
}
```



```
shiguera@shiguera-TITAN: ~/ownCloud...
$> ./p2
ID: 1 NOMBRE: Federico
$> 
```

También se podrían haber inicializado a cero los campos. El siguiente ejemplo es una ampliación del anterior en el que se inicializan a cero los campos y se muestran, antes y después de asignarles valor. En el ejemplo anterior no se pueden enseñar los campos antes de asignar valor, pues las cadenas de caracteres inicializadas con basura no tienen el '\0' final.

Ejemplo 5 Inicializar una estructura con ceros

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

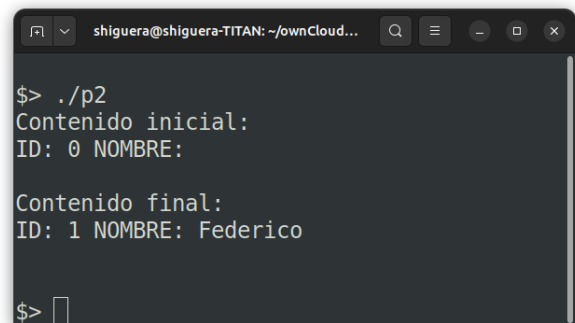
typedef struct {
    int id;
    char nombre[40];
} Autor;

int main() {
    // Se inicializa con basura
    Autor autor1 = {0};

    // Contenido inicial de la estructura: ceros
    printf("Contenido inicial:\n");
    printf("ID: %d NOMBRE: %s\n\n",
        autor1.id, autor1.nombre);

    // Se asigna valor a los campos
    autor1.id = 1;
    strcpy(author1.nombre, "Federico");

    // Se muestra el contenido final de la estructura
    printf("Contenido final:\n");
    printf("ID: %d NOMBRE: %s\n\n",
        autor1.id, autor1.nombre);
}
```



```
shiguera@shiguera-TITAN: ~/ownCloud...
$> ./p2
Contenido inicial:
ID: 0 NOMBRE: 
Contenido final:
ID: 1 NOMBRE: Federico
$> 
```

Si el valor de los campos se conoce en el momento de codificar el programa, se puede usar un método simplificado, similar al que se utiliza con los arrays. Consiste en poner entre llaves y separados por comas los valores que se quieren asignar a los campos. Hay que poner los valores en el mismo orden en el que están declarado los campos en la definición de la estructura.

El siguiente ejemplo utiliza esta técnica:

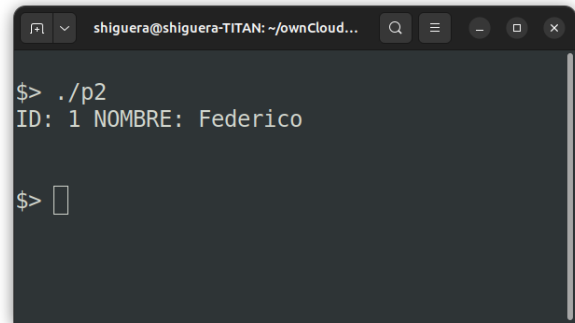
Ejemplo 6 Inicialización de una estructura usando llaves

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

typedef struct {
    int id;
    char nombre[40];
} Autor;

int main() {
    Autor autor1 = {1, "Federico"};

    printf("ID: %d NOMBRE: %s\n\n",
        autor1.id, autor1.nombre);
}
```



```
shiguera@shiguera-TITAN: ~/ownCloud...
$> ./p2
ID: 1 NOMBRE: Federico
$> 
```

4 ESTRUCTURAS ANIDADAS

Los campos de las estructuras pueden ser de cualquier tipo de datos. Por ejemplo, es posible que los campos sean arrays u otras estructuras. En el siguiente ejemplo, la estructura *Libro* tiene un campo que es a su vez una estructura del tipo *Autor*. Observa la forma en la que se ha inicializado la estructura y como se accede a los campos del *Autor*, usando doble punto:

Ejemplo 7 Estructuras anidadas

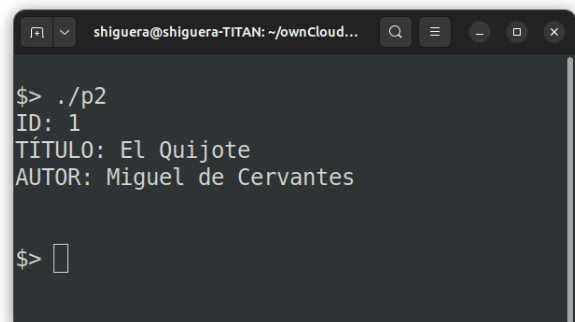
```
#include <stdio.h>
#include <string.h>

typedef struct {
    char nombre[40];
    char apellidos[40];
} Autor;

typedef struct {
    int id;
    char titulo[40];
    Autor autor; // Estructura anidada
} Libro;

int main() {
    Libro libro1 = {1, "El Quijote",
        {"Miguel", "de Cervantes"}};

    printf("ID: %d\n", libro1.id);
    printf("TÍTULO: %s\n", libro1.titulo);
    printf("AUTOR: %s %s\n\n",
        libro1.autor.nombre, libro1.autor.apellidos);
}
```



```
shiguera@shiguera-TITAN: ~/ownCloud...
$> ./p2
ID: 1
TÍTULO: El Quijote
AUTOR: Miguel de Cervantes
$> 
```

5 ARRAYS DE ESTRUCTURAS

Los elementos de un array pueden ser variables de algún tipo de estructura. Un buen ejemplo lo tienes en la estructura *tProgramadoresAplicacion* de la primera parte de la Práctica 4:

- Se definen unos tipos de datos personalizados para cadenas de caracteres: *tId* y *tNombre*.
- Para establecer el tamaño de estas cadenas de caracteres, se utilizan unas constantes que se han definido antes utilizando cláusulas *#define*: *MAX_ID* y *MAX_NOMBRES*.
- Se define una estructura llamada *tDatosProgramador* para guardar los datos de cada programador.

- Se define un tipo de datos llamado *tListaProgramadores* que es un array cuyos elementos son estructuras del tipo *tDatosProgramador*.
- Finalmente, se define una estructura llamada *tProgramadoresAplicacion* con dos campos: una *tListaProgramadores* y un entero llamado *numeroProgramadores* para saber en cada momento cuántos programadores hay registrados en la lista.

El código es el siguiente:

Ejemplo 8 Array compactado de la Práctica 4

```
#define MAX_ID 7 // Cadena 6 caracteres + \n
#define MAX_NOMBRES 31 // Cadena 30 caracteres + \n
#define MAX_PROGRAMADORES 4 // Número máximo de
                             // programadores
                             // asignados a la app

typedef char tId[MAX_ID+1];
typedef char tNombre[MAX_NOMBRES+1];

typedef struct {
    tId idProg;
    tNombre nombre;
    bool personalEmpresa;
} tDatosProgramador;

typedef tDatosProgramador
    tListaProgramadores[MAX_PROGRAMADORES];

typedef struct {
    tListaProgramadores listaProgramadores;
    int numeroProgramadores;
} tProgramadoresAplicacion;
```

