

TEMA 6.

TIPOS DE DATOS ESTRUCTURADOS

Grado en Ingeniería en Tecnologías Industriales
Programación



CONTENIDOS

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

6.1. INTRODUCCIÓN

Datos estructurados frente a datos simples

- Los datos pueden tener o no estructura:
 - **Datos simples (vistos en Tema 1)**
 - Tienen un único valor, son un único elemento
 - **Datos estructurados**
 - Tienen una estructura interna, no son un único elemento

Tipos:

- Numéricos: enteros y reales
- Caracteres
- Punteros

Tipos:

- Vectores y matrices (también llamados arrays)
- Cadenas de caracteres
- Estructuras (también llamadas registros)

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

6.2. DECLARACIÓN Y USO DE ARRAYS

Concepto de array o matriz

- Colección ordenada de objetos que comparten el nombre (identificador) y son todos de **un mismo tipo de datos**.
- Los elementos individuales del array se identifican usando un **índice** que indica una posición dentro del array y que permite acceder a ellos.
- En función de las dimensiones se pueden llamar:
 - Una dimensión = **vector** (también llamado lista)
 - Dos o más dimensiones = **matriz**
 - Dos dimensiones = tabla con filas y columnas

Concepto de array o matriz

Una dimensión = vector

• Ejemplo de vector

- Estructura de datos que permite almacenar la temperatura en Madrid a lo largo de un año:

0	1	2	3	temperatura[]	364
6.2	6.5	9.0	10.7	1.0

- Nombre del vector: temperatura
 - Todas las posiciones son del mismo tipo (int)
 - Comparten un mismo identificador: **temperatura**
 - Pero cada una tiene un valor independiente de los demás.
 - Cada elemento viene identificado por su **índice** (0,1,...,364)
 - El primer elemento de un vector en C tiene siempre índice 0
 - Utilizando el índice podemos encontrar el valor de una posición:
 - Ej: temperatura el tercer día del año

```
temperatura [2] = 9;
```

Concepto de array o matriz

• Ejemplo de matriz de dos dimensiones

- estructura de datos que nos permite almacenar la ocupación de una sala de cine (0 libre, 1 ocupado)

0	1	1	1	1
0	0	1	1	1
1	1	1	1	1
0	0	1	0	0



← Dos dimensiones

- Todos los elementos comparten el nombre del array (**sala**)
- Cada uno de ellos está caracterizado por un índice
 - En este caso, el índice representa por la fila y la columna de cada elemento.

Fila ← → *Columna*

sala [0] [0] = 0;

sala [2] [3] = 1;

- **Ejemplo de matriz con tres dimensiones**

- Fila* *Columna* *planta*
- ↓ ↓ ↙
- ```
sala [1] [0] [0] = 0;
sala [1] [0] [1] = 1;
```

Diagram illustrating a neural network structure with 5 input nodes (Fila) and 5 output nodes (Planta). The connections are represented by a grid of 0s and 1s.

| Columna | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|
| Fila 1  | 0 | 0 | 1 | 1 | 0 |
| Fila 2  | 0 | 1 | 0 | 0 | 0 |
| Fila 3  | 0 | 1 | 0 | 0 | 0 |
| Fila 4  | 0 | 0 | 1 | 0 | 0 |
| Fila 5  | 0 | 0 | 0 | 0 | 0 |

Connections (1) are shown between Fila 1 and Planta 1, 2, 3, 4, 5. Connections (1) are shown between Fila 2 and Planta 1, 2, 3, 4, 5. Connections (1) are shown between Fila 3 and Planta 1, 2, 3, 4, 5. Connections (1) are shown between Fila 4 and Planta 1, 2, 3, 4, 5. Connections (1) are shown between Fila 5 and Planta 1, 2, 3, 4, 5.

# Declaración de arrays

- Para declarar un array hay que indicar:
  - tipo de datos de los elementos que lo forman
  - nombre del array
  - número de dimensiones
  - número de elementos en cada dimensión
- de esta forma el compilador conoce el espacio de memoria que es necesario reservar
  - n variables del mismo tipo
  - se almacenan en direcciones consecutivas
- En el ejemplo anterior
  - tipo de datos de los elementos: integer
  - nombre del array: sala
  - número de dimensiones: 3
  - número de elementos en cada dimensión: 4, 5 y 2
- Declaración: `int sala [4][5][2]`

# Declaración de arrays

- Declaración de un vector (una única dimensión)

- Sigue este esquema:

tipo\_dato    nombre\_array    [tamaño];

- Ejemplos:

```
float temperatura[365];
```

```
int num[10];
```

- array (vector) de 10 números tipo int
    - a cada elemento del array se accede por su índice
    - el índice inferior es siempre 0 y el superior, en este ejemplo, es 9

|   |   |    |   |       |     |
|---|---|----|---|-------|-----|
| 5 | 7 | 15 | 1 | ..... | 250 |
|---|---|----|---|-------|-----|

Indice:    0                    1                    2                    3                    .....                    9

# Declaración de arrays

- Declaración de una matriz

**tipo\_dato nombre\_array [expresion1] [expresion2].... [expresionN];**

- Cada expresión indica el valor de cada dimensión del array.
- Ejemplo: Array de dos dimensiones para guardar una imagen de 800 por 600 puntos en blanco y negro

```
int pantalla [800] [600];
```

- Ejemplo: Array de tres dimensiones para almacenar la inicial del nombre de los espectadores de tres salas de cine con 30 filas y 15 columnas cada una.

```
char sala [30] [15] [3];
```

# Declaración de arrays

- Tipos de datos de un array
  - Los elementos de un array pueden ser de cualquier tipo, incluido otro array.
- En C una matriz de dos dimensiones puede verse como un vector cuyos elementos son a su vez vectores
  - Y lo mismo para más dimensiones
  - Ejemplo
    - `int pantalla [800] [600];`
      - Puede verse como un vector de 800 elementos, en los que cada uno de esos 800 elementos es un vector de 600 elementos
- El primer elemento de un array en C siempre tiene como índice el cero

# Asignación de valores a los elementos de un array

- Asignación de valores a un array:
  - Se asigna valor a un elemento identificado por su índice

```
sala [1][3][1]= 1;
notas[25]= 10;
pantalla[0][0]=1;
```
  - Sólo es posible asignar valor a un elemento indicando su índice
  - No es posible asignar un valor a todo el array

```
pantalla =0; //error
```

# Inicialización de un array

- Es posible declarar e inicializar una matriz en una única instrucción
  - Igual que se hace con datos simples

- Con matrices de una dimensión (vectores)

```
int numeros[5]={6,2,7,4,8};
```

- Otra opción: sin indicar el número de elementos  
Si no se especifica tamaño pero se inicializa, se adopta el número de elementos inicializados como tamaño del vector.

```
int numeros []={6,2,7,4,8};
```

- Esas dos opciones equivalen a primero declarar y luego inicializar:

```
int numeros[5];
numeros[0]=6;
numeros[1]=2;
numeros[2]=7;
numeros[3]=4;
numeros[4]=8;
```

# Inicialización de una matriz

- Con arrays de más de una dimensión:

```
int numeros [3][2]={
 {0,1},
 {10,11},
 {20,21}
};
```

- numeros puede interpretarse como un vector de tres elementos, en el que cada uno es a su vez un vector de dos elementos



# Uso de arrays

- Para obtener el valor de un elemento de un array, hay que indicar el índice correspondiente

```
numeros [4]
```

- Como índice se puede utilizar cualquier expresión, siempre que el valor resultante sea de tipo entero.

```
enteros [5*j+i]=7;
```

- El rango de los valores generados por las expresiones debe estar entre cero y el número de elementos del array - 1:

```
int miVector[4] = {0,1,2,3};
for (i=0;i<4;i++)
printf("el valor del elemento de indice %i es %i \n",
 i, miVector[i]);
```

# Uso de arrays

- Si intentamos acceder a una posición que no ha sido reservada resultará en un error

```
int i;
int numbers[]={ 4, 8, 15, 16, 23, 42}
```

```
numbers[6] = 100;
```

→ Error: se intenta acceder a la posición 6 de un array de 6 elementos, solo son válidas las posiciones de 0 a 5

```
for (i=0; i<=5; i++){
 printf("Element %i: %i \n", i numbers[i]);
}
```

Correcto. El índice *i* toma valores entre 0 y 5

```
for (i=0; i<=6; i++){
 printf("Element %i: %i \n", i numbers[i]);
}
```

Error: El índice *i* toma valores entre 0 y 6

## No se puede trabajar con un array como un bloque

- Los datos de los arrays deben procesarse elemento por elemento, en general no pueden tratarse como un bloque
- No se puede **asignar** un array completo a otro
  - Error de compilación
- No se pueden **comparar directamente** arrays completos
  - Lo que se compararía son las direcciones del primer elemento de cada array
- No se puede **imprimir directamente** un array completo
  - Lo que se imprimiría sería la dirección del primer elemento del array

# Uso de arrays

- No se puede **mostrar el valor** de todos los elementos del array como si fuera un bloque
  - Ni en C ni en la mayoría de los lenguajes
  - Se debe mostrar elemento por elemento

```
int pantalla [800] [600];
printf ("esta instrucción NO muestra el array entero %d \n", pantalla);
```

# Lectura

- Lectura de los elementos de un array

- Para leer un elemento:

```
printf ("escriba color del tercer pixel de tercera
columna");
scanf ("%d", &array[2][2]);
```

- » Para leer todo el array: lectura de los elementos de uno en uno

```
int array[4][2];
for (i=0; i<4; i++) {
 for (j=0; j<2; j++) {
 printf ("Introduzca el elemento %d %d:", i, j);
 scanf ("%d", &array[i][j]);
 }
}
```

```
Introduzca el elemento 0 0:0
Introduzca el elemento 0 1:1

Introduzca el elemento 1 0:10
Introduzca el elemento 1 1:11

Introduzca el elemento 2 0:20
Introduzca el elemento 2 1:21

Introduzca el elemento 3 0:30
Introduzca el elemento 3 1:31
```

# Escritura

- Escritura de un único elemento:

```
printf ("Se muestra el color del segundo pixel de la
tercera columna %i:", pantalla [1][2]);
```

- Escritura de todos los elementos (hay que ir de uno en uno..)

```
int array[4][2];
int i, j;
for (i=0; i<4; i++) {
 for (j=0; j<2; j++) {
 printf ("%d\t", array [i][j]);
 }
}
```

```
0 1
10 11
20 21
30 31
Presione una tecla para continuar . . .
```

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.3. PUNTEROS Y ARRAYS

# Relación entre punteros y arrays

- existe una relación muy estrecha entre arrays y punteros
  - el **nombre** de un array es una variable que contiene la dirección de memoria que almacena el primer elemento del array
    - es un **puntero** a la dirección de memoria que contiene el primer elemento del array.
  - se puede acceder al resto de elementos usando la dirección del primero (el nombre de la variable)



# Punteros y arrays

- supongamos la siguiente declaración de un vector

```
int lista[6] = {10, 7, 4, -2, 30, 6};
```

- cada elemento ocupa 2 bytes de memoria, al ser de tipo int

lista[0] lista[1] lista[2] lista[3] lista[4] lista[5]

|         |   |   |    |   |   |    |    |   |
|---------|---|---|----|---|---|----|----|---|
| Memoria | [ |   |    |   |   |    |    |   |
|         |   | ? | 10 | 7 | 4 | -2 | 30 | 6 |

Dirección de  
Memoria

1500 1502 1504 1506 1508 1510

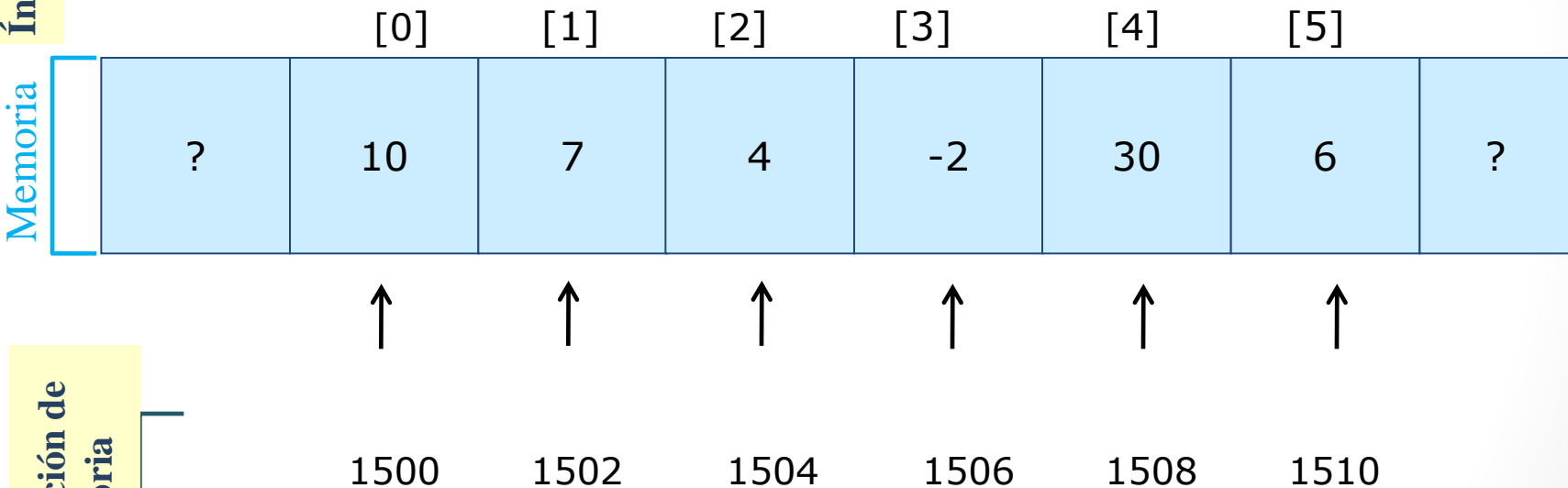
# Punteros y arrays

- Podemos acceder a los elementos del array de dos formas:
  - A. mediante su índice
  - B. mediante su dirección de memoria

Índice

Memoria

Dirección de  
Memoria



# Punteros y arrays

- A. Accediendo mediante su índice.

```
x = lista[1]; // x = 7
y = lista[0]; // y = 10
```

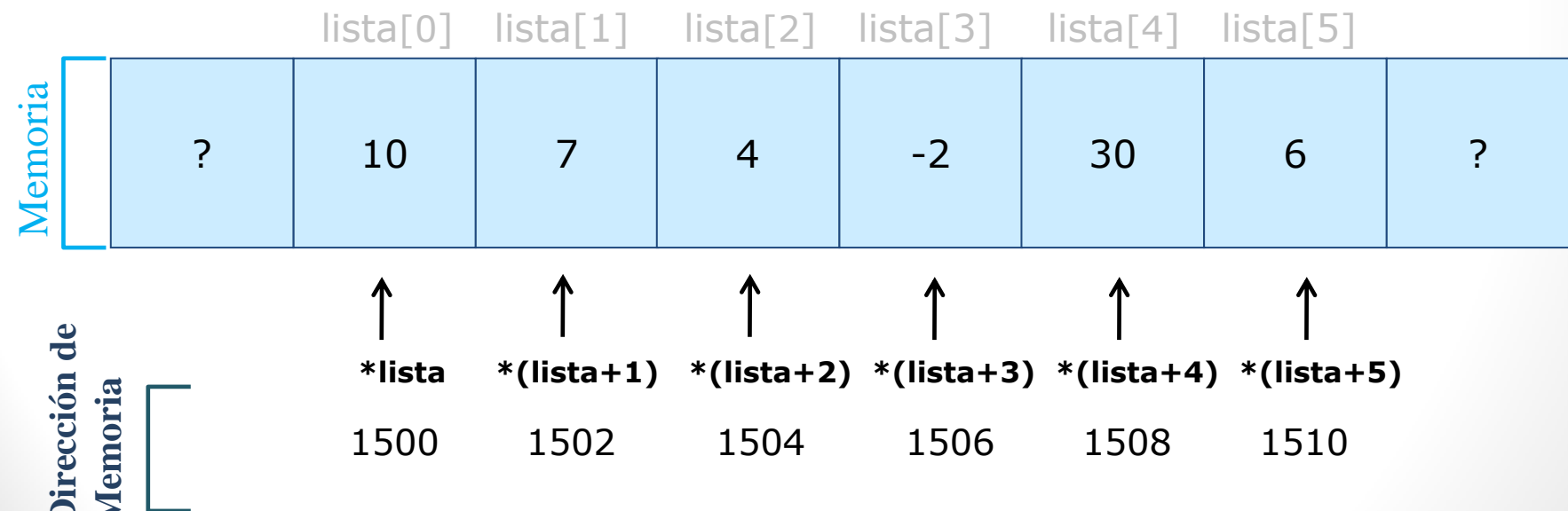
|         | lista[0] | lista[1] | lista[2] | lista[3] | lista[4] | lista[5] |   |   |
|---------|----------|----------|----------|----------|----------|----------|---|---|
| Memoria | ?        | 10       | 7        | 4        | -2       | 30       | 6 | ? |

# Punteros y arrays

- B. Accediendo mediante su dirección de memoria.

```
x = *(lista+1); // x = 7
p1 = &lista[0]; // p1 = 1500
p2 = lista; // p2 = 1500
```

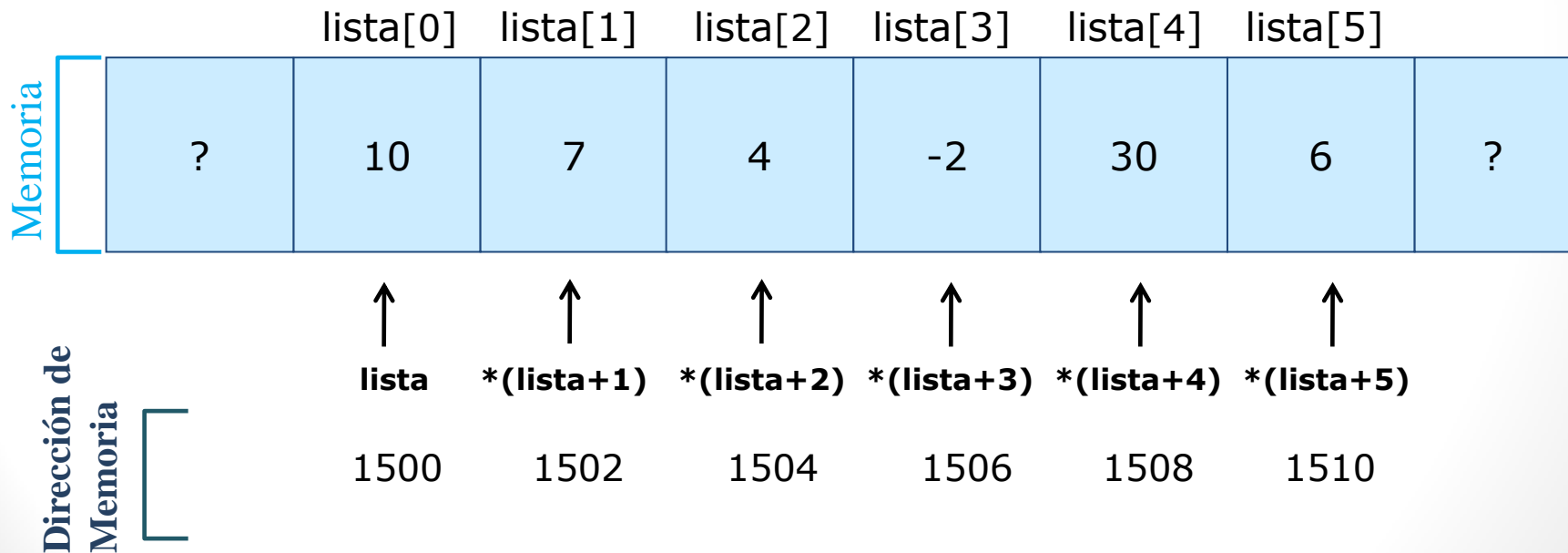
`&lista[0]` es lo mismo que `lista`



# Punteros y arrays

```
p4 = &lista[3]; // p4 = 1506
```

```
p5 = lista + 3; // p5 = 1506
```



# Punteros y arrays

- Código correspondiente al ejemplo anterior

```
int main(void) {
 int lista[6]={7, 10, 4, -2, 30, 6};

 printf("\n%d", &lista[0]);
 //resultado: 1500 (dirección de memoria)
 printf("\n%p", lista);
 //resultado: 1500 (dirección de memoria)
 printf("\n%p", lista+1);
 //resultado: 1502 (dirección de memoria)
 printf("\n%d", *(lista+1));
 //resultado:7 (valor de lista[1] o valor en la dirección 1502)
 return 0;
}
```

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.4. CADENAS DE CARACTERES

# Cadenas de caracteres

- Un **string** o cadena de caracteres es una secuencia de caracteres delimitada por comillas (")

`"que buen tiempo hace hoy"`

- En una cadena puede haber cualquier carácter del código ASCII
  - Incluyendo espacios en blanco
  - Para algunos caracteres es necesario usar **secuencias de escape**
    - La barra \ indica al ordenador que hay que tratar de forma especial lo que va a continuación
      - \n : se inserta un salto de línea, en vez de la letra n
      - \" : se insertan unas comillas dentro de la cadena, en vez de considerar que es el final de la cadena



# Cadenas de caracteres

- Una cadena se almacena en un **array unidimensional de *char***, con la particularidad que se añade al final **una marca de fin**
  - La marca de fin es el carácter nulo , '\0' , cuyo código ASCII es 0
  - El programador no tiene que ocuparse de insertar el carácter nulo, lo añade C automáticamente
- La longitud del vector es la de la cadena más uno
- El carácter nulo marca el fin de la cadena
- Ejemplo
  - `char ciudad[] = "Madrid";`
    - El compilador añade al final el carácter nulo
    - Ciudad tendrá siete elementos, el último almacena el '\0'

# Declaración de cadenas

- Declaración de un vector de caracteres

- `char nombreVector [longitud];`

- Declaración de un string

- `char nombreCadena [longitud];`
- debe terminar en un carácter nulo

```
char cadena_hola[]="Hola";
char otro_hola[]={ 'H', 'o', 'l', 'a', '\0' }; // Igual al anterior
char vector[]={ 'H', 'o', 'l', 'a' }; /* Un vector de 4 elementos,
 con los elementos 'H', 'o', 'l' y 'a' */
char espacio_cadena[1024]="Una cadena en C";
char cadena_vacia[]="";
```

- `cadena_hola` y `otro_hola` tienen una longitud de 5
- `cadena_vacia` tiene longitud 1

# Longitud de una cadena

- La marca de fin permite procesar la cadena sin conocer su longitud
  - Sabemos que termina al llegar a la marca de fin (el carácter '\0')
- Por ejemplo, el siguiente fragmento de código calcula la longitud de una cadena:

```
while (cadena[largo] != '\0') largo++;
```

# Asignación de valores a un string

- en la declaración

```
char cadena1[5]="hola";
char cadena2[]="hola";
char cadena3[5]={'h', 'o', 'l', 'a', '\0'};
char cadena4[]={ 'h', 'o', 'l', 'a', '\0'};
```

- no se puede hacer lo mismo si no es en la declaración

```
cadena = "hola" // no se puede hacer
cadena1 = cadena2 // no se puede hacer
```

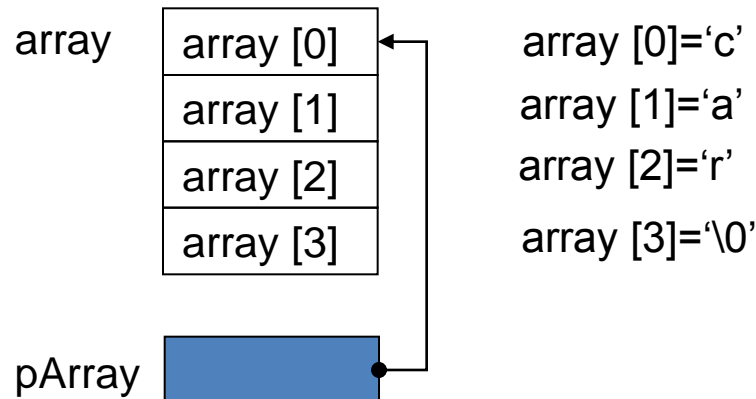
- sí se puede elemento por elemento, sin olvidar el carácter nulo

```
cadena [0]='\h' ;
cadena [1]='\o' ;
cadena [2]='\l' ;
cadena [3]='\a' ;
cadena [4]='\0' ;
```

# Relación entre string y punteros

- La relación entre strings y punteros es igual que la de cualquier tipo de array
  - El nombre de la cadena es la dirección de memoria del primer elemento.

```
char array[]="car";
char *pArray=array;
```



# Lectura y escritura de strings

- Con printf y scanf
  - especificador de formato para cadenas %s
  - Al usar scanf con strings, NO se usa el operador de dirección, '&'
    - ya que el nombre del string es un puntero a la dirección de comienzo de la cadena (se pasa por referencia).
  - La función scanf detiene la lectura cuando encuentra un espacio en blanco.

```
char str[100];
printf("Enter string: ");
scanf("%s", str);
printf("String is: %s", str);
```

```
Enter string: hello world
String is: hello
```

# Asignación de valores a un string usando strcpy

- copiar una cadena en otra
  - `s2 = s1;` // no funciona: se copian como punteros
- string copy **strcpy**
  - Función de biblioteca, está en la biblioteca `string.h`
  - Copia el contenido una cadena en otra
  - Argumentos
    - puntero a la cadena copia = nombre de la cadena copia
    - puntero cadena original = nombre de la cadena original

# strcpy

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

int main (void) {

 char str1[] = "helloworld";
 char str2[50];

 printf("String 1: [%s]\n", str1);
 printf("String 2: [%s]\n", str2);
 strcpy(str2, str1);
 printf("String 1: [%s]\n", str1);
 printf("String 2: [%s]\n", str2);

 return 0;
}
```

```
String 1: [helloworld]
String 2: [\371\277_\377]
String 1: [helloworld]
String 2: [helloworld]
```



# Otras funciones de biblioteca para manejar cadenas

- la biblioteca `string.h` proporciona otras funciones para trabajar con cadenas:

```
largo = strlen(cadena) // Para obtener el largo de una cadena
strcpy(destino, origen) // Copia el contenido de origen en destino
 // destino debe ser lo suficientemente grande
strcat(destino, origen) // Agrega el contenido de origen al final de destino
 // destino debe ser lo suficientemente grande
resultado = strcmp(cadena1, cadena2) // Compara dos cadenas
 // devuelve un valor menor, igual o mayor que 0 según si cadena1 es menor,
 // igual o mayor que cadena2, respectivamente.
posicion = strchr(cadena, caracter) // Devuelve la posición en memoria de la primer
 // aparición de caracter dentro de cadena
posicion = strstr(cadena, subcadena) // Devuelve la posición en memoria de la primer
 // aparición de subcadena dentro de cadena
```

# Ejemplos

- strncat concatena dos cadenas

```
#include <stdio.h>
#include <string.h>
...
char color[] = "rojo";
char grosor[] = "grueso";
...
char descripcion[1024];

strcpy(descripcion, "Lapiz color ");
strncat(descripcion, color, 1024);
strncat(descripcion, " de trazo ", 1024);
strncat(descripcion, grosor, 1024);
// descripcion contiene "Lapiz color rojo de trazo grueso"
```

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)

# Registros o estructuras: definición

- tipos de datos **definidos por el usuario** que permiten agrupar bajo un mismo nombre datos de **igual o distinto tipo** que están **relacionados lógicamente**.
  - Ejemplo a : Almacenar nombre, apellidos y número de teléfono en una estructura denominada contacto

contacto

|                 |
|-----------------|
| nombre          |
| apellidos       |
| numero teléfono |

- Ejemplo b: Estructura para gestionar cuentas de correo electrónico, donde se almacenaran distintos tipos de datos; nombre (login), clave de acceso (password), dirección (e-mail) y número de usuario (numero).

# Declaración una estructura o registro

- Para declarar una estructura se sigue el siguiente patrón

```
struct nombreEstructura {
 TipoDato_1 elemento_1;
 TipoDato_2 elemento_2;

 TipoDato_n elemento_N;
};
```

## Ejemplo

```
struct cuentaCorreo {
 char login [256];
 char password [256];
 char email [256];
 int numUsu;
};
```

# Declaración de una estructura

- Más ejemplos

- Estructura que almacena las coordenadas de un punto

```
struct coordenadas{
 float x;
 float y;
};
```

- Estructura que almacena los datos de una persona

```
struct persona{
 char nombre[20];
 char apellidos [50];
 int edad;
 float altura;
};
```

## Declaración de una variable del tipo definido por la estructura

- Para trabajar con una estructura hay que seguir **dos pasos**:
  - 1. Declarar la estructura
  - 2. Declarar una (o más) variables del tipo definido por la estructura

- Para declarar una variable se sigue el siguiente patrón:  
`struct nombre_Estructura nombre_Variable;`

- Ejemplos

```
struct coordenadas puntoA;
struct persona miVecino ;
struct cuentaCorreo cuentaLuis;
```

# Acceso a un elemento individual de la estructura

- Con las estructuras se puede trabajar a dos niveles, con la estructura completa o con los elementos de la misma de forma independiente
  - Para acceder a un elemento de la estructura se usa el operador miembro (.) y el nombre del elemento  
`cuentaLuis.numUsu`  
`cuentaLuis.login`
  - Diferente de los arrays, en los que para acceder a un elemento se usaba el índice

- más ejemplos

`puntoA.x`

- `puntoA` es una variable del tipo "coordenadas", una estructura definida por el usuario
- `(.)` es el operador miembro
- `x`, es el nombre de un miembro de la estructura "coordenadas"



# Acceso a un elemento individual de la estructura

## Más ejemplos

```
struct coordenadas {
 float x;
 float y;
};
struct persona{
 char nombre[20];
 char apellidos [50];
 int edad;
 float altura;
};
struct coordenadas puntoA;
struct persona miVecino ;

puntoA.y =100;
strcpy(miVecino.nombre, "Juan");
miVecino.edad = 45;
miVecino.altura = 1.90;

printf("%s \n", miVecino.nombre);
printf("%d \n", miVecino.edad);
printf("%4.2f \n", miVecino.altura);
```

## Ejemplo: Programa que determina si dos puntos son iguales

```
// declaramos la estructura coordenadas
struct coordenadas {
 float x;
 float y;
};

int main (void)
{
 // declaramos dos variables del tipo coordenadas
 struct coordenadas puntoA, puntoB;

 //leemos los valores del primer punto
 printf ("Coordenada x del punto A:");
 scanf ("%f", &(puntoA.x));
 printf ("Coordenada y del punto A:");
 scanf ("%f", &(puntoA.y));
```

## Ejemplo: Programa que determina si dos puntos son iguales (cont.)

```
//leemos los valores del segundo punto
printf ("Coordenada x del punto B:");
scanf ("%f", &(puntoB.x));
printf ("Coordenada y del punto B:");
scanf ("%f", &(puntoB.y));

// comprobamos si son el mismo punto
if ((puntoA.x==puntoB.x)&&(puntoA.y==puntoB.y))
 printf ("A y B son iguales\n");
else
 printf ("A y B son distintos\n");
return 0;
}
```

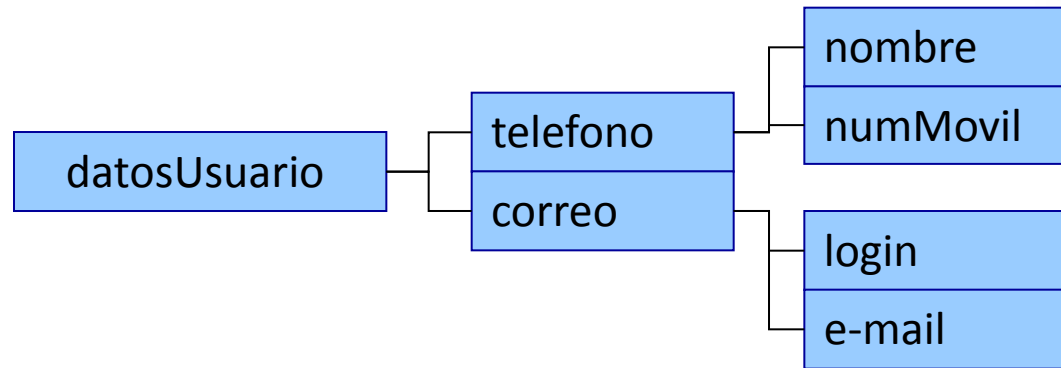
# Estructuras anidadas

- Un miembro de una estructura puede ser de tipo básico (int, float, char,...), pero también puede ser un array, un puntero u otra estructura

```
struct datosTel{
 char nombre[256];
 long numMovil;
};
```

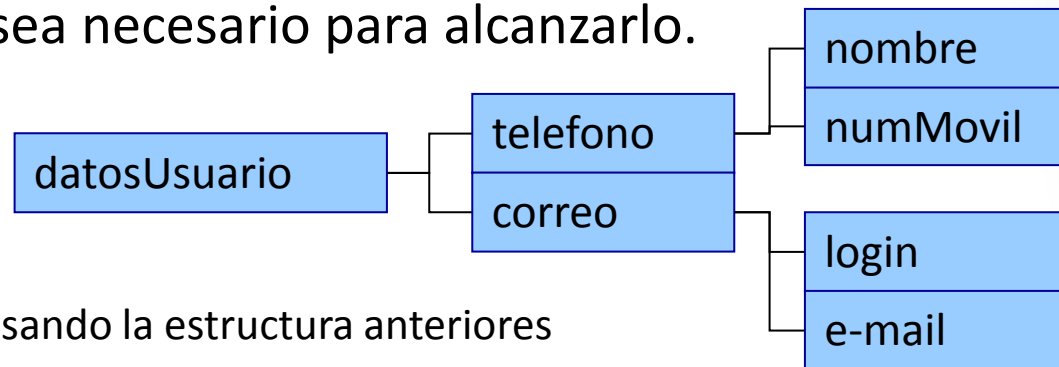
```
struct cuentaCorreo{
 char login[256];
 char email [256];
};
```

```
struct datosUsuario{
 struct datosTel telefono;
 //variable telefono declarada sobre estructura telefonoPersonal
 struct cuentaCorreo correo;
 //variable correo declarada sobre estructura cuentaCorreo
};
```



# Acceso a datos en estructuras anidadas

- Para acceder a un miembro anidado se usa el operador miembro, (.), tantas veces como sea necesario para alcanzarlo.



- Ejemplo:

- declaramos una variable usando la estructura anteriores

```
struct datosUsuario miVecino;
```

- para acceder al miembro numMovil del miembro telefono de la variable miVecino usaríamos:

```
miVecino.telefono.numMovil = 684567123;
printf ("%d", miVecino.telefono.numMovil);
```

- para acceder al primer carácter del miembro nombre de la variable miVecino usaríamos:

```
miVecino.telefono.nombre[0] = 'J';
printf ("%c", miVecino.telefono.nombre[0]);
```

# Inicialización, asignación y copiado de estructuras

- Para asignar valores a los miembros de una estructura podemos hacerlo en su inicialización, mediante asignación de estructuras o mediante asignación a miembros individuales de la estructura.

- Inicialización

- Se pueden iniciar los distintos miembros de una variable de tipo estructura en su declaración.

- Ej. En la estructura coordenadas:

```
struct coordenadas {
 float x;
 float y;
};
```

- Podemos declarar e inicializar la variable “origen”:

```
struct coordenadas origen = {0.0, 0.0};
```

# Inicialización, asignación y copiado de estructuras

- **Asignación:** se pueden copiar estructuras entre si directamente

```
struct ejemplo{
 char letra;
 long entero;
 char string[20];
};

struct ejemplo ejemplo1 = {'a', 23, "Hola"};
struct ejemplo ejemplo2;
struct ejemplo ejemplo3;

//copia en ejemplo2 los datos de todos los miembros de
//ejemplo 1, que ya estaba inicializado
ejemplo2=ejemplo1;
```

## Ejemplo: se puede copiar estructuras directamente

```
int main(int argc, char *argv[])
{
 struct calcular {
 int num1;
 int num2;
 int suma;
 char nombre;};
 struct calcular suma1={2,3,5,'a'};
 struct calcular suma2={3,4,7,'b'};
 struct calcular suma3={};
 struct calcular *resul,*resu2;
 printf ("%d %d %d %c\n", suma1);
 printf ("%d %d %d %c\n", suma2);
 printf ("%d %d %d %c\n", suma3);
 //Ojo es necesario un corversor de formato por cada variable de la estructura
 suma3=suma2;
 suma2=suma1;
 //Es posible copiar una estructura en otra directamente, no miembro a miembro
 //Ahora cada miembro de suma2 contiene lo almacenado en suma1
 printf ("%d %d %d %c\n", suma1);
 printf ("%d %d %d %c\n", suma2);

 resul=&suma2;
 resu2=&suma1;
 printf ("%d\n", resu2->num1);
 printf ("%d\n", resul->suma);
 printf ("%c\n", resu2->nombre);
 resul=&suma3;
 printf ("Tras asignar al puntero resul la direccion de suma3 en nombre hay:%c\n", resul->nombre);
 system("PAUSE");
 return (0);
}
```



## Ejemplo (cont)

```
2 3 5 a
3 4 7 b
0 0 0
2 3 5 a
2 3 5 a
2
5
a
Tras asignar al puntero resu1 la direccion de suma3 en nombre hay:b
Presione una tecla para continuar . . . _
```

# Inicialización, asignación y copiado de estructuras

- Asignación de valores individuales a cada miembro:

```
Ej. struct ejemplo{
 char letra;
 long entero;
 char string[20];
};

struct ejemplo ejemplo1 = {'a', 23, "estructura"};
struct ejemplo ejemplo2;
struct ejemplo ejemplo3;

ejemplo3.letra='b';
ejemplo3.entero=2345;
strcpy (ejemplo3.string1, "Cadena de ejemplo 1");
```

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.6. ARRAYS DE ESTRUCTURAS

# Definición

- Los arrays de estructuras son vectores o matrices en los que cada uno de sus elementos es una estructura.
- Son muy útiles para programar
  - permiten almacenar y gestionar datos de diferente tipo de forma cómoda

- ¿Cómo se declaran?

```
struct nombre_estructura nombre_array [dimensión]
```

# Array de estructuras: ejemplo

- Ejemplo

```
// declaramos la estructura
struct fecha{
 int dia;
 int mes;
 int anyo; //no hay ñs.
};
```

```
// declaramos un vector de cuatro elementos en el que cada
// elemento es una variable del tipo estructura que acabamos
// de declarar
struct fecha fechaNac[4];
```

| dia | mes | anyo |
|-----|-----|------|
|-----|-----|------|

| dia | mes | anyo |             |
|-----|-----|------|-------------|
| 5   | 10  | 1998 | fechaNac[0] |
| 17  | 3   | 2001 | fechaNac[1] |
| 30  | 1   | 2003 | fechaNac[2] |
| 27  | 12  | 2010 | fechaNac[3] |

```
//accedemos al miembro anyo, al tercer elemento del array
fechaNac[2].anyo =2010;
```

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.6.1. ARRAYS COMO PARÁMETROS DE FUNCIONES

- Arrays como parámetros
  - Hay algunas diferencias en función de si el parámetro es un vector (una dimensión) o si es un array de más de una dimensión
- Los arrays se pasan siempre por referencia
  - El nombre del array es un puntero al primer elemento del array

# Vectores como parámetros

- en la declaración:
  - se indica el tipo y el nombre del vector
  - se indica que es un vector añadiendo los [], pero no el tamaño

```
int sumaDeDatos(int v[]);
```

Prototipo de la función

- En la llamada:
  - nombre del vector, sin indicar tamaño ni corchetes
    - El nombre del vector es un puntero al primer elemento

```
int main(void) {
 int datos[100];
 ...
 suma=sumaDeDatos (datos);
 ...
}
```

Llamada a la función



# Vectores como parámetros

- Cuando se pasa un vector a una función lo que se está pasando es la dirección del primer elemento del vector.
  - En C todos los vectores se pasan por referencia (dirección)
  - Dentro de la función se puede cambiar el contenido del vector
  - Para indicar que la función no modifica el vector que se pasa como parámetro se puede utilizar el modificador **const**
    - tanto en el prototipo como en la cabecera)

```
int sumaDeDatos(const int datos[]);
```

```
int main (void){
 int vector[100];
 . . .
 sumaDeDatos(vector)
 . . .
}
```

Para indicar que el vector no se va a modificar en la función

# Vectores como parámetros

- Para facilitar el manejo de vectores, se suele pasar como parámetro adicional el número de elementos del vector

```
#include <stdio.h>
#define TAM 5

void leerVector(int a[]);

int main(void) {
 int v[TAM];
 printf("Introduzca los elementos del Vector\n");
 leerVector(v);
 return 0;
}

void leerVector(int a[]) {
 int i;
 for (i=0; i<TAM; i++)
 scanf("%d", &a[i]);
 return;
}
```

La función leerVector sólo se puede utilizar con vectores de tipo int y longitud TAM

# Vectores como parámetros

Ahora pasamos también el tamaño del vector como parámetro adicional

```
#include <stdio.h>
#define TAM 5

void leerVector(int a[], int longitud);

int main(void) {
 int v[TAM];
 printf("Introduzca los elementos del Vector\n");
 leerVector(v, TAM);
 return 0;
}

void leerVector(int a[], int longitud) {
 int i;
 for (i=0; i<longitud; i++)
 scanf("%d", &a[i]);
 return;
}
```

La función leerVector puede trabajar con vectores de tipo **int** de **cualquier longitud**.

## Ejemplo: copiar vectores

- Escriba un programa que lea los valores de los elementos de dos vectores de enteros, copie esos dos vectores en un tercero y muestre sus valores por pantalla

```
#include <stdio.h>
#define TAM1 5
#define TAM2 3

void leerVector(int a[], int num);
void imprimirVector (const int a[], int n);
void copiarVectores (const int a[], const int b[],
 int c[], int t1, int t2);
```

## Ejemplo: copiar vectores

```
int main(void) {
 int v1[TAM1], v2[TAM2], v3[TAM1+TAM2];

 printf("Introduzca los elementos del Vector1\n");
 leerVector(v1, TAM1);
 printf("Introduzca los elementos del Vector2\n");
 leerVector(v2, TAM2);

 copiarVectores(v1, v2, v3, TAM1, TAM2);

 printf("Los vectores introducidos son\n");
 imprimirVector(v1, TAM1);
 imprimirVector(v2, TAM2);
 printf("El vector obtenido es\n");
 imprimirVector(v3, TAM1+TAM2);
 return 0;
}
```

- permite leer y mostrar vectores de diferentes tamaños

## Ejemplo: copiar vectores

```
void leerVector(int a[], int n){
 int i;
 for (i=0; i<n; i++)
 scanf("%d", &a[i]);
 return;
}

void imprimirVector(const int a[], int n){
 int i;
 for (i=0; i<n; i++)
 printf ("%d\n", a[i]);
 return;
}
```

## Vectores como parámetros: Ejemplo (Cont.)

```
void copiarVectores (const int a[], const int b[],
 int c[], int t1, int t2){

 int i;
 for (i=0; i< t1+t2; i++){
 if (i<t1)
 c[i]=a[i];
 else
 c[i]=b[i-t1];
 }
 return 0;
}
```

# Matrices como parámetros

- El paso de arrays de más de una dimensión es diferente al paso de vectores
- es IMPRESCINDIBLE indicar explícitamente el tamaño de cada una de las dimensiones del array, excepto el de la primera que se puede indicar o no.



# Matrices como parámetros

- Por ejemplo en una tabla
  - Supongamos que la primera dimensión representa las filas y la segunda las columnas
  - Es necesario indicar explícitamente cuantas columnas tiene la matriz
  - Tanto en la cabecera como en el prototipo
  - En caso contrario, se produce un error de compilación.

```
#define FIL 3
#define COL 2

int leermatriz(int matriz[][COL]);
```

Prototipo de la función:  
Se indica explícitamente cuantas  
columnas tiene la matriz

# Matrices como parámetros: Ejemplo

Calcular el máximo de los elementos de una matriz de dos dimensiones

```
#include <stdio.h>

#define FIL 2
#define COL 3

int maximo(int a[][COL]);
void imprimirMatriz(int a[][COL]);

int main(void) {
 int matriz[FIL][COL];
 int i, j;
 // llenamos la matriz con datos
 for (i=0; i<FIL; i++)
 for (j=0; j<COL; j++)
 matriz[i][j]=i+j;

 imprimirMatriz(matriz);
 printf ("El maximo es %d\n", maximo(matriz));
 return 0;
}
```

Se indica explícitamente cuántas columnas tiene la matriz. El número de filas no es necesario indicarlo

## Matrices como parámetros: Ejemplo (Cont.)

```
int maximo(int a[][COL]) {
 int i, j, m;
 m=a[0][0];
 for (i=0; i<FIL; i++)
 for (j=0; j<COL; j++)
 if (m<a[i][j])
 m=a[i][j];
 return m;
}

void imprimirMatriz(int a[][COL]) {
 int i, j;
 for (i=0; i<FIL; i++){
 for (j=0; j<COL; j++)
 printf("%d\t",a[i][j]);
 printf ("\n");
 }
 return;
}
```

# Arrays como parámetros

- Recordar:
  - Para trabajar con funciones que reciben arrays como parámetros es necesario indicar explícitamente cada una de las dimensiones del vector, excepto la primera
  - Por lo que no podremos definir funciones que trabajen con arrays multidimensionales de diferentes tamaños.
- Existe una forma de superar este inconveniente
  - Memoria dinámica (tema 7)

- 6.1. INTRODUCCIÓN. DATOS ESTRUCTURADOS FRENTE A DATOS SIMPLES
- 6.2. DECLARACIÓN Y USO DE ARRAYS
- 6.3. PUNTEROS Y ARRAYS
- 6.4. CADENAS DE CARACTERES
- 6.5. ESTRUCTURAS DE DATOS DEFINIDAS POR EL USUARIO (REGISTROS)
- 6.6. ARRAYS DE ESTRUCTURAS
- 6.6. ARRAYS Y ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

## 6.6.2. ESTRUCTURAS COMO PARÁMETROS DE FUNCIONES

# Estructuras como parámetros

- Las estructuras se tienen que definir antes que los prototipos de las funciones que las usan
  - Se recomienda declarar todas las estructuras antes del main
- Por defecto el paso de estructuras es **por valor**
  - Igual que con int, float, char
  - Las modificaciones del valor de un campo del registro durante la ejecución de la función, no son visibles fuera de la función.
- Para pasar una estructura por referencia hay que
  - Pasar la dirección de la estructura
    - & delante del parámetro real
  - En la función, el parámetro formal es un puntero a la estructura
    - Y para acceder a un miembro de la estructura
      - Se usan paréntesis
      - (\* nombre\_estructura).miembro
      - Ej (\* datos).direccion

# Estructuras como parámetros. Ejemplo sencillo

Escribir un programa que lea las coordenadas de un punto en un espacio de tres dimensiones y calcule la distancia del punto al origen.

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

```
struct tPunto{
 float x, y, z;
};
```

leePunto modifica el valor de la estructura:  
paso por REFERENCIA  
\*p

```
void leePunto(struct tPunto *p);
float distancia(struct tPunto p);
```

distancia NO modifica el valor de la  
estructura: paso por VALOR  
p

```
int main(void) {
```

```
 struct tPunto pto;
 leePunto (&pto);
```

por REFERENCIA  
&pto

```
 printf ("distancia del punto al origen: %f\n", distancia(pto));
 return 0;
```

por VALOR  
pto

```
}
```

# Estructuras como parámetros. Ejemplo sencillo

```
float distancia(struct tPunto p)
{
 return sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
}

void leePunto (struct tPunto *p)
{
 printf ("X: "); scanf ("%f", &(*p).x);
 printf ("Y: "); scanf ("%f", &(*p).y);
 printf ("Z: "); scanf ("%f", &(*p).z);
 return;
}
```

por VALOR: p  
Ojo! aquí el \* es el producto,  
nada de punteros

por REFERENCIA: \*p  
Y para acceder a los  
elementos de la  
estructura se usan  
paréntesis  
(\*p).z

La notación (\*p).x es  
absolutamente equivalente a p->x  
scanf ("%f", &p->x)  
scanf ("%f", &p->y)  
scanf ("%f", &p->z)



## Estructuras como parámetros. Ejemplo avanzado 1/3

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

define NPROD 4

// estructura tipo producto
struct tipoProducto {
 char nombre [15];
 char codigo [10];
 float precio;
 int esFalso; // 1 si es falso, 0 si verdadero
};

void comprobarProducto (struct tipoProducto *p);
```

ComprobarProducto modifica el valor de la estructura: paso por REFERENCIA  
\*p

## Estructuras como parámetros. Ejemplo avanzado 2/3

```
int main(void) {
 // array de cuatro productos
 struct tipoProducto prod[NPROD];
 int i, numFalsos=0;
 //inicializamos los datos
 strcpy (prod[0].codigo, "UEX1002");
 strcpy (prod[1].codigo, "UEX2002");
 strcpy (prod[2].codigo, "UET3002");
 strcpy (prod[3].codigo, "UEZ1002");

 // comprobar cuantos productos son falsos.
 for (i=0; i<NPROD; i++){
 comprobarProducto (&prod[i]);
 numFalsos= numFalsos + prod[i].esFalso;
 }
 printf ("hay %i productos falsos \n", numFalsos);
 return 0;
}
```

por REFERENCIA  
&prod[i]

## Estructuras como parámetros. Ejemplo avanzado 3/3

```
void comprobarProducto (struct tipoProducto *p) {
 // función que recibe como parámetro una estructura de
 // tipo producto
 // y modifica el valor del elemento esFalso en función del
 // código del producto
 // son verdaderos los productos cuyos códigos empiecen por UEX
 (*p).esFalso = 1;

 //verificamos si el código es correcto
 if (((*p).codigo[0]=='U') && ((*p).codigo[1]=='E')) && ((*p).codigo[2]=='X')) {
 (*p).esFalso = 0;
 }
}
```

por REFERENCIA: \*p  
Y para acceder a los elementos  
de la estructura se usan  
paréntesis o simplemente ->  
(\*p).esFalso      (p->esFalso)  
(\*p).codigo[2]    (p->coidgo[2])

# TEMA 6.

# TIPOS DE DATOS ESTRUCTURADOS

Grado en Ingeniería en Tecnologías Industriales  
Programación

