Capítulo

5

Datos estructurados

En el quinto capítulo se avanza en la declaración de datos, viendo nuevos tipos de datos estructurados: primero el array como conjunto de datos del mismo tipo (tanto escalares como estructurados), haciendo énfasis en su equivalencia con los punteros y en un array especial que es el de caracteres o string (cadena); y segundo las estructuras (*struct*) como conjunto de datos de diferente tipo. Se termina con los enumerados.

Arrays y punteros

Los *arrays* surgieron por la necesidad de agrupar un conjunto de elementos del mismo tipo, de una forma eficiente. Vamos a ver esto con un ejemplo, supongamos que estamos haciendo un programa que trabaja con los litros por metro cuadrado caídos en un año, si no dispusiéramos de los *arrays*, tendríamos que definir doce variables de tipo flotante a las que tendríamos que asignar valores, veamos en un fragmento de programa como se haría con y sin *arrays*:

```
float lluvia[12];
     int ind;

for (ind = 1; ind <= 12; ind++)
     scanf("%f", lluvia[ind]);
     .....
```

```
float llu_1, llu_2, llu_3, llu_4,
llu_5, llu_6, llu_7, llu_8, llu_9,
llu_10, llu_11, llu_12;

scanf("%f", &llu_1);
scanf("%f", &llu_2);
scanf("%f", &llu_3);
        .....
```

Se ve claramente que la segunda forma es muy poco práctica, imaginemos lo que tendríamos que hacer si nos interesara la lluvia caída cada día en un año.

Podemos definir a los datos estructurados como una reunión de tipos simples caracterizados por un método de estructuración particular. En concreto, el *array* es un grupo de elementos del mismo tipo a los que se les da un nombre común, y donde los datos individuales de los que está compuesto pueden ser accedidos por medio de un índice.

Los *arrays* se declaran como cualquier tipo de variable vista anteriormente, salvo que habrá que encerrar entre corchetes el número de elementos que queremos almacenar. A diferencia del Fortran, en C los *arrays* comienzan en la posición 0 y a diferencia del Pascal, en C sólo se da este valor en el *array* y no el rango entre los que puede variar el índice, la declaración sería (en C99 se pueden definir arrays asignados dinámicamente -no dinámicos-):

```
int array_de_enteros[100];
int array_de_enteros[n]; /* para C99, n tiene que tener ya valor */
```

En este caso se está declarando un array de 100 enteros. Si queremos acceder a un elemento del *array*, simplemente pondremos el nombre del *array* seguido entre corchetes del índice del elemento en cuestión:

```
array_de_enteros[99] = 10;     /* atencion es el ultimo elemento */
```

El índice de la variable siempre será entero y el tipo de variable de que está compuesto el array puede ser cualquier otro tipo.

Es un error muy común acceder al último elemento de *array* repitiendo el número de elementos (en el ejemplo el elemento 100 [99]). Esto es incorrecto, ya que el *array* empieza en cero, y ese dato no existe. Aún la situación puede ser peor: que el compilador no nos diga nada, y al ejecutar el programa puede que funcione, pero debemos saber que ese elemento está machacando alguna otra variable, con lo que el error puede ser difícil de encontrar.

En ANSI C los arrays se pueden inicializar sin necesidad de ser estáticos (a diferencia del C K&R), si lo son valdrán cero. Además, para inicializarlos explícitamente, habrá que poner entre llaves los valores de cada elemento separados por comas:

```
int dias[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Si no ponemos todos los elementos, los restantes valdrán cero si es estático, si ponemos demasiados el compilador nos dará un error. No siempre es necesario dar un límite a los arrays, éste se puede tomar de la inicialización (como en el ejemplo anterior, en el cual se omite 12).

Un aspecto diferenciador respecto de otros lenguajes, es que los arrays por definición también son punteros (constantes de tipo puntero), su nombre es un puntero al primer elemento del array, por lo tanto es verdad que:

```
array == &array[0];
```

Por lo tanto, el uso de arrays y punteros es equivalente, siempre que tengamos en cuenta que un array es un puntero (una dirección) constante que no puede cambiar de valor.

Las operaciones que se pueden realizar con punteros son:

1. La asignación "=" de dos punteros igualará las direcciones a las que apuntan.

2. La comparación "==" dice si las direcciones a las que apuntan son iguales.

3. La indirección "*" nos da lo que contiene la dirección.

4. La dirección "&" nos da la dirección donde está guardado el puntero, no la dirección a la que apunta (que es el propio puntero).

5. El incremento o decremento de punteros "++" "--" aumenta o reduce la dirección a la que apunta en tantas unidades como bytes ocupe el tipo al que pertenece, es decir, si tenemos un puntero de tipo *float* y en nuestro sistema el tipo *float* ocupa cuatro bytes, un incremento del puntero en una unidad añadirá cuatro bytes a la dirección a la que apunta:

```
puntero_flotante         <=>     dirección
puntero_flotante + 1     <=>     dirección + 4
array + 2                ==      &array[2]
```

6. La diferencia "-" de dos punteros nos dará en número de unidades que les separan (normalmente dentro de un array).

```
&array[2] - &array[4]    ==    2
```

Para ver esto se puede hacer el siguiente programa:

```
#include <stdio.h>
#define TAM 3
main ()
{
   int fechas[TAM], *pun, indice;
   double  facturas[TAM], *punf;

   pun = fechas;          /* asignacion de punteros */
   punf = facturas;
   printf("tamaños %lu %lu \n", sizeof(int), sizeof(double) );
   for (indice = 0; indice < TAM; indice++)
       printf("punteros + %d: %10p %10p \n", indice,
                               pun + indice, punf + indice);
}
```

Que produciría una salida parecida a ésta:

```
tamaños 4 8
punteros + 0: 0x7fff3df925d0 0x7fff3df925e0
punteros + 1: 0x7fff3df925d4 0x7fff3df925e8
punteros + 2: 0x7fff3df925d8 0x7fff3df925f0
```

Esta equivalencia entre punteros y *arrays* también tiene sus consecuencias en las llamadas a funciones, ya que los *arrays* siempre se pasarán por referencia, es decir, pasaremos su dirección (el propio nombre). Así, si hacemos una función que calcule la media de un *array* de enteros:

```
media(int array[], int n)
{
       int indice;
       int suma;
       for (indice = 0, suma = 0; indice < n; indice++)
              suma += array[indice];
       return(suma/n);
}
```

Puede ser fácilmente convertida a punteros:

```
media( int *pa, int n)
{
       int indice;
       int suma;
       for (indice = 0, suma = 0; indice < n; indice++)
              suma += *(pa + indice);
       return(suma/n);
}
```

Observa que cuando se declara el argumento array en la función no hace falta dar su tamaño, ya que éste es tomado de la llamada a la función. Y como la representación de *arrays* o punteros es la misma, de hecho se podría usar cualquiera de las dos funciones sin necesidad de cambiar la llamada. También es de destacar que la operación *(pa + indice) lo que hace es desplazar el puntero según el tipo de elemento al que apunta y después obtener su valor, es equivalente a obtener los valores de los distintos elementos de un array. Hay que tener en cuenta que cuando se utilizan punteros no se chequea que nos hayamos salido del array que apuntamos, lo cual puede ser muy peligroso, y que sólo se pueden incrementar los punteros variables como *pa* pero no los constantes como puede ser el nombre de una array, se puede ver más claro con la siguiente equivalencia imposible:

```
array = array + 3;     ⇔     5 = 5 + 3;
```

es tan absurdo intentar cambiar el valor de la constante entera "5" como la dirección del *array*.

Arrays multidimensionales

Al igual que con otros lenguajes, en C se pueden definir *arrays* de varias dimensiones, simplemente declarando cada dimensión entre corchetes:

```
int matriz[4][5];
```

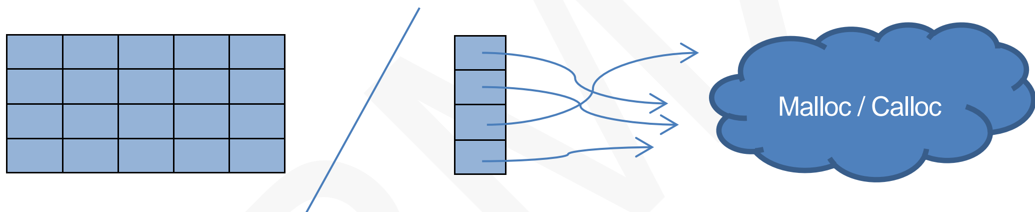
a diferencia del FORTRAN su organización es por filas, el orden de los elementos para el ejemplo anterior será:

```
00, 01, 02, 03, 04, 10, 11, 12, ...
```

En C una *array* de dos dimensiones es una *array* de *arrays* de una dimensión, es decir, un *array* a punteros (es equivalente pero con distintas formas) y por eso, para acceder a un elemento no se escribe matriz[3,4], sino matriz[3][4]. La equivalencia sería:

```
int *matriz_punteros[4];
```

Pero a diferencia de la forma de array multidimensional donde se reservan 4 x 5 enteros, en este caso sólo se crea espacio para cuatro punteros para los que se necesita crear espacio e inicializar un array de 5 enteros con las peticiones de memoria dinámica vistas en el capítulo anterior.



Al igual que los unidimensionales, los multidimensionales se pueden inicializar separando cada dimensión por llaves o directamente teniendo en cuenta su almacenamiento (para una de 2x2):

```
int matriz[2][2] = { {1,2}, {2,4} };
int matriz[2][2] = { 1, 2, 2, 4 };
```

Debido al método de almacenamiento, también se puede trabajar con ellos como si fueran punteros, así, si tengo un array de enteros de 4x2 y un puntero a enteros con la misma dirección, es equivalente irse a la tercera fila segundo elemento, que al almacenado en la sexta posición:

```
puntero + 5 == array[2][1]
```

Otra característica de los *arrays* multidimensionales es que la omisión de una dimensión nos indica una agrupación interna de la misma (*slice*), es decir, si tenemos un array de dos dimensiones (5x8) y sólo utilizamos un corchete, nos estamos refiriendo a las filas. Esto significa, como hemos dicho, que un array de dos dimensiones es equivalente a un array de punteros (a los que hay que definir la segunda dimensión) o a un puntero de punteros al que hay que definir las dos dimensiones:

```
matriz[1] == &matriz[1][0]     /* la fila 1 */
...

int matriz[5][COL] ⇔ int *puntero[5]    /* como array de punteros*/
for(i=0;i<5;i++)  /*peticion de memoria para la segunda dimension */
     pun[i] = (int*) calloc(COL, sizeof(int));
...
pun[2][2] = 28;  /*ejemplos de acceso como matriz o puntero*/
*(pun[2]+2) = 28;
...
```

```
int matriz[FIL][COL] ⇔ int **puntero  /* como puntero a puntero */
/* habria que pedir memoria para los punteros, mas complejo */
doble  = (int**) calloc(FIL, sizeof(int*)); /* primera dim */
for(i=0;i<FIL;i++)
     doble[i] = (int*) calloc(COL, sizeof(int)); /* segunda dim */
...
doble[2][2] = 39; /*ejemplos de acceso como matriz o puntero*/
*(doble[2]+2) = 28;
...
```

Por último, cuando se llama a una función con arrays multidimensionales, siempre hay que decirle el tamaño de la fila (al menos), es decir el número de columnas, ya que lo que hacemos es pasarle a la función realmente un puntero a un array de filas como la segunda versión del ejemplo anterior (es responsabilidad del programador no pasarse con el número de filas al recorrerlas dentro de la función).

Así, estas definiciones son equivalentes:

```
función(int matriz [2][3])  {...}  /* con las dos */
función(int matriz [][3])   {...} /* con una dimensión */
función(int (*matriz)[3])  {...}  /* puntero a array */
```

En el primer caso le estamos pasando un array de seis elementos (la matriz es de 2x3); en el segundo le estamos diciendo que el número de elementos por fila es 3, con lo cual ya se sabe la organización completa (no pasarse con las filas); en el tercero un puntero a un array de 3 enteros (como una fila), se usan los paréntesis ya que el operador “[ ]” tiene más precedencia que el operador “*”. No confundir con `int *matriz [3]` que sería un array de 3 punteros.

Tiras de caracteres: *strings*

En C no existe como tipo el *string* o tira de caracteres (un mensaje o sucesión de caracteres con una determinada longitud), simplemente se asocia este tipo de dato al *array* de caracteres terminados en el carácter nulo '\0' que se usa para conocer la longitud del mismo (funciona como terminador y ocupa una posición):

```
char tira[20];
```

Las tiras, como los arrays, son también punteros (constantes), pero para inicializarlos y no tener que trabajar directamente con el carácter nulo, se usan las comillas, las cuales lo incluyen automáticamente:

```
char tira[20] = "esto es un mensaje";
char *pun_tira = "Caray";
```

En la inicialización, todos los caracteres sin asignar serán puestos a nulos. Siempre hay que poner una longitud igual o mayor que la necesaria, ya que si no estaríamos cometiendo un error. Por eso, también se pueden declarar sin poner la longitud, con lo cual se cogería como longitud la del mensaje más el carácter nulo.

```
char tira[] = "Este es otro mensaje";
```

que tendrá una longitud de 21 (20 del mensaje y uno del nulo).

Se puede acceder a los caracteres individuales utilizando corchetes:

```
tira[12] = 's';
```

Para la entrada y salida se utiliza el conversor %s, al ser la tira por definición un puntero, no hay que leerlo con el operador dirección &, actuando la propia tira de caracteres encerrada entre comillas como un puntero:

```
printf("%s\n", "Otro mensaje");
scanf("%s", tira);
printf("%s, %u, %c \n", "Me", "oyes", *"bien");
```

en el último caso estamos pintando la tira "Me", la dirección de "oyes" (será una dirección del propio código del programa) y lo que contiene la dirección de "bien" (es decir *bien*) como carácter (el primero, o sea, b).

También existe la posibilidad de tener tiras de varias dimensiones en forma de array de punteros a *char* :

```
char *array[5] = {"a","b","c","d","e"};
```

define cinco punteros a caracteres. Así:

```
*array[0] == "a",   *array[1] == "b", ....
```

Esta definición es equivalente a definir un array multidimensional de caracteres:

```
char array [5][8] = {"a","b","c","d","e"};
```

pero en el caso anterior, el tamaño de los arrays era variante y se tomaba de la longitud del mensaje de inicialización (en todos los casos 1) y en este la longitud es fija y es igual a 8.

En C existen varias funciones definidas para trabajar con tiras de caracteres dentro de la librería estándar. El primer grupo de funciones corresponde a la entrada / salida. Al igual que con los caracteres, en los que podíamos usar las funciones *printf* y *scanf* o *getchar* y *putchar*, existen para las tiras las mismas funciones que se conocen como *gets* y *puts*.

El uso del *scanf* es el que ya hemos mencionado, sólo hay que tener en cuenta tres precauciones:

1. Si queremos leer una tira de caracteres con *scanf*, no podemos declararla como puntero a *char*, ya que en esta declaración no decimos en ninguna parte el tamaño de la tira, recordad que se cogía de la inicialización.

2. Si utilizamos el formato de *array* de caracteres, hay que tener en cuenta que no se podrán usar dentro de la tira los caracteres espaciadores como retorno de carro, espacio o tabulador, ya que la función lo considerará como final de entrada.

3. El *scanf* no quita del buffer de lectura el '\n' introducido para leer los datos, con lo que permanecerá para la siguiente lectura.

Debido a esto, el *scanf* sólo se utiliza cuando se quieren leer varios datos de distinto tipo a la vez, cuando sólo se quieren leer tiras, es mucho más cómodo y compacto el uso de *gets*().

El *gets*() toma caracteres desde la entrada hasta que se pulsa el enter ('\n') y una vez encontrado lo convierte al carácter nulo y lo pone al final de la tira. Su declaración está hecha en *<stdio.h>*. Es una función obsoleta y peligrosa que debería sustituirse por fgets (no quita '\n').

Como dijimos, hay que tener en cuenta que siempre se deben usar punteros con direcciones válidas, por eso lo normal es utilizar un array:

```
char tira[81];
gets(tira);
```

pero también se puede usar el valor devuelto que será un puntero a carácter, con lo cual tendríamos:

```
char tira[81];        /* declaracion de array */
char *ptr;            /* declaracion de puntero a carácter */
ptr = gets(tira);     /* lectura de dos formas */
```

Además, si se encuentra el carácter EOF o ha habido algún error en la lectura, la función devolverá nulo que se puede especificar en el programa con la constante NULL, definida en la librería *stdio.h*, y por lo tanto hacer los chequeos de entrada convenientes:

```
while (gets(tira) != NULL)
              ...
```

Para realizar la salida se utiliza *puts*(), que lo único que necesita es una tira:

```
char tira[] = "Hola";
puts(tira);
puts(&tira[2]);
```

Empezará a pintar todos los caracteres que encuentre hasta encontrar uno nulo (si no lo encuentra no parará), el cual convertirá en retorno de carro. La diferencia con *printf* es que esta última no incorpora automáticamente el retorno de carro y hay que ponerlo explícitamente.

A parte de las funciones de entrada/salida, existen definidas en la librería estándar (usar *string.h)* cuatro funciones para trabajar con tiras de caracteres:

- strlen(): toma una tira como argumento y devuelve un entero con su longitud sin contar el carácter nulo.

- strcat(): toma como argumentos dos tiras de caracteres y devuelve en la primera la unión de ambas. Para ello debe asegurarse que en la primera tira caben las dos unidas.

- strcmp(): toma como argumentos dos tiras de caracteres y compara sus contenidos, no sus direcciones. Devuelve un entero que indica si las tiras son iguales o no. En el primer caso devolverá un 0, en el segundo un entero positivo o negativo dependiendo de si la primera tira es mayor o menor que la segunda. El valor absoluto será la diferencia de código ASCII.

- strcpy(): toma como argumentos dos tiras de caracteres y copia la segunda en la primera, es decir sus contenidos, no sus direcciones. Hay que comprobar que la primera tiene espacio suficiente.

En el siguiente ejemplo se muestra el uso de las funciones de strings para hacer una búsqueda de uno, en un array de strings:

```
#include <stdio.h>
#include <string.h>
#define  TAM 5
int main ()
{
    int i;
    char nombre[81];
    char agenda[TAM][81] = {"Antonio", "Ines", "Juan", "Pepe","Pedro"};

    puts("Escribe el nombre : ");
    gets(nombre);

    for(i=0;i<TAM;i++)
    {
```

```
        if (strcmp(nombre, agenda[i]) == 0)
        {
           printf("%s ",nombre);
           puts("encontrado");
           break;
        }
    } // fin de for
   if (i==TAM)
        puts("No existe");
} // fin de main
```

Registros: *struct*

Hemos visto en el apartado anterior, que los *arrays* sirven para tratar conjuntos de datos del mismo tipo. Cuando queremos trabajar con grupos de elementos de distinto tipo se utiliza el *struct* (estructura o registro). A los elementos se les llama miembros y pueden ser de cualquier otro tipo, incluidos *arrays* y *structs*.

Los registros son conceptualmente como las fichas que se utilizan en las bases de datos: personas (nombre, apellidos, año de nacimiento, profesión, etc.); coches (matrícula, marca, modelo, año de fabricación, etc.); libros de una biblioteca (título, autor, editorial, ejemplares...) y cualquier otro tipo de objeto que pueda ser descrito por sus partes o características, como un número complejo (parte real e imaginaria).

La declaración de este tipo de dato estructurado es la siguiente:

```
struct estructura
{
        char nom_miembro1;
        int  nom_miembro2;      /* cualquier tipo */
             .....
        char array[81];
} ;
```

Esto forma un patrón (*template*) con un nombre (*tag*) para después declarar variables de este tipo. En la declaración de la variable, hay que escribir la palabra reservada *struct* seguida del nombre del patrón y del nombre de la variable:

```
struct estructura mia;
```

Como en el caso de las variables escalares, se pueden definir varias de golpe separándolas por comas:

```
struct estructura mia, tuya;
```

También se pueden declarar en bloque el *tag* y la variable, poniendo el nombre de esta justo después de la definición del *tag*:

```
struct estructura {...} mia;
```

Como en el caso de los *arrays*, en ANSI C las estructuras se pueden inicializar en la definición (en la declaración de la variable, no en el *tag*):

```
struct estructura mia = {'a', 23, "pepe"};
```

Para acceder a cada uno de los miembros se tendrá que utilizar un operador nuevo, el punto ".", este operador es binario y tiene la más alta prioridad:

```
mia.nom_miembro1;
```

También se pueden declarar *arrays* de estructuras (para formar por ejemplo bases de datos), estructuras de estructuras y punteros de estructuras:

```
struct estructura mia[100];     /* array de estructuras */

struct fecha        /* estructura con estructuras */
{
       int dia, mes, agno;
} ;

struct hombre
{
       struct fecha nacimiento;
       char nombre[81];
} manuel ;

struct hombre *tu;        /* puntero a estructuras */
```

y utilizarlas:

```
mia[32].nom_miembro1 = '2';
manuel.nacimiento.dia = 13;
tu = &manuel;
```

Cuando son punteros a estructuras, existe un nuevo operador (es una atajo) que accede directamente a los miembros de la estructura, para evitar complicar la nomenclatura de C. Se debería poner `(*tu).nacimiento`, no `*tu.nacimiento`, (el "." tiene más precedencia que el "*"). Ese atajo es el de pertenencia a estructura "->", es binario y tiene máxima prioridad (equivalente al "." pero para punteros):

```
tu -> nacimiento     <=>    manuel.nacimiento
```

Las operaciones válidas sobre estructuras son: copiarlas (asignarlas como una unidad), tomar su dirección ("&"), y tener acceso a sus miembros ("." o "->"). La primera operación implica que se pueden pasar como argumentos a funciones y que además éstas pueden devolver estructuras. Adviértase que se pasa a la función toda la estructura, por lo que suele ser mucho más eficiente pasar un puntero a esa estructura que la estructura entera.

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

struct punto    // esto hace de tipo de dato
{
 char nombre[80];
 int x;
 int y;
} global;

int main ()
{
        struct punto A = {"pepe", 11, 15};  /* en inicialización */
        struct punto B;

        struct punto suma(struct punto, struct punto);   /* prototipo */

 strcpy(B.nombre ,"juan");   /*no se puede hacer = "juan", es constante */
 B.x=4;                      /* asignación */
 B.y=0;

 global=suma(A,B);   /* llamada a función */

 printf("Ahora vale %s %d %d \n", global.nombre, global.x, global.y);

 return(0);
}
```

```
/*definición de la función de tipo punto */
struct punto suma (struct punto uno, struct punto dos)
{
 uno.x += dos.x;
 uno.y += dos.y;
 return uno;
}
```

Un prototipo con punteros a estructuras quedaría como:

```
 struct punto suma(struct punto *, struct punto *);   /* prototipo */
```

Cuando sumamos un puntero a una estructura, lo que estamos haciendo es movernos tantas posiciones de memoria como bytes ocupe la estructura. Pero hay que tener en cuenta que ese número de bytes no tiene porqué ser la suma que ocupan todos sus miembros ya que algunos de ellos pueden estar alineados a posiciones pares o múltiplos de palabra y dejar huecos en la estructura.

typedef

Vimos en el tema uno la palabra reservada *typedef* que nos dejaba renombrar tipos de datos, esto puede ser útil para definir estructuras de forma abreviada. Pero también puede servir para cualquier tipo de dato y aumentar la claridad del código:

```
typedef int Longitud;
typedef char * Cadena;
typedef char CAdena[80];

typedef struct punto Punto;
typedef struct {float real; float imag;} Complejo;
typede float complejo[2];
typedef struct punto * Puntoero;
```

hemos usado la inicial en mayúscula para diferenciarlas del resto (punto es distinto de Punto).

Usar este tipo de datos es más sencillo y claro, incluso se pueden usar en typecast:

```
Longitud longi;
Cadena mensaje;
CAdena otro_mensaje;
Punto geometrico;
Complejo dato;
Puntoero punteroestructura;

mensaje = (Cadena)malloc(80);
otro_mensaje[23]='a';
geometrico.x=3;
dato.imag=1.2;
```

Datos enumerados

El ANSI C ha creado un nuevo tipo de dato no existente en C K&R llamado enumerado, para ello se emplea la palabra reservada *enum* seguida del identificador del patrón de constantes y de una lista de valores constantes entre llaves que la variable puede tomar (es parecido al *tag* de las estructuras):

```
enum colores {verde, rojo, amarillo, violeta, azul, naranja};
```

Los valores se almacenan en enteros, de tal manera que verde vale 0, rojo 1, amarillo 2, etc. Estos valores se pueden alterar poniéndolos en la declaración:

```
enum niveles {bajo = 100, medio = 500, alto = 2000};
```

Si una de esas constantes se especifica, las siguientes tomaran valores consecutivos (medio será 101 y alto 102):

```
enum niveles {bajo = 100, medio, alto};
```

También son válidos los caracteres al ser análogos a enteros:

```
enum controles {campana='\a', línea = '\n', tabulador = '\t'};
```

Los operadores válidos para este tipo de operandos son la asignación simple, la comparación (sólo igualdad y desigualdad) y los aritméticos para constantes. No se pueden usar como índices de un *array*.

Como en el caso de las estructuras, una vez que hemos definido el tipo de datos (*tag*) tendremos que crear la variable correspondiente:

```
enum niveles este_nivel;
```

Ejemplos

```
1   /* Ejemplo 24.c
2
3   Entiendo los punteros y su equivalencia con los arrays
4   Comprende que los arrays son punteros constantes     */
5
6
7
8   #include <stdio.h>
9   #define TAM 3
10  int main ()
11  {
12      int fechas[TAM]={1,2,3}, *pun, indice, suma;
13      double  facturas[TAM]={1.0,2.0,3.0}, *punf;
14
15      pun = fechas;          /* asignacion de punteros */
16      // fechas=pun;         /* array = array + 3;  <=>   5 = 5 + 3; */
17      punf = facturas;
18      printf("tamaños %lu %lu \n", sizeof(int), sizeof(double) );
19      for (indice = 0; indice < TAM; indice++)
20          printf("punteros + %d: %10p %10p \n", indice, pun + indice, punf + indice);
21      for (indice = 0; indice < TAM; indice++)
22          printf("indice  %d: %d  %d %lf  %lf\n", indice, fechas[indice], *pun+indice, facturas[indice], *punf + indice);
23      for (indice = 0, suma = 0; indice < TAM; indice++)
24          suma += *(pun + indice);
25      printf("Suma: %d \n", suma);
26      return(0);
27  }
28
29
```

```
1   /* Ejemplo 25.c
2
3   Entiende la definición de arrays dobles     */
4
5
6   #include <stdio.h>
7   #define TAM 3
8   int main ()
9   {
10  /*  int matriz[TAM][TAM]={1,2,3,4,5,6,7,8,9};  tambien vale */
11      int matriz[TAM][TAM]={{1,2,3},{4,5,6},{7,8,9}};
12      int i,j;
13
14      for (i = 0; i < TAM; i++)
15          for (j = 0; j < TAM; j++)
16              printf("Matriz [%d][%d]: %d\n", i,j,matriz[i][j]);
17
18      return(0);
19  }
20
21
```

```
1   /* Ejemplo 26.c
2
3   Observa como se accede a un puntero doble.
4   Trata de predecir los valores que saldran de las impresiones    */
5
6
7   #include <stdio.h>
8
9   int main(void)
10  {
11
12      int Dato = 9;
13      int *refDato;
14      int **refRefDato;
15
16      refDato = &Dato;
17      refRefDato = &refDato;
18
19      printf("La longitud es %lu\n", sizeof(long));
20
21
22      printf("La direccion de Dato es %p\n", &Dato);
23      printf("La direccion de refDato es %p\n", &refDato);
24      printf("La direccion de refRefDato es %p\n", &refRefDato);
25      printf("La direccion de Dato es %d\n", &Dato);
26      printf("La direccion de refDato es %d\n", &refDato);
27      printf("La direccion de refRefDato es %d\n", &refRefDato);
28
29      printf("La direccion almacenada en refDato es %p\n", refDato);
30      printf("La direccion almacenada en refRefDato es %p\n", refRefDato);
31
32      printf("El valor almacenado en Dato es %ld\n", *refDato);
33      printf("El valor almacenado en refDato es %p\n", *refRefDato);
34
35      printf("El valor almacenado en Dato es %ld\n", **refRefDato);
36
37      return(0);
38  }
39
```

```
1   /* Ejemplo 27.c
2
3   Observa como se accede a una matriz doble.
4   Observa como se accede a un puntero doble.
5   Y comprende las equivalencias entre punteros y que asignaciones están bien y mal    */
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #define TAM 3
10  int main ()
11  {
12      int matriz[TAM][TAM]={{1,2,3},{4,5,6},{7,8,9}};      /* como matriz ya conocido */
13      int *array_pun[TAM];                                  /* como array de punteros*/
14      int *pun;                            /* como puntero unico */
15      int **doble;                            /* como puntero doble */
16      int i,j;
17
18      /* primera version como array de punteros */
19      for(i=0;i<TAM;i++)  /*peticion de memoria para la segunda dimension */
20          array_pun[i] = (int*) calloc(TAM, sizeof(int));
21
22      array_pun[2][2] = 28;  /*ejemplos de acceso como matriz o puntero*/
23      *(array_pun[2]+2) = 28;
24      /* comprobacion */
25      puts("Como array de punteros");
26      for (i = 0; i < TAM; i++){puts("");
27          for (j = 0; j < TAM; j++)
28              printf("Matriz [%d][%d]: %3d     ", i,j,array_pun[i][j]);}
29
30      /* segunda version como puntero doble */
31      doble  = (int**) calloc(TAM, sizeof(int*)); /* primera dimension */
32      for(i=0;i<TAM;i++)
33          doble[i] = (int*) calloc(TAM, sizeof(int)); /* segunda dimension */
34
35      doble[2][1] = 39; /*ejemplos de acceso como matriz o puntero*/
36      *(doble[2]+1) = 39;
37      *(*(doble+1)+1)=27;
38      puts("\nComo puntero doble");
39      /* comprobacion */
40      for (i = 0; i < TAM; i++){puts("");
41          for (j = 0; j < TAM; j++)
42              printf("Matriz [%d][%d]: %3d     ", i,j,doble[i][j]);}
43
44      puts("\nComo puntero simple a la matriz");
45      pun=(int*)matriz; /* solo lo asignamos para acceder a matriz conversion de direcciones*/
46      for (i = 0; i < TAM*TAM; i++)
47          printf("indice  %d: %d  \n", i,*pun+i);
48
49      doble=(int **)matriz;    // si hacemos esto la segunda indireccion ira a 1 error !!
50
51      return(0);
52  }
53
```

```c
/* Ejemplo 28.c

Comprende el uso de un array especial de caracteres que se llama string    */


#include <stdio.h>
#include <string.h>
#define TAM 40
int main ()
{
    char mensaje[TAM]="Hola mundo";
    char otromensaje[]="Hola mundo";
    char *pun="como puntero";        // de donde coge la direccion ?
    int i;


    printf("El mensaje es %s \n",mensaje);
    puts(otromensaje);
    puts(pun);

    for (i = 0; i < TAM; i++)
        printf("%c", mensaje[i]);   // pintar nulos no hace nada

    pun=gets(mensaje);
    puts(mensaje);
    puts(pun);

    //  mensaje="otro mensaje";  /* dara error */
    strcpy(mensaje,"Esto si funciona");
    puts(mensaje);
    pun="esto tambien funciona";   /* esto si */
    puts(pun);

    return(0);
}

```

```c
/* Ejemplo 29.c

Observa como se trabaja con strings y las inicializaciones.
Usa la librería de manejo de strings   */


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

int main ()
{
    int i;
    char nombre[81];
    char agenda[TAM][81] = {"Antonio", "Ines", "Juan", "Pepe", "Zacarias"};

    puts("Escribe el nombre : ");
    gets(nombre);

    for(i=0;i<TAM;i++)
    {
        if (strcmp(nombre, agenda[i]) == 0)
            {
            printf("%s ",nombre);
            puts("encontrado");
            break;
        }
    }
    if (i==TAM)
        puts("No existe");
    return(0);
}

```

```
1   /* Ejemplo 30.c
2
3   Observa como se trabaja con strings y las inicializaciones.
4   Usa la librería de manejo de strings
5   En este caso con un algoritmo más inteligente que la fuerza bruta    */
6
7
8   #include <stdio.h>
9   #include <string.h>
10  #define TAM 5
11  int main ()
12  {
13      int inicio = 0;
14      int mitad, fin = TAM-1;
15      char nombre[81];
16      static char agenda[TAM][81] = {"Antonio", "Ines", "Juan", "Pepe","Zacarias"};  // ordenado
17      puts("Escribe el nombre : ");
18      gets(nombre);
19      do
20      {
21          mitad = (inicio + fin) / 2;
22  //      printf("%d  %d  %d\n",inicio, mitad, fin);
23          if (strcmp(nombre, agenda[mitad]) < 0)
24              fin = mitad - 1;
25          else if (strcmp(nombre, agenda[mitad]) > 0)
26              inicio = mitad + 1;
27          else
28              inicio=fin+2;
29      }
30      while (inicio <= fin);
31      // printf("termino %d  %d  %d\n",inicio, mitad, fin);
32      if (inicio - 1 == fin)
33          puts("No existe");
34      else
35          printf("Existe es el : %d\n", mitad);
36      return(0);
37  }
38
39
```

```c
/* Ejemplo 31.c

Observa como se pasan arrays a funciones. Y como son equivalentes a punteros.
Como por defecto se pasan por referencia pero también se puede hacer análogo a por valor.    */


#include <stdio.h>
#include <stdio.h>
#define TAM 10
int main ()
{
    int media1(int array[], int);
    int media2(int *, int);
    int media3(const int array[], int);

    int array[TAM]={1,2,3,4,5,5};   // el resto a cero

    printf("Normal: %d \n", media1(array, TAM));
    printf("Puntero: %d \n", media2(array, TAM));
    printf("Constante: %d \n", media3(array, TAM));

    return(0);
}

int media3 (const int array[], int n)
{
    int indice;
    int suma;
    for (indice = 0, suma = 0; indice < n; indice++)
        suma += array[indice];
    //array[0]= 3;  // no funciona
    return(suma/n);
}

int media2 (int *pa, int n)
{
    int indice;
    int suma;
    for (indice = 0, suma = 0; indice < n; indice++)
        suma += *(pa + indice);
    return(suma/n);
}

int media1 (int array[], int n)
{
    int indice;
    int suma;
    for (indice = 0, suma = 0; indice < n; indice++)
        suma += array[indice];
    array[0]= 3;  // si funciona
    return(suma/n);
}
```

```c
/* Ejemplo 32.c

Analiza como se hacen las plantillas de los structs.
Como se pueden pasar y devolver a/de funciones.     */


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

struct punto    // esto hace de tipo de dato
{
    char nombre[80];
    int x;
    int y;
} global;

int main ()
{
        struct punto A = {"pepe", 11, 15};  /* en inicializaciÃ³n */
        struct punto B;

        struct punto suma(struct punto, struct punto);   /* prototipo */

    strcpy(B.nombre ,"juan");   /*no se puede hacer = â€œjuanâ€ , es constante */
    B.x=4;                      /* inicializaciÃ³n con asignaciÃ³n */
    B.y=0;

    global=suma(A,B);   /* llamada a funciÃ³n */

    printf("Ahora vale %s %d %d \n", global.nombre, global.x, global.y);

    return(0);
}

/*definiciÃ³n de la funciÃ³n de tipo punto */
struct punto suma (struct punto uno, struct punto dos)
{
    uno.x += dos.x;
    dos.y += dos.y;
    return uno;
}


```

© RMR