CAPÍTULO 12

ESTRUCTURAS ESTÁTICAS DE DATOS Y DEFINICIÓN DE TIPOS

En uno de los primeros temas hablamos largamente de los tipos de dato. Decíamos que un tipo de dato determina un dominio (conjunto de valores posibles) y unos operadores definidos sobre esos valores.

Hasta el momento hemos trabajado con tipos de dato estándar en C. Pero con frecuencia hemos hecho referencia a que se pueden crear otros diversos tipos de dato, más acordes con las necesidades reales de muchos problemas concretos que se abordan con la informática.

Hemos visto, de hecho, ya diferentes tipos de dato a los que por ahora no hemos prestado atención alguna, pues aún no habíamos llegado a este capítulo: tipo de dato ***size_t***, ó ***time_t***: cada vez que nos los hemos encontrado hemos despejado con la sugerencia de que se considerasen, sin más, tipos de dato iguales a ***long***.

En este tema vamos a ver cómo se pueden definir nuevos tipos de dato.

Tipos de dato enumerados

La enumeración es el modo más simple de crear un nuevo tipo de dato. Cuando definimos un tipo de dato enumerado lo que hacemos es definir de forma explícita cada uno de los valores que formarán parte del dominio de ese nuevo tipo de dato: es una definición por extensión.

La sintaxis de creación de un nuevo tipo de dato enumerado es la siguiente:

enum identificador {id_1[, id_2, …, id_N]};

Donde ***enum*** es una de las 32 palabras reservadas de C. Donde *identificador* es el nombre que va a recibir el nuevo tipo de dato. Y donde *id_1*, etc. son los diferentes identificadores de cada uno de los valores del nuevo dominio creado con el nuevo tipo de dato.

Mediante la palabra clave ***enum*** se logran crear tipos de dato que son subconjunto de los tipos de dato ***int***. Los tipos de dato enumerados tienen como dominio un subconjunto del dominio de ***int***. De hecho las variables creadas de tipo ***enum*** son tratadas, en todo momento, como si fuesen de tipo ***int***. Lo que hace ***enum*** es mejorar la legibilidad del programa. Pero a la hora de operar con sus valores, se emplean todos los operadores definidos para ***int***.

Veamos un ejemplo:

```
enum semaforo {verde, amarillo, rojo};
```

Al crear un tipo de dato así, acabamos de definir:

1. Un dominio: tres valores definidos, con los literales "verde", "amarillo" y "rojo". En realidad el ordenador los considera valores 0, 1 y 2.
2. Una ordenación intrínseca a los valores del nuevo dominio: *verde* menor que *amarillo*, y *amarillo* menor que *rojo*.

Acabamos, pues, de definir un conjunto de valores (subconjunto de los enteros), con identificadores propios y únicos, que presenta la propiedad de ordenación.

Luego, en la función que sea necesario utilizar ese tipo de dato, se pueden declarar variables con la siguiente sintaxis:

enum identificador nombre_variable;

En el caso del tipo de dato semáforo, podemos definir en una función una variable de la siguiente forma:

```
enum semaforo cruce;
```

Veamos otro ejemplo

```
enum judo {blanco,amarillo,naranja,verde,azul,marron,negro};

void main(void)
{
      enum judo c;
      printf("Los colores definidos son ... \n");
      for(c = blanco ; c <= negro ; c++)
            printf("%d\t",c);
}
```

La función principal mostrará por pantalla los valores de todos los colores definidos: el menor es el banco, y el mayor el negro. Por pantalla aparecerá la siguiente salida:

```
Los colores definidos son ...
0     1     2     3     4     5     6
```

Dar nombre a los tipos de dato

A lo largo del presente capítulo veremos la forma de crear diferentes estructuras de datos que den lugar a tipos de dato nuevos. Luego, a estos tops de dato se les puede asignar un identificador o un nombre para poder hacer referencia a ellos a la hora de crear nuevas variables.

La palabra clave ***typedef***, de C, permite crear nuevos nombres para los tipos de dato creados. Una vez se ha creado un tipo de dato y se ha

creado el nombre para hacer referencia a él, ya podemos usar ese identificador en la declaración de variables, como si fuese un tipo de dato estándar en C.

En sentido estricto, las sentencias ***typedef*** no crean nuevos tipos de dato: lo que hacen es asignar un identificador definitivo para esos tipos de dato.

La sintaxis para esa creación de identificadores es la siguiente:

typedef tipo nombre_tipo;

Así, se pueden definir los tipos de dato estándar con otros nombres, que quizá convengan por la ubicación en la que se va a hacer uso de esos valores definidos por el tipo de dato. Y así tenemos:

```
typedef unsinged size_t;
typedef long time_t;
```

O podemos nosotros mismos reducir letras:

```
typedef unsigned long int uli;
typedef unsigned short int usi;
typedef signed short int ssi;
typedef signed long int sli;
```

O también:

```
typedef char* CADENA;
```

Y así, a partir de ahora, en todo nuestro programa, nos bastará declarar las variables enteras como uno de esos nuevos cuatro tipos. O declarar una cadena de caracteres como una variable de tipo *CADENA*. Es evidente que con eso no se ha creado un nuevo tipo de dato, sino simplemente un nuevo identificador para un tipo de dato ya existente.

También se puede dar nombre a los nuevos tipos de dato creados. En el ejemplo del tipo de dato ***enum*** llamado *semaforo*, podríamos hacer:

```
typedef enum {verde, amarillo, rojo} semaforo;
```

Y así, el identificador semáforo quedaría definitivamente como nuevo tipo de dato. Luego, en una función que necesitase un tipo de dato de este tipo, ya no diríamos:

```
enum semaforo cruce;
```

Sino simplemente

```
semaforo cruce;
```

Ya que el identificador *semaforo* ya ha quedado como identificador válido de tipo de dato en C.

Estructuras de datos y tipos de dato estructurados

Comenzamos ahora a tratar de la creación de verdaderos nuevos tipos de dato. En C, además de los tipos de dato primitivos, se pueden utilizar otros tipos de dato definidos por el usuario. Son tipos de dato que llamamos **estructurados**, que se construyen mediante componentes de tipos más simples previamente definidos o tipos de dato primitivos, que se denominan elementos de tipo constituyente. Las propiedades que definen un tipo de dato estructurado son el número de componentes que lo forman (que llamaremos cardinalidad), el tipo de dato de los componentes y el modo de referenciar a cada uno de ellos.

Un ejemplo de tipo de dato estructurado ya lo hemos definido y utilizado de hecho: las matrices y los vectores. No hemos considerado esas construcciones como una creación de un nuevo tipo de dato sino como una colección ordenada y homogénea de una cantidad fija de elementos, todos ellos del mismo tipo, y referenciados uno a uno mediante índices.

Pero existe otro modo, en C, de crear un tipo de dato estructurado. Y a ese nos queremos referir cuando decimos que creamos un nuevo tipo de dato, y no solamente una colección ordenada de elementos del mismo tipo. Ese tipo de dato se llama **registro**, y está formado por yuxtaposición de elementos que contienen información relativa a una

misma entidad. Por ejemplo, el tipo de dato asignatura puede tener diferentes elementos, todos ellos relativos a la entidad asignatura, y no todos ellos del mismo tipo. Y así, ese tipo de dato registro que hemos llamado asignatura tendría un elemento que llamaríamos *clave* y que podría ser de tipo ***long***; y otro campo se llamaría *descripción* y sería de tipo ***char****; y un tercer elemento sería el número de *créditos* y sería de tipo ***float***, etc. A cada elemento de un registro se le llama **campo**.

Un registro es un tipo de dato estructurado heterogéneo, donde no todos los elementos (campos) son del mismo tipo. El dominio de este tipo de dato está formado por el producto cartesiano de los diferentes dominios de cada uno de los componentes. Y el modo de referenciar a cada campo dentro del registro es mediante el nombre que se le dé a cada campo.

En C, se dispone de una palabra reservada para la creación de registros: la palabra ***struct***.

Estructuras de datos en C

Una estructura de datos en C es una colección de variables, no necesariamente del mismo tipo, que se referencian con un nombre común. Lo normal será crear estructuras formadas por variables que tengan alguna relación entre sí, de forma que se logra compactar la información, agrupándola de forma cabal. Cada variable de la estructura se llama, en el lenguaje C, elementos de la estructura. Este concepto es equivalente al presentado antes al hablar de campos.

La sintaxis para la creación de estructuras presenta diversas formas. Empecemos viendo una de ellas:

```
struct nombre_estructura
{
        tipo_1 identificador_1;
        tipo_2 identificador_2;
        ...
        tipo_N identificador_N;
```

};

La definición de la estructura termina, como toda sentencia de C, en un punto y coma.

Una vez se ha creado la estructura, y al igual que hacíamos con las uniones, podemos declarar variables del nuevo tipo de dato dentro de cualquier función del programa donde está definida la estructura:

struct nombre_estructura variable_estructura;

Y el modo en que accedemos a cada uno de los elementos (o campos) de la estructura (o registro) será mediante el **operador miembro**, que se escribe con el identificador punto (.):

variable_estructura.identificador_1

Y, por ejemplo, para introducir datos en la estructura haremos:

variable_estructura.identificador_1 = valor_1;

La declaración de una estructura se hace habitualmente fuera de cualquier función, puesto que el tipo de dato trasciende el ámbito de una función concreta. De todas formas, también se puede crear el tipo de dato dentro de la función, cuando ese tipo de dato no va a ser empleado más allá de esa función. En ese caso, quizá no sea necesario siquiera dar un nombre a la estructura, y se pueden crear directamente las variables que deseemos de ese tipo, con la siguiente sintaxis:

```
struct
{
        tipo_1 identificador_1;
        tipo_2 identificador_2;
        ...
        tipo_N identificador_N;
}nombre_variable;
```

Y así queda definida la variable *nombre_variable*, de tipo de dato ***struct***. Evidentemente, esta declaración puede hacerse fuera de cualquier función, de forma que la variable que generemos sea de ámbito global.

Otro modo de generar la estructura y a la vez declarar las primeras variables de ese tipo, será la siguiente sintaxis:

```
struct nombre_estructura
{
        tipo_1 identificador_1;
        tipo_2 identificador_2;
        ...
        tipo_N identificador_N;
}variable_1, ..., variable_N;
```

Y así queda definido el identificador *nombre_estructura* y quedan declaradas las variables *variable_1*, …, *variable_N*, que serán locales o globales según se hay hecho esta declaración en uno u otro ámbito.

Lo que está claro es que si la declaración de la estructura se realiza dentro de una función, entonces únicamente dentro de su ámbito el identificador de la estructura tendrá el significado de tipo de dato, y solamente dentro de esa función se podrán utilizar variables de ese tipo estructurado.

El método más cómodo para la creación de estructuras en C es mediante la combinación de la palabra ***struct*** de la palabra ***typedef***. La sintaxis de esa forma de creación es la siguiente:

```
typedef struct
{
        tipo_1 identificador_1;
        tipo_2 identificador_2;
        ...
        tipo_N identificador_N;
} nombre_estructura;
```

Y así, a partir de este momento, en cualquier lugar del ámbito de esta definición del nuevo tipo de dato, podremos crear variables con la siguiente sintaxis:

```
nombre_estructura nombre_variable;
```

Veamos algún ejemplo: podemos necesitar definir un tipo de dato que podamos luego emplear para realizar operaciones en variable compleja.

Esta estructura, que podríamos llamar *complejo*, tendría la siguiente forma:

```
typedef struct
{
      double real;
      double imag;
}complejo;
```

Y también podríamos definir una serie de operaciones, mediante funciones: por ejemplo, la suma, la resta y el producto de complejos. El programa completo podría tener la siguiente forma:

```
#include <stdio.h>
typedef struct
{
      double real;
      double imag;
}complejo;

complejo sumac(complejo, complejo);
complejo restc(complejo, complejo);
complejo prodc(complejo, complejo);
void mostrar(complejo);

void main (void)
{
      complejo A, B, C;
      printf("Introducción de datos ... \n");
      printf("Parte real de A ......... ");
      scanf("%lf",&A.real);
      printf("Parte imaginaria de A ... ");
      scanf("%lf",&A.imag);
      printf("Parte real de B ......... ");
      scanf("%lf",&B.real);
      printf("Parte imaginaria de B ... ");
      scanf("%lf",&B.imag);
// SUMA ...
      printf("\n\n");
      mostrar(A);
      printf(" + ");
      mostrar(B);
      C = sumac(A,B);
      mostrar(C);

// RESTA ...
      printf("\n\n");
      mostrar(A);
      printf(" - ");
      mostrar(B);
```

```
      C = restc(A,B);
      mostrar(C);

// PRODUCTO ...
      printf("\n\n");
      mostrar(A);
      printf(" * ");
      mostrar(B);
      C = prodc(A,B);
      mostrar(C);
}

complejo sumac(complejo c1, complejo c2)
{
      c1.real += c2.real;
      c1.imag += c2.imag;
      return c1;
}

complejo restc(complejo c1, complejo c2)
{
      c1.real -= c2.real;
      c1.imag -= c2.imag;
      return c1;
}

complejo prodc(complejo c1, complejo c2)
{
      complejo S;
      S.real = c1.real + c2.real; - c1.imag * c2.imag;
      S.imag = c1.real + c2.imag + c1.imag * c2.real;
      return S;

}

void mostrar(complejo X)
{
      printf("(% .2lf%s%.2lf * i) ", X.real, X.imag > 0 ? "
              +" : " " , X.imag);
}
```

Así podemos ir definiendo un nuevo tipo de dato, con un dominio que es el producto cartesiano del dominio de los ***double*** consigo mismo, y con unos operadores definidos mediante funciones.

Las únicas operaciones que se pueden hacer sobre la estructura (aparte de las que podamos definir mediante funciones) son las siguientes: operador dirección (&), porque toda variable, también las estructuradas,

tienen una dirección en la memoria; operador selección (.) mediante el cual podemos acceder a cada uno de los elementos de la estructura; y operador asignación, que sólo puede de forma que los dos extremos de la asignación (tanto el Lvalue como el Rvalue) sean variables objeto del mismo tipo. Por ejemplo, se puede hacer la asignación:

```
complejo A, B;
A.real = 2;
A.imag = 3;
B = A;
```

Y así, las dos variables valen lo mismo: al elemento *real* de *B* se le asigna el valor consignado en la parte *real* de *A*; y al elemento *imag* de *B* se le asigna el valor del elemento *imag* de *A*.

Otro ejemplo de estructura podría ser el que antes hemos iniciado, al hablar de los registros: una estructura para definir un tipo de dato que sirva para el manejo de asignaturas:

```
typedef struct
{
      long clave;
      char descripcion[50];
      float creditos;
}asignatura;
```

Vectores y punteros a estructuras

Una vez hemos creado un nuevo tipo de dato estructurado, no resulta extraño que podamos crear vectores y matrices de este nuevo tipo de dato.

Si, por ejemplo, deseamos hacer un inventario de asignaturas, será lógico que creemos un array de tantas variables *asignatura* como sea necesario.

```
asignatura curricula[100];
```

Y así, tenemos 100 variables del tipo asignatura, distribuidas en la memoria de forma secuencial, una después de la otra. El modo en que

accederemos a cada una de las variables será, como siempre mediante la operatoria de índices: *curricula[i]*. Y si queremos acceder a algún miembro de una variable del tipo estructurado, utilizaremos de nuevo el operador de miembro: *curricula[i].descripción*.

También podemos trabajar con operatoria de punteros. Así como antes hemos hablado de curricula[i], también podemos llegar a esa variable del array con la expresión **(curricula + i)*. De nuevo, todo es igual.

Donde hay un cambio es en el operador de miembro: si trabajamos con operatoria de punteros, el **operador de miembro** ya no es el punto, sino que está formado por los caracteres "->". Si queremos hacer referencia al elemento o campo *descripcion* de una variable del tipo asignatura, la sintaxis será: **(curricula + i)->descripcion*.

Y también podemos trabajar con asignación dinámica de memoria. En ese caso, se declara un puntero del tipo estructurado, y luego se le asigna la memoria reservada mediante la función *malloc*. Si creamos un array de asignaturas en memoria dinámica, un programa de gestión de esas asignaturas podría ser el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
      long clave;
      char descr[50];
      float cred;
}asig;

void main(void)
{
      asig *curr;
      short n, i;
      printf("Indique n° de asignaturas de su CV ... ");
      scanf("%hd",&n);
/* La variable n recoge el número de elementos de tipo
   asignatura que debe tener nuestro array. */
      curr = (asig*)malloc(n * sizeof(asig));
      if(curr == NULL)
      {
            printf("Memoria insuficiente.\n");
            printf("La ejecucion se interrumpira.\n");
```

```
            printf("Pulse una tecla para terminar ... ");
            getchar();
            exit(0);
      }
      for(i = 0 ; i < n ; i++)
      {
            printf("\n\nAsignatura %hd ... \n",i  + 1);
            printf("clave ......... ");
            scanf("%ld",&(curr + i)->clave);
            printf("Descripcion ... ");
            gets((curr + i)->descr);
            printf("creditos ...... ");
            scanf("%f",&(curr + i)->cred);
      }
// Listado ...
      for(i = 0 ; i < n ; i++)
      {
            printf("(%10ld)\t",(curr + i)->clave);
            printf("%s\t",(curr + i)->descr);
            printf("%4.1f creditos\n",(curr + i)->cred);
      }
}
```

Observamos que *(curr + i)* es la dirección de la posición i-ésima del vector *curr*. Es, pues, una dirección. Y *(curr + i)->clave* es el valor del campo *clave* de la variable que está en la posición i-ésima del vector *curr*. Es, pues, un valor: no es una dirección. Y *(curr + i)->descr* es la dirección de la cadena de caracteres que forma el campo *descr* de la variable que está en la posición i-ésima del vector *curr*. Es, pues, una dirección, porque dirección es el campo *descr*: un array de caracteres.

Que accedamos a la variable estructura a través de un puntero o a través de su identificador influye únicamente en el operador de miembro que vayamos a utilizar. Una vez tenemos referenciado a través de la estructura un campo o miembro concreto, éste será tratado como dirección o como valor dependiendo de que el miembro se haya declarado como puntero o como variable de dato.

Anidamiento de estructuras

Podemos definir una estructura que tenga entre sus miembros una variable que sea también de tipo estructura. Por ejemplo:

```
typedef struct
{
      unsigned short dia;
      unsigned short mes;
      unsigned short anyo;
}fecha;

typedef struct
{
      unsigned long clave;
      char descripcion[50];
      double creditos;
      fecha convocatorias[3];
}asignatura;
```

Ahora a la estructura de datos asignatura le hemos añadido un vector de tres elementos para que pueda consignar sus fechas de exámenes en las tres convocatorias. EL ANSI C permite hasta 15 niveles de anidamiento de estructuras.

El modo de llegar a cada campo de la estructura *fecha* es, como siempre, mediante los operadores de miembro. Por ejemplo, si queremos que la primera convocatoria se realice el 15 de enero del presente año, la segunda convocatoria el 21 de junio y la tercera el 1 de septiembre, las órdenes deberán ser:

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

typedef struct
{
      unsigned short dia;
      unsigned short mes;
      unsigned short anyo;
}fecha;

typedef struct
{
      unsigned long clave;
      char descripcion[50];
      double creditos;
      fecha c[3];
}asignatura;

void main(void)
{
```

```
        asignatura asig;
        time_t bloquehoy;
        struct tm *hoy;
        bloquehoy = time(NULL);
        hoy = localtime(&bloquehoy);

        asig.clave = 10102301;
        *asig.descripcion  = '\0';
        strcat(asig.descripcion,"fundamentos de informática");
        asig.creditos = 7.5;

        asig.c[0].dia = 15;
        asig.c[0].mes = 1;
        asig.c[0].anyo = hoy->tm_year - 100;

        asig.c[1].dia = 21;
        asig.c[1].mes = 6;
        asig.c[1].anyo = hoy->tm_year - 100;

        asig.c[2].dia = 1;
        asig.c[2].mes = 9;
        asig.c[2].anyo = hoy->tm_year - 100;

        printf("Asignatura %10ld\n",asig.clave);
        printf("%s\t",asig.descripcion);
        printf("%4.1lf\n", asig.creditos);
        printf("\npriemra convocatoria ... ");
        printf("%2hu-%2hu-%02hu", asig.c[0].dia,
                 asig.c[0].mes,asig.c[0].anyo);
        printf("\nsegunda convocatoria ... ");
        printf("%2hu-%2hu-%02hu", asig.c[1].dia,
                 asig.c[1].mes,asig.c[1].anyo);
        printf("\ntercera convocatoria ... ");
        printf("%2hu-%2hu-%02hu", asig.c[2].dia,
                 asig.c[2].mes,asig.c[2].anyo);
}
```

Hemos asignado a cada uno de los tres elementos del vector *c* los valores de día, mes y año correspondientes a cada una de las tres convocatorias. Hemos utilizado índices de vectores para referenciar cada una de las tres fechas. Podríamos haber trabajado también con operatoria de punteros. Por ejemplo, la referencia al día de la segunda convocatoria es, con operatoria de índices

```
asig.c[1].dia = 21;
```

y mediante operatoria de punteros:

```
(asig.c + 1)->dia = 21;
```

Donde *asig.c* es la dirección del primer elemento del vector *c*.

Y donde *(asig.c + 1)* es la dirección del segundo elemento del vector *c*.

Y donde *(asig.c + 1)->dia* es el valor del campo día de la variable de tipo *fecha* cuya dirección es *(asig.c + 1)*.

Respecto a la función *localtime*, el tipo de dato estructurado *tm*, y sus campos (entre ellos el campo *tm_year*) puede encontrarse abundante información sobre todo ello en la ayuda on line que ofrece cualquier compilador. Son definiciones que vienen recogidas en la biblioteca **time.h** y son estándares de ANSI C.

Tipo de dato ***union***

Además de las estructuras, el lenguaje C permite otra forma de creación de un nuevo tipo de dato: mediante la creación de una unión (que se define mediante la palabra clave en C ***union***: por cierto, con ésta, acabamos de hacer referencia en este manual a la última de las 32 palabras del léxico del lenguaje C).

Una unión es una posición de memoria compartida por dos o más variables diferentes, y en general de distinto tipo. Es una región de memoria que, a lo largo del tiempo, puede contener objetos de diversos tipos. Una unión permite almacenar tipos de dato diferentes en el mismo espacio de memoria. Como las estructuras, las uniones también tienen miembros; pero a diferencia de las estructuras, donde la memoria que ocupan es igual a la suma del tamaño de cada uno de sus campos, la memoria que emplea una variable de tipo unión es la necesaria para el miembro de mayor tamaño dentro de la unión. La unión almacena únicamente uno de los valores definidos en sus miembros.

La sintaxis para la creación de una unión es muy semejante a la empleada para la creación de una estructura:

typedef union
{
tipo_1 identificador_1;
tipo_2 identificador_2;
...
tipo_N identificador_N;
} nombre_union;

O en cualquiera otra de las formas que hemos visto para la creación de estructuras.

Es responsabilidad del programador mantener la coherencia en el uso de esta variable: si la última vez que se asignó un valor a la unión fue sobre un miembro de un determinado tipo, luego, al acceder a la información de la unión, debe hacerse con referencia a un miembro de un tipo de dato adecuado y coherente con el último que se empleó. No tendría sentido almacenar un dato de tipo ***float*** de uno de los campos de la unión y luego querer leerlo a través de un campo de tipo ***char***. El resultado de una operación de este estilo es imprevisible.

Veamos un ejemplo, y comparémoslo con una estructura de definición similar:

```
#include <stdio.h>
#include <string.h>
typedef union
{
      long dni;       // número de dni.
      char ss[30];    // número de la seg. social.
}ident1;

typedef struct
{
      long dni;       // número de dni.
      char ss[30];    // número de la seg. social.
}ident2;

void main(void)
{
      ident1 id1;
      ident2 id2;

      printf("tamaño de la union: %ld\n",sizeof(ident1));
      printf("tamaño de la estructura:%ld\n",sizeof(ident2));

// Datos de la estructura ...
```

```
      id2.dni = 44561098;
      *(id2.ss + 0) = NULL;
      strcat(id2.ss,"12/0324/5431890");
      printf("\nid2.dni = %ld\n",id2.dni);
      printf("id2.ss = %s\n",id2.ss);

// Datos de la unión ...
      *(id1.ss + 0) = NULL;
      strcat(id1.ss,"12/0324/5431890");
      printf("\nid1.dni = %ld (mal)\n",id1.dni);// Mal.
      printf("id1.ss = %s\n",id1.ss);
      id1.dni = 44561098;
      printf("\nid1.dni = %ld\n",id1.dni);
      printf("id1.ss = %s(mal)\n",id1.ss);      // Mal.
}
```

El programa ofrece la siguiente salida por pantalla;

```
tamaño de la union: 30
tamaño de la estructura:34

id2.dni = 44561098
id2.ss = 12/0324/5431890

id1.dni = 808399409 (mal)
id1.ss = 12/0324/5431890

id1.dni = 44561098
id1.ss = ╩═°☻324/5431890 (mal)
```

El tamaño de la estructura es la suma del tamaño de sus miembros. El tamaño de la unión es el tamaño del mayor de sus miembros.

En la estructura se tienen espacios disjuntos para cada miembro: por un lado se almacena el valor ***long*** de la variable *dni* y por otro la cadena de caracteres *ss*. En la unión, si la última asignación se ha realizado sobre la cadena, no tiene sentido que se pretenda obtener el valor del miembro long *dni*; Y si la última asignación se ha realizado sobre el campo *dni*, tampoco tiene sentido pretender leer el valor de la cadena *ss*.

Una buena prueba de que la unión comparte la memoria la tenemos en el ejemplo donde ha quedado como mal uso la impresión del dni. Si vemos el valor impreso (808399409) y lo pasamos a hexadecimal tenemos 302F3231. Y si separamos esa cifra en pares, tendremos

cuatro pares: el 30 (que es el código ASCII del carácter '0'); el 2F (que es el código ASCII del carácter '/'); el 32 (que es el código ASCII del carácter '2'); y el 31, que es el código ASCII del carácter '1'). Y si vemos el valor de los cuatro elementos de la cadena, tenemos que son "12/0". Precisamente los cuatro que antes hemos reconocido en la codificación del mal leído entero. Y es que hemos leído los cuatro primeros caracteres de la cadena como si de un entero se tratase.

En este aspecto, una buena utilidad del mal uso de las uniones (en ese caso no sería mal uso: sería una treta del programador) será el poder obtener el código interno de la información de los valores de tipo ***float***. Veamos como podríamos hacerlo:

```
#include <stdio.h>
typedef union
{
      float fvalor;
      unsigned long lvalor;
}codigo;

void main(void)
{
      unsigned long Test = 0x80000000;
      codigo a;

      printf("Introduzca float del que desea ");
      printf("conocer su codificacion binaria ... ");
      scanf("%f", &a.fvalor);

      while(Test)
      {
            printf("%c",a.lvalor & Test ? '1' : '0');
            Test >>= 1;
      }
}
```

La función principal lee el valor y lo almacena en el campo de tipo ***float***. Pero luego lo lee como si fuese un dato de tipo ***long***. Y si a ese valor ***long*** le aplicamos el operador *and* a nivel de bit (permitido en las variables ***long***, y no permitido en las ***float***) llegamos a poder obtener la codificación interna de los valores en coma flotante. No hemos explicado en este manual la norma IEEE 754 que emplean los PC para codificar

esa clase de valores, pero quien quiera conocerla y cotejarla bien puede hacerlo: la norma está fácilmente accesible; y el programa que acabamos de presentar permite la visualización de esa codificación.

Ejercicios

68. ***Definir un tipo de dato entero, de longitud tan grande como se quiera, y definir también los operadores básicos de ese nuevo tipo de dato mediante la definición y declaración de las funciones que sean necesarias.***

La estructura que define el nuevo tipo de dato podría ser como la que sigue:

```
#define Byte4 32
typedef unsigned long int UINT4;

typedef struct
{
/* número de elementos UINT4 reservados. */
      UINT4 D;
/* número de elementos UINT4 utilizados actualmente. */
      UINT4 T;
/* número de bits significativos. */
      UINT4 B;
/* El número, que tendrá tantos elementos como indique D. */
      UINT4 *N;
}NUMERO;
```

Que tiene cuatro elementos, que servirán para conocer bien las propiedades del número (nuevo tipo de dato entero) y que nos permitirán agilizar luego numerosas operaciones con ellos. El puntero *N* recogerá un array (en asignación dinámica) donde se codificarán los numeros; a este puntero se le asignan tantos elementos enteros largos consecutivos como indique el campo *D*. Y una vez creado el número (reservada la memoria), siempre convendrá mantener actualizado el

valor de *T* y de *B*: el número de elementos enteros largos que realmente se emplean en cada momento para la codificación del entero largo; y el número de bits empleados en la codificación del número actual.

Podemos ahora definir una serie de operadores, empleando funciones. Lo primero será definir la función que asigne memoria al puntero *N* y aquella que ponga a cero el nuevo entero creado. A una la llamamos ***PonerACero***, y a la otra ***CrearNumero***:

```
void PonerACero(NUMERO*n)
{
/* Esta función pone a cero los campos B y T de la variable
NUMERO recibida por referencia. Y deja también a cero todos
los dígitos del número. Considera que el campo T viene
correctamente actualizado. */
      n->B = 0;
      while(n->T) *(n->N + --n->T) = 0x0;
}



#define msg01_001                                                \
"01_001: Error de asignación de memoria en CrearNumero()\n"
void CrearNumero(NUMERO*n)
{
/* Con esta función asignamos una cantidad de memoria a n->N
(tantos elementos UINT4 como indique n->D) */

      n->N = (UINT4*)malloc(n->D * sizeof(UINT4));
      if(n->N == NULL)
      {
            printf(msg01_001);
            exit(1);
      }
      n->T = n->D;
      PonerACero(n);
}
```

Y podemos definir ahora otras funciones, necesarias para trabajar cómodamente en este nuevo tipo de dato. Vamos introduciéndolas una a una:

Función diseñada para copiar el valor de un *NUMERO* en otro *NUMERO*. No bastaría hacer copia mediante el asignación dirección, porque en ese caso se copiaría la dirección de *N* de uno a otro pero lo

que nos interesa es que sean variables diferentes con direcciones diferentes, que codifiquen el mismo número; y se copiarían los valores del campo *D*, y eso no nos interesa porque el campo *D* indica cuántos elemento de tipo *UINT4* se han reservado en el puntero, que depende de cada *NUMERO*. La función queda:

```
#define msg01_002                                                \
"01_002: No se puede hacer copia de la variable. Error en
CopiarNumero()\n"
void CopiarNumero(NUMERO*original,NUMERO*copia)
{
/* Esta función copia el valor de número grande original->N
en copia->N. Previo a esta operación, verifica que el tamaño
del original no supera la capacidad (copia->D) de la copia.*/

      UINT4 c;
      if(original->T > copia->D)
      {
            printf(msg01_002);
            exit(1);
      }
/* Si el original y la copia no son la misma variable ... */
      if(original->N != copia->N)
      {
            PonerACero(copia);
            for(c = 0 ; c < original->T ; c++)
            *(copia->N + c) = *(original->N + c);
            copia->T = original->T;
            copia->B = original->B;
      }
}
```

Función que actualice los valores de los campos B y T:

```
void longitud(NUMERO*n)
{
/* De entrada se le supone el tamaño de la dimensión ----- */
      n->T = n->D;
      while(*(n->N + n->T - 1) == 0 && n->T != 0)
            (n->T)--;
/* Una vez tenemos determinado el número de elementos UINT4
que forman el número, vamos ahora a calcular el número de
bits a partir del más significativo.  */
      n->B = Byte4 * n->T;
      if(n->B)
      {
      UINT4 Test = 0x80000000;
      while(!(*(n->N + n->T - 1) & Test))
      {
```

```
            Test >>= 1;
            n->B--;
      }
      }
}
```

La siguiente función lo que hace es intercambiar los valores de dos variables NUMERO:

```
#define msg01_003                                                    \
"01_003: No se pueden intercambiar valores. Error en
inv_v()\n"
void inv_v(NUMERO*n1,NUMERO*n2)
{
/* Si los dos tamaños de los arrays son iguales, entonces
intercambiamos los punteros de los números. */
      if(n1->D == n2->D)
      {
            UINT4*aux;
            aux = n1->N;
            n1->N = n2->N;
            n2->N = aux;
      }
/* En caso contrario ... */
      else
      {
            UINT4 L;
            L  = (n1->T >= n2->T) ? n1->T : n2->T;
            if(n1->T > n2->D || n2->T > n1->D)
            {
                  printf(msg01_003);
                  exit(1);
            }
/* Intercambiamos cada uno de los dígitos */
            while(L)
            {
                  L--;
                  *(n1->N + L) ^= *(n2->N + L);
                  *(n2->N + L) ^= *(n1->N + L);
                  *(n1->N + L) ^= *(n2->N + L);
            }
      }
/* Intercambiamos los valores de los tamaños. */
      n1->T ^= n2->T;
      n2->T ^= n1->T;
      n1->T ^= n2->T;
/* Intercambiamos los valores del número de bits. */
      n1->B ^= n2->B;
      n2->B ^= n1->B;
      n1->B ^= n2->B;
```

```
/* Evidentemente, no podemos intercambiar las dimensiones en
que han sido definidos cada uno de los dos números grandes.*/
}
```

Y la siguiente función determina cuál de los dos *NUMERO's* que se reciben como parámetro es mayor y cuál menor:

```
typedef signed short int SINY2;
typedef signed long int SINT4;
SINT2 orden(NUMERO*n1,NUMERO*n2)
{
/*    Devuelve +1 si n1 > n2.
      -1 si n1 < n2.
      0 si n1 == n2. */
      SINT4 c;
      if(n1->B < n2->B)
            return -1; /* Es decir, n1 < n2. */
      if(n1->B > n2->B)
            return +1; /* Es decir, n1 > n2. */
/* LLegados aquí tenemos que *n1->T es igual a n2->T... */
      for(c = n1->T - 1 ; c >= 0 ; c--)
      {
            if(*(n1->N + c) < *(n2->N + c))
                  return -1; /* Es decir, n1 < n2. */
            if(*(n1->N + c) > *(n2->N + c))
                  return +1; /* Es decir, n1 > n2. */
      }
      return 0;
}
```

Podemos seguir definiendo funciones auxiliares, como desplazamiento a izquierda o desplazamiento a derecha, etc. Lo dejamos como ejercicio para resolver. También queda pendiente trabajar las operaciones aritméticas de suma, resta, producto, cociente, módulo, etc. No es objeto del libro mostrar todo ese código. Hemos mostrado el que precede porque está formado en su mayor parte por funciones sencillas y fáciles de implementar y porque todas ellas trabajan con estructuras.