Estructuras

Para definir una estructura usamos el siguiente formato:

```
struct nombre_de_la_estructura {
    campos de estructura;
    };
```

NOTA: Es importante no olvidar el ';' del final, si no a veces se obtienen errores extraños.
Para nuestro ejemplo podemos crear una estructura en la que almacenaremos los datos de cada persona. Vamos a crear una declaración de estructura llamada amigo:

```
struct estructura_amigo {
    char nombre[30];
    char apellido[40];
    char telefono[10];
    char edad;
    };
```

A cada elemento de esta estructura (nombre, apellido, teléfono) se le llama campo o miembro. (NOTA: declaramos edad como char porque no conozco a nadie con más de 127 años).
Ahora ya tenemos definida la estructura, pero aun no podemos usarla. Necesitamos declarar una variable con esa estructura.

```
struct estructura_amigo amigo;
```

Ahora la variable amigo es de tipo estructura_amigo. Para acceder al nombre de amigo usamos: amigo.nombre. Vamos a ver un ejemplo de aplicación de esta estructura. (NOTA: En el siguiente ejemplo los datos no se guardan en disco así que cuando acaba la ejecución del programa se pierden).

```
#include <stdio.h>

struct estructura_amigo {/* Definimos la estructura estructura_amigo */
    char nombre[30];
    char apellido[40];
    char telefono[10];
    char edad;
    };
struct estructura_amigo amigo;

int main()
   {
   printf( "Escribe el nombre del amigo: " );
   fflush( stdout );
   scanf( "%s", &amigo.nombre );
   printf( "Escribe el apellido del amigo: " );
   fflush( stdout );
```

```
    scanf( "%s", &amigo.apellido );
    printf( "Escribe el número de teléfono del amigo: " );
    fflush( stdout );
    scanf( "%s", &amigo.telefono );
    printf( "El amigo %s %s tiene el número: %s.\n", amigo.nombre,
          amigo.apellido, amigo.telefono );
    }
```

Este ejemplo estaría mejor usando gets que scanf, ya que puede haber nombres compuestos que scanf no los tomaría por los espacios.
Se podría haber declarado directamente la variable amigo:

```
struct estructura_amigo {
     char nombre[30];
     char apellido[40];
     char telefono[10];
     } amigo;
```

Veamos otro ejemplo de tipo estructurado :

```
struct Persona {
        char nombre [40];
        char apellidos [80];
        long telefono;
        long dni;
        char sexo;
};
struct Persona usuario;

main ()
{
        usuario.dni = 43742809;
        strcpy ( usuario.apellidos, "Santana Melián" );
        strcpy ( usuario.nombre, "Jacinto Lerante" );
        usuario.telefono = 908330033;
        usuario.sexo = 'V';
}
```

Si hay campos del mismo tipo, se pueden declarar en la misma línea, separándolos por comas. También se permite omitir el tipo de la estructura (estructura anónima):

```
struct {
  char nombre [8];
  char extension [3];
} nombre_fichero;
```

En este caso, no se puede crear nuevas variables de ese tipo. No se recomienda usar este tipo de declaraciones.

Definición de tipos: typedef.
Se puede dar un nombre nuevo a cualquier tipo de datos mediante typedef. La sintaxis es:

```
typedef declaración;
```

donde declaración tiene la forma de una declaración de variable, sólo que se está definiendo un tipo de datos.

```
typedef long pareja [2];
```

define un tipo pareja que se puede usar en declaraciones de variables:

```
pareja p;
```

es equivalente a

```
long p [2];
```

Ejemplos de typedef con estructuras

```
typedef struct Persona PERSONA;
PERSONA dato;       /* igual que struct Persona dato; */
```

Un uso típico es la redefinición de tipos estructurados:

```
typedef struct  /* estructura anónima */
{
  char nombre[80];
  char sexo;
  int edad;
} Persona;              /* se declara el tipo Persona */
...
Persona p;
...
p.edad = 44;
```

Arreglos de Estructuras

Con C también se pueden tener arreglos de estructuras:

```
typedef struct direc
{
   char nombre[30];
   char calle[40];
   char ciudad[20];
   char estado[3];
   unsigned int codigo;
} info_direc;

info_direc artistas[1000];
```

por lo anterior, artistas tiene 1000 elementos del tipo info_direc. Lo anterior podría ser accesado de la siguiente forma:

```
artistas[50].codigo=22222;
```

Supongamos ahora que queremos guardar la información de varios amigos. Con una variable de estructura sólo podemos guardar los datos de uno. Para manejar los datos de más gente (al conjunto de todos los datos de cada persona se les llama REGISTRO) necesitamos declarar arrays de estructuras.

¿Cómo se hace esto? Siguiendo nuestro ejemplo vamos a crear un array de ELEMENTOS elementos:

```
struct estructura_amigo amigo[ELEMENTOS];
```

Ahora necesitamos saber cómo acceder a cada elemento del array. La variable definida es amigo, por lo tanto para acceder al primer elemento usaremos amigo[0] y a su miembro nombre: amigo[0].nombre. Veámoslo en un ejemplo en el que se supone que tenemos que meter los datos de tres amigos:

```
#include <stdio.h>
#define ELEMENTOS      3
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      char telefono[10];
      int edad;
      };
struct estructura_amigo amigo[ELEMENTOS];

int main()
    {
    int num_amigo;
    for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
        printf( "\nDatos del amigo número %i:\n", num_amigo+1 );
        printf( "Nombre: " ); fflush( stdout );
        gets(amigo[num_amigo].nombre);
        printf( "Apellido: " ); fflush( stdout );
        gets(amigo[num_amigo].apellido);
        printf( "Teléfono: " ); fflush( stdout );
        gets(amigo[num_amigo].telefono);
        printf( "Edad: " ); fflush( stdout );
        scanf( "%i", &amigo[num_amigo].edad );
        while(getchar()!='\n');
        }
    /* Ahora imprimimos sus datos */
    for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ )
        {
        printf( "El amigo %s ", amigo[num_amigo].nombre );
        printf( "%s tiene ", amigo[num_amigo].apellido );
        printf( "%i años ", amigo[num_amigo].edad );
        printf( "y su teléfono es el %s.\n" , amigo[num_amigo].telefono );
        }
    }
```

IMPORTANTE: Quizás alguien se pregunte qué hace la línea while(getchar()!='\n');. Esta línea se usa para vaciar el buffer de entrada. Para más información consulta Qué son los buffer y cómo funcionan.

Inicializar una estructura

A las estructuras se les pueden dar valores iniciales de manera análoga a como hacíamos con los arrays. Primero tenemos que definir la estructura y luego cuando declaramos una variable como estructura le damos el valor inicial que queramos. Recordemos que esto no es en absoluto necesario. Para la estructura que hemos definido antes sería por ejemplo:

```
struct estructura_amigo amigo = {
```

```
        "Juanjo",
        "Lopez",
        "592-0483",
        30
        };
```

NOTA: En algunos compiladores es posible que se exija poner antes de struct la palabra static.
Por supuesto hemos de meter en cada campo el tipo de datos correcto. La definición de la estructura es:

```
struct estructura_amigo {
    char nombre[30];
    char apellido[40];
    char telefono[10];
    int edad;
    };
```

por lo tanto el nombre ("Juanjo") debe ser una cadena de no más de 29 letras (recordemos que hay que reservar un espacio para el símbolo '\0'), el apellido ("Lopez") una cadena de menos de 39, el teléfono una de 9 y la edad debe ser de tipo char.
Vamos a ver la inicialización de estructuras en acción:

```
#include <stdio.h>

struct estructura_amigo {
    char nombre[30];
    char apellido[40];
    char telefono[10];
    int edad;
    };
struct estructura_amigo amigo = {
    "Juanjo",
    "Lopez",
    "592-0483",
    30
    };

int main()
    {
    printf( "%s tiene ", amigo.apellido );
    printf( "%i años ", amigo.edad );
    printf( "y su teléfono es el %s.\n" , amigo.telefono );    }
```

También se puede inicializar un array de estructuras de la forma siguiente:

```
struct estructura_amigo amigo[] =
    {
    "Juanjo", "Lopez", "504-4342", 30,
    "Marcos", "Gamindez", "405-4823", 42,
    "Ana", "Martinez", "533-5694", 20
    };
```

En este ejemplo cada línea es un registro. Como sucedía en los arrays si damos valores iniciales al array de estructuras no hace falta indicar cuántos elementos va a tener. En este caso la matriz tiene 3 elementos, que son los que le hemos pasado.

Estructuras dentro de estructuras (Anidadas)

Es posible crear estructuras que tengan como miembros otras estructuras. Esto tiene diversas utilidades, por ejemplo tener la estructura de datos más ordenada. Imaginemos la siguiente situación: una tienda de música quiere hacer un programa para el inventario de los discos, cintas y cd's que tienen. Para cada título quiere conocer las existencias en cada soporte (cinta, disco, cd), y los datos del proveedor (el que le vende ese disco). Podría pensar en una estructura así:

```
struct inventario {
      char titulo[30];
      char autor[40];
      int existencias_discos;
      int existencias_cintas;
      int existencias_cd;
      char nombre_proveedor[40];
      char telefono_proveedor[10];
      char direccion_proveedor[100];
      };
```

Sin embargo utilizando estructuras anidadas se podría hacer de esta otra forma más ordenada:

```
struct estruc_existencias {
     int discos;
     int cintas;
     int cd;
     };

struct estruc_proveedor {
     char nombre_proveedor[40];
     char telefono_proveedor[10];
     char direccion_proveedor[100];
     };
struct estruc_inventario {
      char titulo[30];
      char autor[40];
      struct estruc_existencias existencias;
      struct estruc_proveedor proveedor;
      } inventario;
```

Ahora para acceder al número de cd de cierto título usaríamos lo siguiente:

```
inventario.existencias.cd
```

y para acceder al nombre del proveedor:

```
inventario.proveedor.nombre
```

Paso de estructuras a funciones

Las estructuras se pueden pasar directamente a una función igual que hacíamos con las variables. Por supuesto en la definición de la función debemos indicar el tipo de argumento que usamos:

int nombre_función (struct nombre_de_la_estructura nombre_de_la variable_estructura)

En el ejemplo siguiente se usa una función llamada suma que calcula cual será la edad 20 años más tarde (simplemente suma 20 a la edad). Esta función toma como argumento la variable estructura arg_amigo. Cuando se ejecuta el programa llamamos a suma desde main y en esta variable se copia el contenido de la variable amigo.

Esta función devuelve un valor entero (porque está declarada como int) y el valor que devuelve (mediante return) es la suma.

```
#include <stdio.h>
struct estructura_amigo {
      char nombre[30];
      char apellido[40];
      char telefono[10];
      int edad;
      };
int suma( struct estructura_amigo arg_amigo )
   {
   return arg_amigo.edad+20;
   }
int main()
    {
    struct estructura_amigo amigo = {
          "Juanjo",
          "López",
          "592-0483",
          30
     };
    printf( "%s tiene ", amigo.apellido );
    printf( "%i años ", amigo.edad );
    printf( "y dentro de 20 años tendrá %i.\n", suma(amigo) );
    system( "PAUSE" );
    }
```

Si dentro de la función suma hubiésemos cambiado algún valor de la estructura, dado que es una copia no hubiera afectado a la variable amigo de main. Es decir, si dentro de 'suma' hacemos arg_amigo.edad = 20; el valor de arg_amigo cambiará, pero el de amigo seguirá siendo 30.

También se pueden pasar estructuras mediante punteros o se puede pasar simplemente un miembro (o campo) de la estructura.

Si usamos punteros para pasar estructuras como argumentos habrá que hacer unos cambios al código anterior (en negrita y subrrayado):

```
#include <stdio.h>
...
int suma( struct estructura_amigo *arg_amigo )
   {
```

```
    return arg_amigo->edad+20;
    }
int main()
    {
    ...
    printf( "%s tiene ", amigo.apellido );
    printf( "%i años ", amigo.edad );
    printf( "y dentro de 20 años tendrá %i.\n", suma(&amigo) );
    system( "PAUSE" );
    }
```

Lo primero será indicar a la función suma que lo que va a recibir es un puntero, para eso ponemos el * (asterisco). Segundo, como dentro de la función suma usamos un puntero a estructura y no una variable estructura debemos cambiar el '.' (punto) por el '->'. Tercero, dentro de main cuando llamamos a suma debemos pasar la dirección de amigo, no su valor, por lo tanto debemos poner '&' delante de amigo.
Si usamos punteros a estructuras corremos el riesgo (o tenemos la ventaja, según cómo se mire) de poder cambiar los datos de la estructura de la variable amigo de main.

Pasar sólo miembros de la estructura

Otra posibilidad es no pasar toda la estructura a la función sino tan sólo los miembros que sean necesarios. Los ejemplos anteriores serían más correctos usando esta tercera opción, ya que sólo usamos el miembro 'edad':

```
int suma( char edad )
    {
    return edad+20;
    }
int main()
    {
    printf( "%s tiene ", amigo.apellido );
    printf( "%i años ", amigo.edad );
    printf( "y dentro de 20 años tendrá %i.\n", suma(amigo.edad) );
    }
```

Por supuesto a la función suma hay que indicarle que va a recibir una variable tipo char (amigo->edad es de tipo char).