Tomado de:
http://cfievalladolid2.net/tecno/cyr_01/control/lengua_C/estructuras.htm

Estructuras

Introducción

Supongamos que queremos hacer una agenda con los números de teléfono de nuestros amigos. Necesitaríamos un *array* de cadenas para almacenar sus nombres, otro para sus apellidos y otro para sus números de teléfono. Esto puede hacer que el programa quede desordenado y difícil de seguir. Y aquí es donde presenta ventajas el uso de las 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. En caso contrario se obtienen, a veces, 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];
        int edad;
        };
```

A cada elemento de esta estructura (`nombre`, `apellido`, `teléfono`, `edad`) se le denomina **campo** o **miembro**, y al conjunto de los datos se le denomina **registro**.

Aunque tenemos definida la estructura, todavía 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` (al operador `'.'` le denominamos selector directo de miembro) Vamos a ver un ejemplo de aplicación de esta estructura.

```
#include <stdio.h>

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

struct estructura_amigo amigo; /* Declaramos la variable amigo con esa
estructura */

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

Este ejemplo sería más correcto usando `gets` que `scanf`, ya que puede haber nombres compuestos que `scanf` no cogería por los espacios.

Se podría haber declarado directamente la variable `amigo`:

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

Arrays de estructuras

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 necesitamos declarar *arrays* de estructuras.

Siguiendo con el ejemplo anterior vamos a crear un *array* de registros:

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

Ahora necesitamos saber cómo acceder a cada elemento del *array*. La variable definida es `amigo`; por tanto, para acceder al primer elemento usaremos `amigo[0]`, y a su nombre: `amigo[0].nombre`. Veámoslo con un ejemplo en el que se supone que tenemos que introducir 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];

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: " );
                gets(amigo[num_amigo].nombre);
                printf( "Apellido: " );
                gets(amigo[num_amigo].apellido);
                printf( "Teléfono: " );
                gets(amigo[num_amigo].telefono);
                printf( "Edad: " );
                scanf( "%i", &amigo[num_amigo].edad );

                while(getchar()!= '\n');       /* Vacía el buffer de
entrada */
        }

        /* Impresión de los datos */
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ ) {
```

```
                printf( "Mi 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 );
        }
}
```

Obsérvese la línea `while(getchar()!= '\n')`. Esta línea se usa para vaciar el buffer de entrada (véase Cómo funcionan los *buffer*).

Inicialización de una estructura

A las estructuras se les pueden dar valores iniciales de manera análoga a como se hace con los *arrays*. Primero debemos definir la estructura y después, cuando declaramos una variable como estructura, le damos el valor inicial que queremos. Por ejemplo, para la estructura que hemos definido antes podría ser:

```
struct estructura_amigo amigo = {
        "Juanjo",
        "López",
        "983403367",
        30
        };
```

Por supuesto, hemos de introducir en cada campo el tipo de datos correcto. Por lo tanto el nombre ("Juanjo") debe ser una cadena de no más de 29 caracteres (recordemos que hay que reservar un espacio para el símbolo `'\0'`), el apellido ("López") una cadena de menos de 39, el teléfono una de 9 y la edad debe ser de tipo `int`.

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",
        "López",
        "983403367",
        30
```

```
        };

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", "López", "983403367", 30,
        "Marcos", "Ramos", "983427890", 42,
        "Ana", "Martínez", "983254789", 20
        };
```

En este ejemplo cada línea es un registro. Como sucedía en los *arrays*, si asignamos 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 introducido.

Punteros a estructuras

También se pueden usar punteros con estructuras. Antes de nada, hay que definir la estructura de igual forma que hacíamos antes. La diferencia está en que al declarar la variable de tipo estructura debemos anteponerle el operador `'*'` para indicarle que es un puntero.

Veamos un ejemplo. Este programa utiliza un puntero para acceder a la información de la estructura:

```
#include <stdio.h>

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

struct estructura_amigo amigo = {
        "Juanjo",
        "López",
        "983403367",
        30
        };
```

```
main()
{
        struct estructura_amigo *p_amigo;

        p_amigo = &amigo;

        printf( "%s tiene ", p_amigo->apellido );
        printf( "%i años ", p_amigo->edad );
        printf( "y su teléfono es el %s.\n" , p_amigo->telefono );
}
```

Hasta la definición del puntero `p_amigo` todo resulta igual que antes. Éste es un puntero a la estructura `estructura_amigo`. Dado que es un puntero, debemos indicarle a dónde debe apuntar; en este caso hacemos que apunte a la variable `amigo`: `p_amigo = &amigo;` (recordemos que el operador **`&`** significa 'dame la dirección donde está almacenado...').

Ahora queremos acceder a cada campo de la estructura. Antes lo hacíamos usando el operador `'.'`, pero, como muestra el ejemplo, si se trabaja con punteros se debe usar el operador `'->'`. Este operador, denominado selector indirecto de miembro, viene a significar algo así como: "dame acceso al miembro ... del puntero ...".

Vamos a ver ahora con un ejemplo cómo introducir datos en las estructuras:

```
#include <stdio.h>

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

main()
{
        struct estructura_amigo *p_amigo;

        p_amigo = &amigo;

        /* Introducimos los datos mediante punteros */
        printf("Nombre: ");
        gets(p_amigo->nombre);
        printf("Apellido: ");
        gets(p_amigo->apellido);
        printf("Edad: ");
        scanf( "%i", &p_amigo->edad );

        /* Mostramos los datos */
        printf( "Mi amigo %s ", p_amigo->nombre );
        printf( "%s tiene ", p_amigo->apellido );
        printf( "%i años.\n", p_amigo->edad );
}
```

NOTA: `p_amigo` es un puntero que apunta a la estructura `amigo`. Sin embargo, `p_amigo->edad` es una variable de tipo `int`. Por eso al usar el `scanf` debemos anteponer `&`.

Punteros a *arrays* de estructuras

Por supuesto, también se pueden usar punteros con *arrays* de estructuras. La forma de trabajar es la misma, aunque hay que asegurarse de que el puntero inicialmente apunte al primer elemento, después saltar al siguiente, etc., hasta llegar al último.

```
#include <stdio.h>

#define ELEMENTOS      3

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

struct estructura_amigo amigo[] = {
        "Juanjo", "López", "983403367", 30,
        "Marcos", "Ramos", "983427890", 42,
        "Ana", "Martínez", "983254789", 20
        };

main()
{
        struct estructura_amigo *p_amigo;
        int num_amigo;

        p_amigo = amigo;       /* apunta al primer elemento del array
*/

        /* Imprimimos los datos */
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ ) {

                printf( "Mi amigo %s ", p_amigo->nombre );
                printf( "%s tiene ", p_amigo->apellido );
                printf( "%i años ", p_amigo->edad );
                printf( "y su teléfono es el %s.\n" , p_amigo-
>telefono );

                /* saltamos al siguiente elemento */
                p_amigo++;
        }
}
```

En vez de `p_amigo = amigo;` se podía usar la forma `p_amigo = &amigo[0]`, es decir, que apunte al primer elemento (el elemento 0) del *array*. La primera forma es más frecuente pero la segunda, quizás, indica más claramente al principiante lo que se pretende.

Ahora veamos el ejemplo anterior de cómo introducir datos en un array de estructuras mediante punteros:

```
#include <stdio.h>

#define ELEMENTOS      3

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

main()
{
        struct estructura_amigo amigo[ELEMENTOS], *p_amigo;
        int num_amigo;

        p_amigo = amigo;       /* apunta al primer elemento del array
*/

        /* Introducimos los datos mediante punteros */
        for( num_amigo=0; num_amigo<ELEMENTOS; num_amigo++ ) {
                printf( "\nDatos del amigo %i:\n", num_amigo+1 );

                printf( "Nombre: " );
                gets(amigo[num_amigo].nombre);
                printf( "Apellido: " );
                gets(amigo[num_amigo].apellido);
                printf( "Teléfono: " );
                gets(amigo[num_amigo].telefono);
                printf( "Edad: " );
                scanf( "%i", &amigo[num_amigo].edad );

                while(getchar()!= '\n');       /* Vacía el buffer de
entrada */

                p_amigo++;                     /* Siguiente elemento */

        }

        /* Imprimimos los datos */
        p_amigo = amigo;

        for( num amigo=0; num amigo<ELEMENTOS; num amigo++ ) {

                printf( "Mi amigo %s ", p_amigo->nombre );
                printf( "%s tiene ", p_amigo->apellido );
                printf( "%i años ", p_amigo->edad );
                printf( "y su teléfono es el %s.\n" , p_amigo-
```

```
>telefono );

                p_amigo++;
        }
}
```

Es importante no olvidar que al terminar el primer bucle `for` el puntero `p_amigo` apunta al último elemento del *array* de estructuras. Para mostrar los datos debemos hacer que vuelva a apuntar al primer elemento, por eso usamos de nuevo `p_amigo = amigo`;.

Paso de estructuras a funciones

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

```
tipo 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`.

```
#include <stdio.h>

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

struct estructura_amigo amigo = {
        "Juanjo",
        "López",
        "983403367",
        30
        };

int suma( struct estructura_amigo );

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

```
}

int suma( struct estructura_amigo arg_amigo )
{
        return arg_amigo.edad + 20;
}
```

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, por ejemplo, `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):

```
#include <stdio.h>

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

struct estructura_amigo amigo = {
        "Juanjo",
        "López",
        "983403367",
        30
        };

int suma( struct estructura_amigo * );

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

int suma( struct estructura_amigo *arg_amigo )
{
        return arg_amigo->edad + 20;
}
```

Lo primero será indicar a la función suma que va a recibir es puntero, por lo cual ponemos `*` a `arg_amigo`. Segundo, como dentro de la función suma usamos un puntero a estructura y no una variable estructura, debemos cambiar el `'.'` por el `'->'`. Tercero, dentro de `main`, cuando llamamos a `suma`, debemos pasar la dirección de `amigo`, no su valor, por lo tanto debemos anteponer `'&'`.

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`.

Paso de miembros de una estructura a funciones

Otra posibilidad es no pasar toda la estructura a la función, sino tan sólo los miembros que sean necesarios. El ejemplo anterior sería más correcto usando esta tercera opción, ya que sólo usamos el miembro `edad`:

```
int suma( int );

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

int suma( int edad )
{
        return edad + 20;
}
```

Estructuras dentro de estructuras (estructuras anidadas)

Es posible crear estructuras que tengan como miembros otras estructuras. Esto tiene diversas utilidades, por ejemplo contar con estructuras de datos más ordenadas. Imaginemos la siguiente situación: una tienda de música quiere hacer un programa para el inventario de los discos, cintas y cd's. Para cada título se quiere conocer las existencias en cada soporte (cinta, disco, cd), y los datos del proveedor. Podría pensarse 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 est_existencias {
        int discos;
        int cintas;
        int cd;
        };

struct est_proveedor {
        char nombre_proveedor[40];
        char telefono_proveedor[10];
        char direccion_proveedor[100];
        };

struct est_inventario {
        char titulo[30];
        char autor[40];
        struct est_existencias existencias;
        struct est_proveedor proveedor;
        } inventario;
```

Ahora, para acceder al número de cd de cierto título usaríamos:

```
        inventario.existencias.cd
```

y para acceder al nombre del proveedor:

```
        inventario.proveedor.nombre
```