Capítulo 8.
Estructuras

Índice del capítulo

1. IntroducciÓn

Hasta el momento hemos visto cómo se pueden guardar datos en variables, arrays o cadenas de caracteres. En una variable se guarda un tipo de dato y en los arrays, por ejemplo, se guardan muchos datos del mismo tipo. De este modo a menudo superamos los problemas que se nos puedan presentar a la hora de procesar distintos tipos de datos. Pero hay ocasiones en las cuales, utilizando el mismo nombre, deseamos almacenar información que contenga tipos de datos diferentes. En estos casos las formas de guardar información estudiadas hasta el momento (variables, arrays, cadenas de caracteres) no son suficientes.

Supongamos que queremos guardar información relativa a un trabajador de una empresa. Por ejemplo *el nombre del director* (cadena de caracteres), *el número de departamento* (quizás un número entero), *el sueldo* (número real), *su nombre* (cadena de caracteres), etc. Con el conocimiento que tenemos hasta el momento, deberíamos definir para cada dato un tipo de variable, o array y en ningún caso nos podríamos referir al conjunto de la información como un único elemento.

Para poder hacer frente a este problema el lenguaje C nos ofrece un tipo de dato especial: *la estructura*.

Una estructura es un tipo de datos especial utilizada para guardar información que contiene diferentes tipos de datos. La utilización de estructuras en el lenguaje C requiere el uso de la palabra clave **struct.**

2. Objetivos

El objetivo de este capítulo es presentar un nuevo tipo de datos denominado ***estructura (struct)*** e iniciar al alumno en su utilización. En concreto estos son los objetiovos que se pretenden conseguir:

- Introducir el concepto de estructura.
- Conocer cómo se declaran las estructuras de datos en C.
- Aprender a manipular los datos almacenados en una estructura.
- Estudiar la utilización de punteros aplicada a estructuras de datos.
- Analizar la utilización combinada de diversos tipos de datos: arrays y estructuras fundamentalmente.
- Realizar programas que manipulen estructuras de datos.

3. DeclaraciÓn de estructuras

Antes de utilizar una estructura hay que declararla. Cuando declaramos una estructura creamos un nuevo tipo de dato. Por ejemplo,

```
struct mi_estructura
{
        int num;
        char car;
};
```

cuando declaramos esta estructura definimos el nuevo tipo de dato **mi_estructura** . Las variables que posean este tipo de dato están compuestas por dos elementos: el número entero denominado *num* y el carácter denominado *car*.

Se debe tener en cuenta que al realizar esta declaración no reservamos memoria. Tan sólo le señalamos al compilador cómo es este nuevo tipo de dato.

Por lo tanto *podemos decir que la estructura es un tipo de dato especial que define el programador.*

3.1. Declaración de una variable estructura

Después de definir la estructura podemos declarar las variables que contengan ese nuevo tipo de dato. Siguiendo el ejemplo definiremos la variable **estruct_1** .

```
struct mi_estructura estruct_1;
```

Una vez realizada esta declaración, en memoria se reserva espacio suficiente para la variable estructura estruct_1 la cual necesita 3 bytes en este caso. Tal y como se ve en este ejemplo, podemos definir muchas variables estructura que contengan el mismo tipo de datos.

```
struct mi_estructura estruct_1;
struct mi_estructura estruct_2;
```

3.2. LECTURA Y ESCRITURA DE COMPONENTES DE UNA ESTRUCTURA

Una vez declarada una variable tipo estructura debemos ser capaces de introducir información y de leer la información almacenada en dicha estructura. ¿Cómo leemos el contenido del elemento de una estructura? o ¿cómo se introduce un dato en un elemento de una estructura? Para ello se utiliza el operador "punto"•.

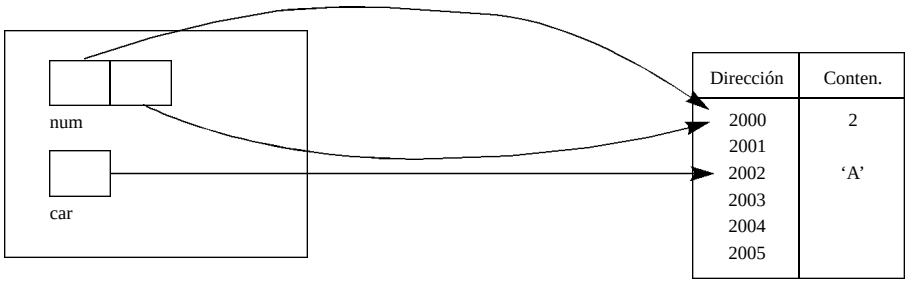
A continuación vamos a examinar el manejo de dicho operador "punto"•. En este ejemplo definimos un nuevo tipo de dato mediante la estructura denominada *sencillo*, que nos posibilita almacenar un número entero y un carácter.

```
#include <stdio.h>

void main (void)
{
        struct sencillo
        {
                int num;
                char car;
        };

        struct sencillo mi_variable;

        mi_variable.num = 2;
        mi_variable.car = 'A';

        printf ("Componentes de la estructura : %d, %c\n",
        mi_variable.num,
        mi_variable.car);
}
```

cuando declaramos la estructura *mi_variable* (*struct sencillo mi_variable*), se reserva en la memoria espacio para guardar un carácter y un número entero.

Cuando realizamos la asignación *mi_variable.num = 2*, se guarda el valor 2 en la variable para guardar números enteros declarada mediante esta estructura.

Cuando realizamos la asignación *mi_variable.car = 'A'*, se guarda el valor 'A' en la variable para guardar caracteres declarada mediante esta estructura.



Variable estructura mi_variable *que contiene el tipo de dato sencillo.*

3.3. Declaración conjunta de Estructuras y Variables

Cuando realizamos la declaración de la estructura también nos es posible definir las variables que van a almacenar ese nuevo tipo de dato, tal y como puede observarse en el siguiente ejemplo:

```
struct sencillo
{
        int num;
        char car;
} sen_1, sen_2;
```

En este ejemplo, declaramos las variables *sen_1* y *sen_2*, las dos son variables estructura que almacenarán el tipo de dato *sencillo.*

3.4. Inicialización de una variable estructura

En el siguiente ejemplo se describe cómo se realiza la inicialización de variables estructura. Para ello declaramos una nueva estructura que guarda una cadena de caracteres y un número entero, a la que denominaremos *trabajadores.*

```
#include <stdio.h>
#define LONGITUD 30

void main (void)
{

        struct trabajadores
        {
                char nombre [LONGITUD];
                int num;
        };

        /*Inicialización de las estructuras tra_1 y tra_2 */
        struct trabajadores tra_1 = {“Holmes”, 001};
        struct trabajadores tra_2 = {“James Bond”, 007};

        printf (“LISTA DE AGENTES SECRETOS:\n”);
        printf (“\tNombre : %s \n”, tra_1.nombre);
        printf (“\t Número del agente %d\n”, tra_1.num);
        printf (“\tNombre : %s \n”, tra_2.nombre);
        printf (“\t Número del agente %d\n”, tra_2.num);

}
```

Al ejecutar este programa se consigue el siguiente resultado:

```
LISTA DE AGENTES SECRETOS:
Nombre : Holmes
   Número del agente 001
   Nombre : James Bond
   Número del agente 007
```

3.5. Sentencias de asignación utilizando estructuras

Hoy en día en el lenguaje C es posible realizar asignaciones entre estructuras, esto es, es posible asignar el valor de una variable estructura a otra variable estructura que contenga el mismo tipo de datos.

Para probar esta opción repetiremos el anterior ejemplo.

```
#include <stdio.h>
#define LONGITUD 30

void main (void)
{
        struct trabajadores
        {
                char nombre [LONGITUD];
                int num;
        };

        struct trabajadores tra_1 ;
        struct trabajadores tra_2 ;

        printf ("\n1.Datos del Agente. \nIntroduce el nombre: ");
        gets (tra_1. nombre);
        printf ("\nIntroduce el número: ");
        scanf ("%d", &tra_1.num);

        tra_2 = tra_1;

        printf ("LISTA DE AGENTES SECRETOS :\n");
        printf ("\Nombre : %s \n", tra_1. nombre);
        printf ("\t Número del agente %d\n", tra _1.num);
        printf ("\tNombre: %s \n", tra _2. nombre);
        printf ("\tNúmero del agente%d.\n", tra _2.num);
}
```

Al ejecutar este programa veríamos lo siguiente en pantalla:

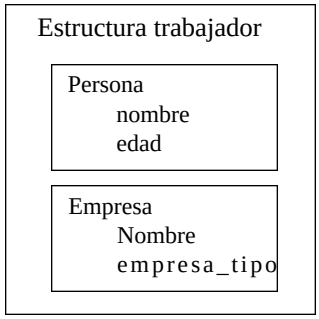
```
Datos del Agente.
Introduce el nombre: James Bond
Introduce el número: 7
LISTA DE AGENTES SECRETOS:
Nombre : James Bond
    Número del agente 7
    Nombre : James Bond
    Número del agente 7
```

Como puede verse al hacer la asignación *tra_2* = *tra_1*, la información contenida en la variable estructura *tra_1* se copia en la estructura *tra_2* y es por ello que la misma información se visualiza dos veces.

4. Estructuras anidadas

Hemos visto cómo un array puede ser uno de los componentes internos de otro array. También en las estructuras cabe la posibilidad de que uno de sus componentes sea a su vez una estructura. De esta manera se crean tipos de datos complejos y especiales. Supongamos que queremos crear una nueva estructura de nombre *trabajador* para guardar la información de un trabajador. Entre otras cosas queremos guardar información de su empresa, nombre y actividad, e información sobre la persona, nombre y edad por ejemplo. Se ve claramente que podemos crear una estructura y en su interior otras dos:

```
struct persona
{
        char nombre [LONGITUD];
        int edad;
};

struct empresa
{
        char nombre [LONGITUD];
        char empresa_tipo [LONGITUD];
};

struct trabajador
{
        struct persona currela;
        struct empresa mi_empresa;
};
```



En el siguiente ejemplo se muestra el uso de esta estructura anidada:

```
#include <stdio.h>
#define LONGITUD 30

void main (void)
{
        struct persona
        {
                char nombre [LONGITUD];
                int edad;
        };
```

```
        struct empresa
        {
                char nombre [LONGITUD];
                char empresa_tipo [LONGITUD];
        };

        struct trabajador
        {
                struct persona currela;
                struct empresa mi_empresa;
        };

        struct trabajador chico;

        printf ("\nDatos del trabajador. \nIntroduce el nombre: ");
        gets (chico.currela.nombre);
        printf ("\nEdad : ");
        scanf ("%d", &chico.currela.edad);
        fflush (stdin);

        printf ("\nIntroduce el nombre de la empresa: ");
        gets (chico.mi_empresa. nombre);
        printf ("\nActividad de la empresa: ");
        gets (chico.mi_empresa. empresa _tipo);

        printf ("INFORMACIÓN DEL TRABAJADOR:\n");
        printf ("\tNombre : %s \n", chico. currela. nombre);
        printf ("\tEdad : %d\n", chico.currela.edad);
        printf ("\tNombre de la empresa: %s \n", chico.mi_empresa. nombre);
        printf ("\Actividad de la empresa: %s.\n", chico.mi_empresa.empresa_tipo);
}
```

Al ejecutar este programa esto es lo que se vería en pantalla:

```
Datos del trabajador.
Introduce el nombre: Iñaki Goirizelaia
Edad : 25
Introduce el nombre de la empresa: Adicorp
Actividad de la empresa : Visión Artificial

INFORMACIÓN DEL TRABAJADOR:
Nombre: Iñaki Goirizelaia
   Edad : 25
   Nombre de la empresa : Adicorp
   Actividad de la empresa : Visión Artificial
```

Como puede verse hemos creado una *estructura* de nombre *chico* que contiene el *tipo de dato trabajador*. Dentro de esta *estructura* se hallan estas otras dos: *currela* (que contiene el tipo de dato persona) y *mi_empresa* (que contiene el tipo de dato empresa). Por lo tanto, para saber qué edad posee el *chico*, debemos utilizar el operador "punto" (.) dos veces, tal y como se muestra en la siguiente asignación:

printf ("\tEdad : %d\n", chico.currela.edad);

5. Las estructuras como argumentos de funciones

Hemos visto que las variables y los arrays pueden pasarse como argumentos a las funciones y todo ello sin problemas. ¿Por lo tanto puede ocurrir lo mismo con las estructuras? la respuesta, como todos esperábamos, es afirmativa. Es posible pasar estructuras a las funciones como argumentos. En el siguiente ejemplo escribiremos el programa anterior, pero ahora mediante el uso de funciones. Para ello escribiremos dos nuevas funciones. Una de ellas leerá del teclado la información sobre un nuevo trabajador y la otra visualizará en pantalla la información de un trabajador.

```
#include <stdio.h>
#define LONGITUD 30

struct trabajadores
{
        char nombre [LONGITUD];
        int num;
};

/****
la función nuevo_nombre lee la información de un nuevo trabajador y la función visualizar_datos visua -
liza la información de un trabajador.
****/
struct trabajadores nuevo_nombre(void);
void visualizar_datos (struct trabajadores currela);

void main (void)
{
        struct trabajadores tra_1 ;
        struct trabajadores tra_2 ;

        tra_1 = nuevo_nombre();
        tra_2 = nuevo_nombre ();

        printf ("LISTA DE AGENTES SECRETOS:\n");
        visualizar_datos (tra_1);
```

```
        visualizar_datos (tra_2);
}

/*
Como se puede apreciar esta función devuelve una estructura tipo trabajadores donde se almacena la infor -
mación necesaria sobre cada trabajador, utilizando para ello la sentencia return
*/

struct trabajadores nuevo_nombre(void)
{

        struct trabajadores currela;

        printf ("\nDatos del agente. \nIntroduce el nombre: ");
        gets (currela.nombre);
        printf ("Introduce el número: ");
        scanf ("%d", & currela.num);
        fflush (stdin);

        return (currela);

}

void visualizar_datos (struct trabajadores currela)
{

        printf ("\n\n Agente secreto\n");
        printf ("\tNombre : %s \n", currela.nombre);
        printf ("\t Número del agente %d\n", currela.num);
}
```

Cuando compilamos este programa el compilador puede dar un mensaje diciendo que la función *visua - lizar_datos* toma la estructura como valor y no como dirección. ¿Esto cómo se soluciona? Seguramente muchos de vosotros tenéis en mente la solución mediante la utilización de punteros. En el apartado siguiente se estudia el uso de punteros a estructuras y su utilización en el paso de estructuras como argumentos de funciones.

6. Punteros a estructuras

En el ejemplo del apartado anterior sobre agentes secretos hemos definido la siguiente estructura :

```
struct trabajadores
    {
        char nombre [LONGITUD];
        int num;
    };
```

En este apartado vamos a estudiar cómo podemos utilizar punteros a estructuras. El puntero a una estructura, siguiendo el formato que ya hemos utilizado en muchas otras ocasiones, se define de este modo:

```
struct trabajadores *tra_ptr;
```

Mediante esta declaración decimos que **tra_ptr** es una variable puntero y además es un puntero a la estructura *trabajadores*.

Por otra parte definiremos *la variable estructura agente*:

```
struct trabajadores agente;
```

A continuación se asigna la dirección de la variable estructura *agente* a la variable puntero *tra_ptr*.

```
tra_ptr = &agente;
```

De esta forma, hemos conseguido asignar al puntero *tra_ptr* la dirección de memoria donde se encuentra almacenada la variable estructura *agente*.

Ahora surge otra pregunta, utilizando el puntero *tra_ptr* ¿cómo leemos el contenido de la estructura? Para ello se utiliza el operador ->.

```
tra_ptr -> nombre
tra_ptr -> num
```

Por lo tanto, la siguiente expresión es cierta

```
tra_ptr -> num= = agente.num
```

y expresan lo mismo. Rehagamos el anterior ejemplo del apartado 5 utilizando punteros de estructuras.

```
#include <stdio.h>
#define LONGITUD 30

struct trabajadores
{
        char nombre [LONGITUD];
        int num;
};

struct trabajadores nuevo_nombre(void);
void visualizar_datos(struct trabajadores*tra_ptr);

void main (void)
{
        struct trabajadores tra_1 ;
        struct trabajadores tra _2 ;

        tra_1 = nuevo_nombre();
        tra_2 = nuevo_nombre ();

        printf ("LISTA DE AGEN TES SECRETOS:\n");
        visualizar_datos (&tra_1);
        visualizar_datos (&tra_2);
}

struct trabajadores nuevo_nombre (void)
{
        struct trabajadores currela;

        printf ("\nDatos del agente. \nIntroduce el nombre: ");
        gets (currela.nombre);
        printf ("Introduce el número: ");
        scanf ("%d", & currela.num);
        fflush (stdin);

        return (currela);
}

void visualizar_datos (struct trabajadores*tra_ptr)
{
        printf ("\n\n Agente secreto\n");
        printf ("\tNombre : %s \n", tra_ptr -> nombre);
        printf ("\tNúmero del agente %d\n", tra_ptr -> num);
}
```

7. Arrays de estructuras

Supongamos que queremos crear una estructura de datos compleja para guardar una larga lista de agentes secretos. Hasta el momento hemos examinado cómo se guarda la información de cada agente mediante una estructura. Pero ¿cómo se guarda la información de todos los agentes?

Podemos crear un array cuyos componentes sean estructuras. Observemos cómo se puede llevar a cabo:

Declaración de la estructura:

```
struct trabajadores
{
        char nombre [LONGITUD];
        int num;
};
```

Y el array lo declaramos de la siguiente manera:

```
struct trabajadores agentes_secretos [LONGITUD] ;
```

Mediante esta declaración decimos que *la variable agentes_secretos* es un array y que en cada elemento del array se almacenan estructuras del *tipo de dato trabajadores*.

En el siguiente ejemplo y utilizando un array, se crea una lista de 30 agentes secretos. El progrma presenta un menú que da opción a elegir entre introducir un nuevo agente, ver la lista completa o salir del programa.

```
#include <stdio.h>
#include <conio.h>
#define LONGITUD 30
#define VERDAD 1
#define FALSO 0

void nombre_nuevo (struct trabajadores agentes_secretos [], int *cantidad);
void visualizar_datos (struct trabajadores agentes_secretos[] , int * cantidad);

struct trabajadores
{
      char nombre [LONGITUD];
```

```
        int num;
};

void main (void)
{
        struct trabajadores agentes_secretos [LONGITUD] ;
        int signo = VERDAD;
        int car;
        int cantidad = 0;

        clrscr () ;



        while (signo == VERDAD)
        {
                clrscr ();
                printf (" \n Pulsa S para introducir un nuevo agente");
                printf (" \n Pulsa Z para conseguir la lista de todos los agentes ");
                printf (" \n Pulsa X para salir del programa ");
                printf (" \n \n de momento tenemos %d agentes.", cantidad);
                printf (" \n \n \n introduce la opción que quieras: ");

                car= getche ();

                switch (car)
                {
                        case 'S' :
                                nombre_nuevo (agentes_secretos, &cantidad);
                                break;
                        case 'Z' :
                                visualizar_datos (agentes_secretos, &cantidad);
                                break;
                        case 'X' :
                                signo= FALSO;
                                break;
                        default :
                                puts(" \n Introduce la opción adecuada, por favor:");
                }
        }
}



void nombre_nuevo (struct trabajadores agentes_ocultos [], int *cuantos)
```

```
{
        int donde = *cuantos;

        if ((*cuantos) < LONGITUD)
        {
                printf ("\nDatos del agente. \nIntroduce el nombre: ");
                gets (agentes_ocultos [donde].nombre);
                printf ("Introduce el número : ");
                scanf ("%d", & agentes_ocultos [donde].num);
                fflush (stdin);
                *cuantos = *cuantos + 1;
        }
        else
        {
                printf ("\No tengo más espacio");
        }
}

void visualizar_datos (struct trabajadores agentes_ocultos [], int *cuanto)
{
        int cont;

        if (*cuanto > 0)
        {
                for (cont = 0; cont < *cuanto; cont ++)
                {
                        printf ("\n\n%d.Datos del agente secreto\n", cont + 1);
                        printf ("Nombre : %s", agentes_secretos [cont]. nombre);
                        printf ("\nNúmero : %d", agentes_secretos [cont]. num);
                }
        }
        else
        {
                printf ("\n\nTodavía no hay agentes ocultos en la lista.");
        }

        printf ("\n\n\nPulsa cualquier tecla para continuar.");
        getch ();
}
```

8. Bibliografía

AUTONAKOS, K.C., MANSFIELD, K.C. *Programación Estructurada en C.* Ed. Prentice Hall

KERNIGHAN, BW., RITCHIE D.M. *The C Programming Language.* Ed. Prentice Hall.

PRESS, W.H., TEKOLSKY, S.A., VETTERLING, W.T., FLANNERY, B.P. *Numerical Recipes in C.* Ed. Cambridge

QUERO, E., LÓPEZ HERRANZ, J. *Programación en Lenguajes Estructurados.* Ed. Paraninfo.

WAITE, M., PRATA, S., MARTIN, D. *Programación en C. Introducción y Conceptos Avanzados.* Ed. Anaya Multimedia.

9. Ejercicios propuestos

Se desea realizar un programa que controle el proceso de alquiler de un stand de una feria de muestras. Para ello se guarda en un array unidimensional la siguiente información sobre d stands alquilados por las distintas empresas:

- Nombre de la empresa
- Número de pabellón
- Duración del alquiler
- Metros cuadrados del stand
- Precio total del alquiler

La información de este array se guarda utilizando los nombres de las empresas en orden alfabético.

Utilizaremos otro array unidimensional para guardar la siguiente información sobre los 5 pabellones de la Feria:

- Tarifa (Ptas / m2 día)
- Metros cuadrados alquilados en cada pabellón

Al principio el programa preguntará por la tarifa de cada pabellón. A continuación seguirá ejecutándose bajo el control del siguiente menú:

1. Añadir un stand
2. Quitar un stand
3. Cambiar la tarifa de un pabellón
4. Visualizar la información de los stands y los pabellones
5. Salir del programa

1. Opción

Pedirá información del nuevo stand (NOMBRE de la empresa, NÚMERO del pabellón, DÍAS, METROS). Partiendo de estos datos el programa calculará y visualizará el precio total del alquiler, luego lo guardará en el array que guarda los stands.

2. Opción

Pedirá el nombre de la empresa y borrará su stand. En ese momento se deberán adecuar los metros cuadrados alquilados de dicho pabellón.

3. Opción

Pedirá el número del pabellón y la nueva tarifa. Luego esa tarifa se actualizará junto a los alquileres de todos los stand que se encuentren en ese pabellón y que ven modificado su precio.

4. Opción

Visualizará la información de todos los stand de la feria de muestras y de los 5 pabellones.

5. Opción

Finalizará la ejecución del programa.

Nota

El array donde guardamos los stands también puede estar vacío. En ese caso si se opta por la 2ª o la 4ª opción el programa nos pondrá en conocimiento de ello.