Fundamentos de Programación – Programación Estructurada en C:
9.- ESTRUCTURAS

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

9.- ESTRUCTURAS

9.- ESTRUCTURAS

Una **estructura** es una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificativo.

Un array también agrupa un conjunto de datos bajo el mismo identificativo, pero en el caso de los arrays, todos los datos son del mismo tipo.

9.1.- Definición y declaración de estructuras

Antes de poder usar variables estructura, hay que **definir** la estructura **y después**, como con cualquier tipo de dato, **declarar** las variables.

La **definición** de una estructura se hace de la siguiente forma:

```
struct <nombre_estructura>
{
    <declaración campo 1>
    ...
    <declaración campo n>
};
```

La estructura estará constituida por una serie de elementos, a los que se llama **campos**. Cada uno de los campos de una estructura, es una variable de cualquier tipo (tipos básicos, arrays u otros tipos de estructuras), su inclusión en la estructura se hace mediante su declaración.

Ejemplo:

```
struct alumno
{
    char nombre[30];
    char direccion[20];
    int n_matricula;
    float notas[10];
};
```

Una vez hecha la definición de la estructura, dicha estructura se puede usar como si fuera un nuevo tipo de dato. Al definir una estructura no se está haciendo ninguna reserva de memoria. Sólo se indica cómo es la estructura, para luego poder usarla como nuevo tipo de dato y declarar variables de ese tipo.

Así, la **declaración** de una variable estructura se hace como la declaración de una variable de cualquier otro tipo de dato:

```
struct <nombre_estructura> <nombre_variable_estructura>;
```

Ejemplo:

```
struct alumno alumno1;
```

Es necesario que antes de declarar variables de un tipo de estructura, ese tipo de estructura esté definida.

La declaración de una variable estructura, al igual que la declaración de cualquier variable de cualquier tipo, implica una reserva de memoria. En el caso de una estructura, la cantidad de memoria reservada es la necesaria para albergar todos los campos de la misma. Todos los bytes de memoria que se reservan, están seguidos. Por ejemplo:

```
int i;

/*
Situación en memoria (P. e., se reserva a partir de la dirección 1000):
```

Variable	Dirección	Memoria	Tamaño
i	1000 ... 1003	-	4 bytes

```
*/

struct alumno alumno1;

/*
Situación en memoria (P. e., se reserva a partir de la dirección 2000):
```

Variable		Dirección	Memoria	Tamaño
alumno1	nombre	2000 ... 2029	-	30 bytes
	direccion	2030 ... 2049	-	20 bytes
	n_matricula	2050 ... 2053	-	4 bytes
	notas	2054 ... 2093	-	40 bytes

```
*/
```

Normalmente, se suelen poner las definiciones de las estructuras a usar en un programa, entre los *#define* y las declaraciones de los prototipos de las funciones; así pueden empezar a usarse desde las mismas declaraciones de prototipos de las funciones.

Se pueden declarar también arrays de estructuras.

Ejemplo:

```
struct alumno clase[100];
```

9.2.- Acceso a los campos estructura

Para acceder a los campos de una variable estructura, se usa el operador **punto** (.). El acceso a cada campo se hace mediante en nombre de la variable estructura seguido de un punto y el nombre del campo al que se accede.
Ejemplo:

```
struct alumno
{
    char nombre[30];
    char direccion[20];
    int n_matricula;
    float notas[10];
};
...

struct alumno alumno1;

alumno1.n_matricula=10451;
```

El acceso a los campos de una estructura es necesario, por ejemplo, para dar valores a los campos de una variable estructura en concreto. Cuando se declara la variable estructura, simplemente se hace la reserva de memoria necesaria para albergarla. Para que los campos tengan los valores que han de tener, habrá que acceder a ellos y asignárselos. Para las asignaciones, hay que tener en cuenta que los arrays son tipos de datos especiales. No se puede asignar valores a un array en completo, hay que hacerlo de elemento a elemento; y así hay que hacerlo también cuando un array es campo de una estructura. Es decir:

Si tenemos la siguiente definición:

```
struct alumno
{
    char nombre[30];
    char direccion[20];
    int n_matricula;
    float notas[10];
};
...

struct alumno alumno1;
```

```
alumno1.nombre="Aitor"; //esto no es correcto
alumno1.direccion="San Mamés"; //esto no es correcto
alumno.n_matricula=10451; //esto sí es correcto
alumno1.notas={5,6,7,8,9,10,5,6,7,8};//esto no es correcto
```

Las asignaciones de valores para los campos de tipo array no son correctas, porque tampoco lo son para variables arrays. Esas asignaciones directas de valores sólo se pueden hacer en las declaraciones. En el caso de estructuras, se podrían usar esas asignaciones en las declaraciones de los campos de las definiciones de las estructuras, pero habrá que tener en cuenta, que todas las variables estructura declaradas, tendrán automáticamente los valores indicados en las declaraciones de los campos.
Ejemplo:

```
struct alumno
{
    char nombre[30]="Aitor";
    char direccion[20]="San Mamés";
    int n_matricula=10451;
    float notas[10];
};
...

struct alumno alumno1,alumno2;
```

Para acceder a los campos arrays, habrá que hacerlo elemento a elemento. El caso de las cadenas de caracteres es especial, ya que se pueden usar las funciones de la librería *string*.

Ejemplo:

```
struct alumno
{
    char nombre[30];
    char direccion[20];
    int n_matricula;
    float notas[10];
};
...

struct alumno alumno1,alumno2;
int i;

printf("\nIntroduzca el nombre del alumno:");
gets(alumno1.nombre);

printf("\nIntroduzca la matrícula: ");
scanf("%d",&alumno1.n_matricula);

for(i=0;i<10;i++)
{
    printf("\nIntroduzca la nota %d",i+1);
    scanf("%f",&alumno1.notas[i]);
}

alumno2.nombre=alumno1.nombre; //ERROR! Los arrays sólo se pueden copiar elemento a elemento
strcpy(alumno2.nombre,alumno1.nombre); //SÍ, pq son dos strings

alumno2.n_matricula=alumno1.n_matricula; //SÍ, pq se copian dos variables float

alumno2.notas=alumno1.notas; //ERROR! Los arrays sólo se pueden copiar elemento a elemento
for(i=0;i<10;i++) //SÍ
{
    alumno2.notas[i]=alumno1.notas[i];
} //No hay otra forma, no se puede usar strcpy, pq los arrays notas no son strings
```

Si lo que se quiere es copiar los valores de **todos** los campos de una estructura en otra, no es necesario hacerlo campo a campo. Para el ejemplo anterior:

```
alumno2=alumno1;
```

sería correcto, y después de esa sentencia, todos los campos de alumno2 tendrían los mismos valores que los de alumno1. Cuando no se quiere copiar el valor de todos los campos, habrá que acceder a cada uno de los campos que se quiera copiar y hacerlo de la forma apropiada.

Para el caso de arrays de estructuras, se tiene que trabajar con cada elemento del array, y se mantiene todo lo dicho hasta ahora:

```
clase[6].nombre

clase[3]=alumno1;
```

9.3.- Estructuras anidadas

Los campos de las estructuras pueden ser de cualquier tipo de dato, incluso, un tipo de estructura. Así, se tienen estructuras anidadas, y distintos niveles de anidamiento.

Ejemplo:

```
struct persona
{
    char nombre[30];
    char direccion[20];
};

struct alumno
{
    struct persona individuo;
    int n_matricula;
    float notas[10];
};

int main()
{
    struct alumno alumno1;

    printf("\nIntroduzca el nombre del alumno:");
    gets(alumno1.individuo.nombre);

    printf("\nIntroduzca la matrícula: ");
    scanf("%d",&alumno1.n_matricula);

    for(i=0;i<10;i++)
    {
        printf("\nIntroduzca la nota %d",i+1);
        scanf("%f",&alumno1.notas[i]);
    }

    return 0;
}
```

Tal como se ve en el ejemplo anterior, para acceder a un campo que pertenece a una estructura anidada, hay que usar el operador punto tantas veces como sea necesario para ir avanzando en el anidamiento de las estructuras.

9.4.- Punteros a estructuras

Tal como declaramos punteros a cualquier tipo de dato, también se pueden declarar punteros a estructuras. Son necesarios, por ejemplo, a la hora de **pasar estructuras por referencia a funciones**. Tal como pasaba con otros

tipos de datos, el pasar por referencia implica que lo que realmente se pasa a la función es la dirección de la variable, y en la función esa dirección se recoge en un puntero.

Así, cuando tenemos una variable estructura, para acceder a un campo concreto, según lo aprendido, lo haríamos de la siguiente forma:

<variable estructura>.<campo>

Si en vez de tener una variable estructura, lo que tenemos es un **puntero a estructura**, como con cualquier otra variable, se cumplirá que:

***<puntero>**	⇔	<variable>
***<puntero a estructura>**	⇔	<variable estructura>
(*<puntero a estructura>**).**<campo>	⇔	<variable estructura>.<campo>

Para el acceso a los campos en el caso de punteros a estructura, se puede usar el operador flecha (->).

(*<puntero a estructura>**).**<campo> ⇔ <puntero a estructura>**->**<campo>

Es decir, que en vez de tener que usar el '*' y el '.', se puede usar sólo el operador flecha, para acceder al campo de una estructura a través del puntero que la apunta.

Ejemplo:

```
struct alumno
{
    char nombre[30];
    char direccion[20];
    int n_matricula;
    float notas[10];
};

void rellenar(struct alumno *);

int main()
{
    struct alumno alumno1;

    rellenar(&alumno1);
    ...

    return 0;
}

void rellenar(struct alumno *alum)
{
    printf("\nIntroduzca el nombre del alumno:");
    gets(alum->nombre);

    printf("\nIntroduzca la matrícula: ");
    scanf("%d",&(alum->n_matricula));

    for(i=0;i<10;i++)
    {
        printf("\nIntroduzca la nota %d",i+1);
        scanf("%f",&(alum->notas[i]));
    }
}
```