ESTRUCTURAS EN LENGUAJE C

Las estructuras ya estaban presentes en C. Hay quien las ve como una clase, pero sin métodos (solo almacena datos).

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 vienen en nuestro auxilio 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.

Vamos a crear una declaracion 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.

Una vez definida la estructura, podemos usarla declarando 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.

Arrays de estructuras

Supongamos ahora que queremos guardar la informacion de varios amigos. Con una variable de estructura solo podemos guardar los datos de uno. Necesitamos declarar arrays de estructuras:

struct estructura\_amigo amigo[10];

Ahora necesitamos saber como 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*.

Inicializar una estructura

Primero se define la estructura y luego al declarar una variable como estructura le damos el valor inicial que queramos. Ejemplo:

struct estructura\_amigo amigo = {  
 "Juanjo",  
 "Lopez",  
 "592-0483",  
 30  
 };

Por supuesto hemos de meter en cada campo el tipo de datos correcto.

Punteros a estructuras

Primero hay que definir la estructura igual que antes, pero al declarar la variable de tipo estructura debemos ponerle el operador '\*' para indicarle que es un puntero.

Es importante recordar que un puntero no debe apuntar a un lugar cualquiera, debemos darle una direccion valida donde apuntar. No podemos por ejemplo crear un puntero a estructura y meter los datos directamente mediante ese puntero, no sabemos donde apunta el puntero y los datos se almacenarían en un lugar cualquiera.

Y para comprender como funcionan nada mejor que un ejemplo. Este programa utiliza un puntero para acceder a la informacion 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",  
 "Lopez",  
 "592-0483",  
 30  
 };  
struct estructura\_amigo \*p\_amigo;  
int main()  
 {  
 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 );  
 }

*p\_amigo* es un puntero a la estructura *estructura\_amigo*. Dado que es un puntero tenemos que indicarle donde debe apuntar, en este caso vamos a hacer que apunte a la variable *amigo*:

p\_amigo = &amigo;

El operador & que significa 'dame la direccion donde esta almacenado...'.

Para 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 '->'.

Punteros a arrays de estructuras

Por supuesto también podemos usar punteros con arrays de estructuras.

Paso de estructuras a funciones

Las estructuras se pueden pasar directamente a una funcion igual que hacíamos con las variables. Ejemplo:

int suma( struct estructura\_amigo arg\_amigo ) {

return arg\_amigo.edad+20;

}

Inicialización de estructuras

^

De un modo parecido al que se inicializan los arrays, se pueden inicializar estructuras, tan sólo hay que tener cuidado con las estructuras anidadas. Por ejemplo:

struct A {

int i;

int j;

int k;

};

struct B {

int x;

struct C {

char c;

char d;

} y;

int z;

};

A ejemploA = {10, 20, 30};

B ejemploB = {10, {'a', 'b'}, 20};

Cada nueva estructura anidada deberá inicializarse usando la pareja correspondiente de llaves "{}", tantas veces como sea necesario.

Asignación de estructuras

^

La asignación de estructuras está permitida, pero sólo entre objetos del mismo tipo de estructura, (salvo que se usen constructores), y funciona como la intuición nos dice que debe hacerlo.

Veamos un ejemplo:

struct Punto {

int x, y;

Punto() {x = 0; y = 0;}

} Punto1, Punto2;

int main() {

Punto1.x = 10;

Punto1.y = 12;

Punto2 = Punto1;

}

La línea Punto2 = Punto1; equivale a Punto2.x = Punto1.x; Punto2.y = Punto1.y;.

Quizás te hayas quedado intrigado por el comentario anterior, que adelantaba que se pueden asignar estructuras diferentes, siempre que se usen los constructores adecuados.

Esto, en realidad, se puede extender a cualquier tipo, no sólo a estructuras. Por ejemplo, definiendo el constructor adecuado, podemos asignar un entero a una estructura. Veamos cómo hacer esto.

Hasta ahora, los constructores que hemos visto no usaban argumentos, pero eso no significa que no puedan tenerlos.

Crearemos como ejemplo, una estructura para manejar números complejos. Un número complejo está compuesto por dos valores reales, el primero contiene lo que se llama la parte real y el segundo la parte imaginaria.

struct complejo {

double real;

double imaginario;

};

6

Esta estructura es suficiente para muchas de las cosas que podemos hacer con números imaginarios, pero aprovechando que podemos crear funciones, podemos añadir algunas que hagan de una forma más directa cosas que de otro modo requieren añadir código externo.

Por ahora nos limitaremos a añadir unos cuantos constructores. El primero es el más lógico: un constructor por defecto:

struct complejo {

complejo() { real=0; imaginario = 0; }

double real;

double imaginario;

};

Este construtor se usará, por ejemplo, si declaramos un array:

complejo array[10];

El constructor por defecto será llamado para cada elemento del array, aunque no aparezca tal llamada en ningún punto del programa.

Otro constructor nos puede servir para asignar un valor a partir de dos números:

struct complejo {

complejo() { real=0; imaginario = 0; }

complejo(double r, double i) { real=r; imaginario = i; }

double real;

double imaginario;

};

Mediante este constructor podemos asignar valores inciales en la declaración:

complejo c1(10.23, 213.22);

Los números reales se consideran un subconjunto de los imaginarios, en los que la parte imaginaria vale cero. Esto nos permite crear otro constructor que sólo admita un valor real:

struct complejo {

complejo() { real=0; imaginario = 0; }

complejo(double r, double i) { real=r; imaginario = i; }

complejo(double r) { real=r; imaginario = 0; }

double real;

double imaginario;

};

Este constructor nos permite, como en el caso anterior, inicializar un valor de un complejo en la declaración, pero también nos permite asignar un valor double a un complejo, y por el sistema de promoción automático, también podemos asignar valores enteros o en coma flotante:

complejo c1(19.232);

complejo c2 = 1299.212;

int x = 10;

complejo c3 = x;

Este tipo de constructores se comportan como conversores de tipo, nada nos impide crear constructores con cualquier tipo de parámetro, y tales constructores se podrán usar para convertir cualquier tipo al de nuestra estructura.

Arrays de estructuras

La combinación de las estructuras con los arrays proporciona una potente herramienta para el almacenamiento y manipulación de datos.

Ejemplo:

struct Persona {

char Nombre[65];

char Direccion[65];

int AnyoNacimiento;

} Plantilla[200];

Vemos en este ejemplo lo fácil que podemos declarar el array Plantilla que contiene los datos relativos a doscientas personas.