Anexo II: Lenguaje C

1. Compilación de programas en C.

Para compilar en *lucano* utilizaremos la siguiente instrucción:

lucano% gcc programa.c [−o ejecutable] [−lm] [−g]

- *gcc* nombre del compilador de C.
- *programa.c* es el código fuente, un fichero texto ASCII.
- *−o ejecutable* es el nombre que le queremos dar al programa ejecutable; de no incluirse esta opción el nombre que toma la salida binaria por defecto es *a.out*.
- *−lm* incluye la librería matemática, para programas que usen la raíz, potencia, etc.
- *−g* posibilita la depuración con el programa *ddd*.

Existen otros compiladores, como por ejemplo el *cc*, sin bien lo normal será que compilemos con el *gcc*.

Es posible que sea necesario especificar que estamos en el directorio actual, para poder ejecutar el programa, ya que por razones de seguridad en algunas máquinas los administradores eliminan del *`path'* de los usuarios el directorio actual. De modo que para ejecutar un programa que se encuentre en el directorio actual del usuario es posible tener que escribir:

lucano% ./programa

2. Estructura general de un programa en C.

```
#include <stdio.h>

main ( )

{

int a, b;

int c;

a = 5;

b = 3;

c = a + b;

printf (%d + %d = %d, a, b, c);

}
```

Función Principal
Declaración de objetos
Asignación de valores
Sentencias (Cuerpo del programa)

En C, todas las sentencias van seguidas de *`;'* (excepto las funciones). Además, todos los *objetos* han de ser declarados previamente, ya sean variables, funciones, constantes, etc.

La instrucción *printf*, imprime una cadena formateada. La cual va entrecomillada y en la que se pueden escribir tanto letras y números como formatos para los datos de una variable que se quiere mostrar, como secuencias de escape. Los formatos para los datos de las variables se abordarán en el apartado *1.4. Tipos de datos*, mientras que las secuencias de escape se revisarán en el apartado *1.5 Secuencias de escape*.

3. Identificadores

Los nombres que pueden recibir los objetos son cadenas alfanuméricas que no comiencen por un número, si bien, dependiendo del compilador, solo diferenciarán unos objetos de otros los primeros *`m'* caracteres de la cadena. Así podemos encontrar compiladores que usan desde 8 hasta 32 caracteres en los nombres de objetos.

Tampoco es posible usar caracteres especiales (`,,/,\,*, *etc.*) para dar nombre a un objeto; así como tampoco pueden usarse los nombres reservados de C: tanto funciones como tipos de datos, etc. Y, al igual que las máquinas Unix, el lenguaje C diferencia entre mayúsculas y minúsculas, por lo que para él no será lo mismo el objeto *`aux'*, que el objeto *`aUx'*

4. Tipos de Datos.

La siguiente tabla muestra un resumen de los tipos de dato disponibles en C y el rango de números que abarca cada uno de ellos.

Tipo de Dato	**Significado**	**Rango**
int	Entero	−32.768 a 32.767
unsigned int	entero sin signo	0 a 65.535
short int	entero corto	−32.768 a 32.767
unsigned short int	entero corto sin sgino	0 a 65.535
long int	entero largo	−2.147.483.648 a 2.147.483.647
unsigned long int	entero largo sin signo	0 a 4.294.967.295
float	Real	3,4E +/− 38 (7 dígitos)
double	real de doble precisión	1,7E +/− 308 (15 dígitos)
long double	real de doble precisión largo	1,7E +/− 4932 (15 dígitos)
char	Carácter	−128 a 127

Los **enteros** suelen tener un tamaño de 2 bytes, es decir, 16 bits lo que posibilita poder almacenar 216 números o lo que es lo mismo un rango que va desde 0 a 65535. Si bien los *enteros largos* suelen ser el doble de grandes que los enteros, así si para una máquina el entero es de 2 bytes, el entero largo será de 4, lo que posibilitará un rango de 0 a 4.294.967.295. Es posible, dependiendo de la máquina que el entero tenga un tamaño de 4 bytes, en cuyo caso el entero largo tendrá un tamaño de 8 bytes.

La modificación de cualquiera de estos por la palabra *unsigned* hace que los rango sean estrictamente positivos; en su ausencia, estos rangos están particionados en mitad de positivos y mitad de negativos.

Los **reales** son tipos de datos que sirven para representar números decimales y en coma flotante. En C los tenemos de simple y doble precisión, que únicamente varían en la cantidad de decimales que pueden almacenar.

Por su parte el tipo **carácter** es un tipo de dato muy especial en C. Sirve para almacenar un único carácter, si bien C lo almacena en memoria como su correspondiente ASCII, de modo que dependiendo del formato de salida que le demos al dato así podremos obtener una letra u un número. (A65, B66, C67, $36, a97, b98, f102, etc.)

Cabe señalar que en C no existe un tipo de dato **cadena** específico. C almacena las cadenas como vectores de caracteres y su tratamiento requiere de funciones específicas que nos permitan asignar una cadena a un variable en tiempo de compilación.

A la hora de formatear la salida de una variable para su correspondiente visionado por el terminal se utilizan los especificadores de formato de tipos, que pueden usarse tanto con la instrucción *printf* como la intrucción *scanf*.

Cáracter	**Argumento**	**Salida resultante**
d	entero	Entero con signo en base decimal.
i	entero	Entero con signo en base decimal.
o	entero	Entero sin signo en case octal.
u	entero	Entero sin signo en base decimal.
x	entero	Entero sin signo en base hexadecimal usando letras minúsculas.
X	entero	Entero sin signo en base hexadecimal usando letras mayúsculas.
f	real	Número real con signo.
e	real	Número real con signo usando notación e.
E	real	Número real con signo usando notación E.
g	real	Número real con signo en formato e ó f, pero de tamaño corto
G	real	Número real con signo en formato E ó f, pero de tamaño corto
c	carácter	Un carácter individual.
S	cadena de caracteres	Imprime cadena de caracteres.
%	ninguno	Imprime el símbolo %

05. Secuencias de escape:

Las secuencias de escape son códigos utilizados a la hora de formatear una cedana de texto. Se utilizan para añadir lineas en blanco, para desplazar la posición de puntero en la pantalla, para hacer que `pite' la máquina, etc. Las más comunes son las siguientes:

Cáracter	**Argumento**
\n	nueva línea
\t	tabulador (aproximadamente 5 caracteres)
\b	retroceso (un solo carácter)
\r	retorno de carro (hasta comienzo de línea)
\f	alimentacion de impresión
\\	*back slash* coloca una barra de este tipo `\'
\'	coloca un signo de comilla
\ddd	secuencias númericas, combinaciones en binario
\007	hace pitar sonar un pitido

6. Operadores:

En C existen una gran varieadad de operadores que permiten realizar desde una simple asignación hasta un conjunto considerable de operaciones aritmético − lógicas.

6.1Operadores aritméticos:

+	suma
−	resta
*	producto
/	división entera (si int /int o división real si int/float,doublefloat double o si float,double/intfloat, double)
%	módulo (resto de una división entera). Puede ser int%int o char %int.

Por ejemplo:

Float a

a=5/2 da un int, porque la división es entera, lo asignes a lo que lo asignes.

a=−a

6.2.Operadores lógicos:

<	menor que
>	mayor que
>=	mayor o igual que
<=	menor o igual que
==	equivalencia o comparación
!=	diferente
&&	y
||	o
!	no

6.3.Operadores de incremento y decremento.

++	incremento
−−	decremento
a++	es lo mismo que a +1
a−−	es lo mismo que a −1

b=++2 No es lo mismo: a=a+1 (1º se incrementa)

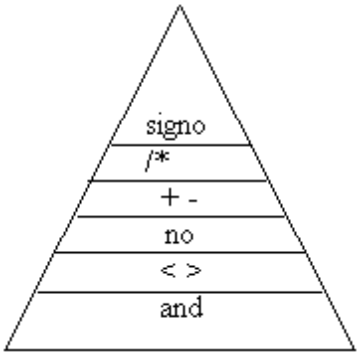
b=a++ b=a (2º se asigna)

b=a (1º se asigna)

a=a+1(2º se incrementa.

Aunque a++ y ++a son iguales.

6.4. Jerarquía de operaciones.



6.5.Operadores de asignación compuesta:

+=	Estos operadores funcionan de la siguiente forma: suma +=i es igual que poner suma=suma+i E igual para todas las demás operaciones.
−=	
*=	
/=	
%=	

7.Jerarquía de datos.

char < int < long < float < double

int/float (float)

int/float float.

Esta jerarquía, se pude alterar usando un:

casting : altera la jerarquía, para que nos devuelva un tipo de dato determinado.

int a;

float x;

a=2

x=3.15

a=x; asignamos un float a un entero. Dependiendo del margen de error, dará un error y otro; no dará error si damos un cast a un entero (int),

a=(int)x; (que x, un float, se reduzca hasta el tamaño de un entero, truncando, y luego asigne)

x= a/(int)x; divide un entero entre otro entero (un float truncado que se reduce hasa un entero), que dará un entero, en este caso y con los valores anteriores de a y x, dará 0.

x= (float)(a/(int)x); daría 0 (aunque lo pongamos como un float)

x=(float)a/(int)x; float/int = float(0.)

Hay conversines que no se pueden hacer,

a=2;

x=1003+20;

Si hacemos a=x; o da un error, o al ejecutar el programa no podrá representarlo, y si él hace un cast por su cuenta, no resultará lo que deseábamos.

Se puede promocionar de menor a mayor, pero no de mayor a menor.

8.Estructuras de control.

8.1.Esquemas condicionales:

Una condición, es cualquier sentenica, que devuelva V si se verifica, o F en caso contrario (con>,<,<=,>=,...). Por ejemplo:

if (condición)

{

;

;

}

else

{

;

;

}

En C no hay tipos lógicos. Así que lo que devuelve es:

0 si es falso

1 si es verdadero

char calificacion;

```
if (puntuación >=5)
calificación = `A';
else
calificación = `S';
printf (calificación %c, calificación);
```

O también,

```
if
{
calificación = `A';
printf (calisificación);
}
else
{
calificación = `N';
printf f (calificación);
}
printf(%c, calificación);
```

O incluso:

```
if (puntuación = =5)
calificación = `A';
else if (puntuación = = 6)
calificación = `B';
else if (puntuació = = 7)
calificación ='N';
else if (puntuación = = 8)
...
```

Los corchetes, sólo son necesarios si hay más de una sentencia.

```
if (puntuación >5)
calificación='A';
else
calificación ='N';
printf (calificación =%c=,calificación)
if (puntuación ==5)
calificación ='A';
else if (puntuación =='6')
calificación ='B'
else if (puntuación ==7)
calificación='N';
else if ...
```

8.2.Esquemas iterativos.

Mientras (condición) hacer{sentencias}:

```
while (condición)
{
...;
}
```

Hacer {sentencias}mientras(condición):

```
do
{
...;
}
while (condición)
```

Similar a Para:

```
for (inicializa contador ; condición ; revaluar contador)
{
...;
}
```

El siguiente ejemplo imprimiría 9 asteriscos (para i de 1 a 9 inc 1):

```
for (i=1 ; i<10;i++)
printf (*);
```

Otro ejemplo:

```
for (i=1, i=3, k=5 ; i<10 ; i++, k−−)
{
suma + = i
print (%d,i);
}
```

En un bucle for, puede desaparecer cualquier parte,

```
for (; ;)
a = a + 1
```

(que repite infinitamente a = a + 1)

Y si escribimos:

```
for (; ;);
a = a + 1
```

repetirá la línea `for (; ;);' y no pasará de aquí

Dentro de los esquemas iterativos, en C, utilizamos dos palabras clave:

- **break**, que se utiliza para salir del bucle y ejecutar la siguiente acción del bucle
- **continue**, que vuelve al principio del bucle, a evaluar de nuvo la condición

No es recomendable abusar de break y continue, aunque a veces facilite el trabajo.

Si en `for ' tenemos un índice, `i=1 ', no se deberá modificar éste en el cuerpo del for, sólo en la cabecera.

```
switch (expresión entera)

{

case constante 1:

sentencia 1.1;

sentencia 1.2;

case constante 2:

sentencia 2.1;

sentencia 2.2;

default:

;

;
```

Pero en C, se se cumple constante 1, se harán las sentencias 1.1 y 1.2, pero también la 2.1, 2.2, Si se cumple constante 2, se harán las sentencias 2.12, 2.2, y todas las restantes. Si no se cumple nada (sólo el default), se harán las del default. Para soluciónar so se coloca un break,

```
main ( )

int i=2

switch

{

case 1:

printf (opción 1);

break; (pasa a lo siguiente al switch, y no al case 1)

case 2:

prinf (opción 2);

break;

default:

printf (opción general);
```

9.Funciones:

tipo nombre_función (parámetros formales) Estos parámetros, puede o no tenerlos.

{

declaraciones;

cuerpo;

return valor; Devuelve un valor y pasa a la función (lo que devuelve tiene que ser del mismo tipo que la función)

}

main es similar; si no le indicamos el tipo (int, char) considera por defecto un entero.

return variable devuelve una variable

Los **parámetros**, por ahora, **son de entrada**. La única salida es la que devuelve la función (veremos más al llegar a los punteros).

Para llamar una función, *nombre_función (parámetros actuales) [estos últimos son*

```
#include <stdio.h>

main ( )

{

int a, b, c;

c = suma (a,b);

printf (a + b = % d, c);

}

suma (int x, int y)

{

int k;

k = x + y

return k

}
```

Las ***funciones void***, no devuelven nada.

Otro ejemplo sería,

```
main ()
{
printf (Estoy en main \n);
alaska (); llama a la función alaska, se ejecuta y al terminar vuelve a la función main
printf Estoy en main \n);
bermudas ();
printf (Estoy en main \n);
siberia ();
printf (Estoy en main \n);
}
alaska ()
{
printf (Estoy en alaska);
}
bermudas ()
{
printf (Estoy en bermudas);;
}
siberia ()
{
printf (Estoy en siberia);
}
```

stdio.h no es un librería, lo que contiene son declaraciones

Otro ejemplo sería,

```
# include <stdio.h>
main ()
```

```
{

int a=5;

int b;

b = cuadrado (a);

printf (El cuadrado de %d es %d \n, a, b);

}

cuadrado (int x) [x recibe el valor de a] [la función es de tipo entero al no decir nada]

{

int c;

c = x*x;

return c;

}
```

Así, al final b es igual a 25, es decir, el cuadrado de a.

Si una función devuelve un double, todo lo relacionado con esa variable debe ser también double. Por defecto C guarda espacio para asignaciones de tipo entero en memoria. Habría que indicarle desde el rpincipio que se trata de un double, reservando espacio para tal. (Colocando la función con el double antes del main).

Podemos declarar las funciones globalmente, o internas a main (locales). Al declararlas globales, sirven para todo el programa, y dicha declaración se haría fuera del main. Las funciones locales se declararían dentro del main, y sólo se podrían llamar desde éste.

Igual pasaría con las variables (variable externa), sin embargo nunca se deben utilizar declaraciones de variables externas, pues podemos modificar el valor de la variable en toda la función, produciéndose así efectos colaterales.

10. Vectores y Matrices Numéricas

Un vector y una matriz (y en general, un *array* de **n** dimensiones) son entidades que utilizan múltiples elementos a los que se accede por medio de un índice, estos deben tener unas dimensiones específicas que deben ser especificadas en su iniciación para poder trabajar con ellos apropiadamente.

Para iniciar un *array* tenemos que poner la siguiente sentencia en la sección de declaraciones de la función:

```
tipo nombre [dimensión1] [dimensión2] [dimensión3] ... ;
```

Descripción:

tipo − Es el tipo de datos que queremos que reserve el *array*. Estos que podemos poner son: **char** (hablamos de una cadena de caracteres), **int**, **double**, **float** ...

nombre − Nombre que queramos darle a la matriz.

[dimensión1] [dimensión2] [dimensión3] ... − Ponemos dentro de cada corchete el número de elementos que se van a almacenar en esa dimensión. Podemos además, crear un *array* con las dimensiones que queramos, es decir, si ponemos una dimensión (poniendo solo un número entre corchetes) declaramos lo que llamaremos un vector, con dos dimensiones hablamos de una matriz y con **n** dimensiones nos referimos a un *array* **n** dimensional.

Existen por tanto muchos tipos de iniciaciones con *arrays*, aquí tenemos algunos ejemplos:

int a [3]; − Declaramos un vector de 3 elementos.

float b [3][2]; − Una matriz de 3 filas y dos columnas.

char c [3][2][5]; − Un array tridimensional de caracteres.

En la inicialización podemos asignarle también valores a la matriz. Por ejemplo, para darles valores a los elementos del vector int a [3] podemos escribirlos en la inicialización de la siguiente forma:

int a [3] = {23, 5, 8}

Donde el vector va a tener los valores 23 para el primer elemento, 5 para el segundo y 8 para el tercero.

Esto mismo se puede aplicar ahora para matrices. Las matrices en C son en realidad un conjunto de vectores puestos uno detrás de otro, por lo que los elementos se almacenan en memoria por filas, tal y como se ve en la figura. Por lo que a la hora de recorrer matrices es mejor hacerlo por fila para no tener que estar continuamente saltando de una dirección a otra de la memoria sino que pasamos de un lugar al siguiente de forma continua (por lo que nuestro programa será más rápido).

La declaración también se realizan por filas. Basándose en esto, para indicar los valores de una matriz como foat a [3][2] haríamos lo siguiente:

foat a [3][2] = { {5.2 , 3.2},

{3.3 , 0},

{8 , 0.67} };

Aunque esto también lo podemos poner en una sola línea, de esta forma vemos perfectamente la forma que va a tener nuestra matriz (esta será de 3 filas y de 2 columnas).

11. Cadenas de Caracteres

En C no existe un tipo de datos específico para cadena de caracteres (o lo que es lo mismo, una serie de caracteres que forman una palabra u oración). Para ello se usan vectores que manejan elementos del tipo *char* (carácter). Estos caracteres, como cualquier elemento en un vector, están posicionados en memoria uno detrás de otro. Con la particularidad de que el último carácter de la cadena se reserva siempre para el denominado **cáracter nulo**, que se representa por el símbolo \0, y nos indica que es el final de la cadena.

La inicialización es por tanto exactamente igual que la de un vector:

```
char nombre [] = cadena;
```

La descripción de los parámetros es:

nombre − es el nombre con el que vamos a llamar a la cadena.

cadena − es el valor que va a contener la cadena, es decir los caracteres que están dentro de las comillas son los que se van a introducir dentro de la cadena.

Los caracteres que pongamos entre las comillas se introducirán uno detrás del otro en memoria, y para poder referirnos a uno de ellos (como en cualquier vector) solo tenemos que escribir la posición que ocupan dentro de la cadena. El caracter final será siempre el \0, que se introduce de forma automática, por lo no tenemos que escribirlo nosotros al declarar la matriz. Este es un ejemplo para ver como nos podemos referir a un carácter de la cadena, básicamente el procedimiento es igual que con vectores:

```
main(){

char cadena [20] = Hola;

int i;

while (cadena[i] != `\0')

{

printf (%c, cadena[i]);

i++;

}

}
```

Lo que hacemos en este ejemplo es recorrer los caracteres de la cadena por medio de un bucle, este otro programa recorre la cadena elemento a elemento:

```
main(){

char cadena [20];

cadena [0] = `h';

cadena [1] = `e';

cadena [2] = `\0';

}
```

11.1. Función Scanf

Con esta función podemos hacer que el programa espere a que el usuario introduzca un dato por el ordenador (esto se indica mediante un cursor parpadeante). scanf va a recoger todos los datos de un tipo específico hasta que el usuario introduzca un espacio.

La sintaxis de la función es la siguiente:

scanf (%tipo, &nombre);

Descripción:

%tipo − es el tipo de dato que queremos que scanf recoja, este lo tenemos que dar según la clave que usa scanf y printf, que un símbolo % seguido de una de estas letras: s (tipo cadena), d (entero), c (carácter), f (entero) ...

&nombre − ponemos la dirección de memoria donde queremos almacenar los datos introducidos por el usuario. Como dicha dirección no la conocemos habitualmente podemos valernos del símbolo & delante del nombre de la variable que vamos a usar, ya que este símbolo sirve para indicar la dirección de una variable. Tenemos que recordar que si el tipo de dato que introducimos es una cadena de caracteres, al poner el nombre de la cadena ya estamos poniendo la dirección de dicha cadena (ya que el nombre es en realidad una variable que contiene la dirección del primer carácter de la cadena en cuestión, esto se explicará mejo más adelante), por lo que no hay que ponerle & delante.

Unos de los problemas de scanf es que al introducir cadenas el usuario no puede introducir el espacio, ya que scanf solo recoge la cadena hasta un espacio. Lo que si se puede hacer es recoger más de una cadena a la vez, para ello escribiremos la función de esta forma:

scanf (%s %s %s, nombre, apellido1, apellido2);

(Las variables nombre, apellido1 y apellido2 son cadenas de caracteres).

De esta forma la primera palabra introducida por el usuario se almacenará en la variable nombre, la segunda en apellido1 y la tercera en apellido2.

Otra consecuencia de esta situación es que la información introducida por el usuario que no recoge un scanf la va a recoger el siguiente que aparezca en el programa, para evitar esto es recomendable escribir fflush(stdin); después de cada scanf para eliminar los datos almacenados en el buffer de entrada.

12. Ámbito de las variables

Cuando declaramos una variable de cualquier tipo debemos de fijarnos que esta solo se inicializa en la función en la que estemos trabajando. Es decir, las variables que se declaran en la función principal no van a estar declaradas en cualquier otra función, por lo que en dichas funciones debemos de declarar otras variables. El ámbito de cada variables es por consiguiente la función en la que está declarada, estas son las variables de ámbito local. También podemos iniciar variables con ámbito global, las cuales se declaran fuera de la función principal y el resto de las funciones. Este último tipo de variables no debe usarse salvo raras excepciones y para declarar constantes.

El siguiente ejemplo explica como funciona el ámbito de las variables:

```
int a;

main (){

int b = 5;

a = b;
```

```
funcion(b);

a *= 2

}

funcion (int b) {

a = a + b;

return b;

}
```

Vemos el problema de usar variables globales, la variable a es de ámbito global y aunque parezca que al final de main su valor es 10, en realidad es 20 (ya que a ha sido modificada en funcion). Como vemos, controlar el valor de las variables globales es difícil o casi imposible.

13. Estructuras y registros

Una estructura es una secuencia de variables (denominados miembros de la estructura) que pueden tener tipos diferentes.

La declaración de una estructura tiene la siguiente forma:

```
struct {

tipo identificador1;

tipo identificador2;

...

...

...

tipo identificadorN;

} identificador_estructura;
```

Descripción:

tipo identificador1 ... tipo identificadorN − estas son todas las variables que queramos que recoja nuestra estructura, cada una tiene un tipo distinto y un nombre para identificarlas (como se ve, la declaración de cada variable es exactamente igual que si fueran independientes).

identificador_estructura − este es el nombre de la variable que recoge nuestra estructura, cuando queramos referirnos a la estructura completa usaremos este identificador.

Vamos a ver un ejemplo de cómo podemos referirnos a una estructura y a cada una de sus partes:

```
#include <stdio.h>

#include <string.h>

main (){

struct {

char fabricante[20];

int cantidad;

float precio_unidad;

} resistencias;

char fab[20] = philifs;

resistencias.cantidad = 10;

resistencias.precio_unidad = 9.8;

strcopy (resistencias.fabricante, fab);

}
```

(La última orden simplemente copia el contenido de la cadena fab a la cadena resistencias.fabricante, estos comandos ya se verán más adelante)

Como vemos, para referirnos a una variable de la estructura tenemos que poner el nombre de esta seguido de un punto y el nombre de la variable que queramos tratar.

Al trabajar con estructuras, tenemos que declararlas en cada función en la que queramos usarlas (como cualquier variable). Pero, aunque esto parezca muy engorroso, ya que para declarar una estructura necesitamos varias líneas de código, podemos valernos de algunas opciones que presenta C para hacer este proceso más rápido. Una de estas formas es darle a la declaración de la estructura en si un nombre (que se denomina etiqueta), por lo que cada vez que declaremos una estructura solo tenemos que poner esta etiqueta para indicar que las variables que contiene son las mismas que la que hemos etiquetado. Esta declaración modelo tiene por tanto que declararse de forma global para que tenga efecto en todas las funciones de nuestro programa, el problema está en que vamos a declarar una variable global con todos sus inconvenientes, por lo que esta variable no debe usarse. Para declarar una estructura y darle a la vez una etiqueta hacemos lo siguiente (lo hacemos de forma global, por lo que esta declaración se hace fuera de cualquier función):

```
struct etiqueta {

tipo identificador1;

tipo identificador2;

...

...
```

```
...

tipo identificadorN;

} no_usar;
```

La estructura no_usar no debe usarse, ya que presenta los inconvenientes de las variables globales. Por otro lado, la etiqueta de la estructura (que puede tener el nombre que queramos) se puede usar luego para declarar otras estructura del mismo tipo, estas nuevas declaraciones si deben hacerse dentro de las funciones:

```
struct etiqueta resgistro1, registro2 ... ;
```

Otra forma de declarar estructuras mucho más rápida es crear un tipo de estructura. Es decir, considerar la estructura como un tipo de datos y no como una variable. Por lo que cuando queramos crear una estructura solo tenemos que decirle que va a tener ese tipo de dato. Para declarar un tipo de datos estructura tenemos que poner estas líneas (por supuesto esta declaración va a ser de tipo global):

```
typedef struct {

tipo identificador1;

tipo identificador2;

...

...

...

tipo identificadorN;

} nombre_del_tipo;
```

Descripción:

nombre_del_tipo − Este es el nombre que va a tener el tipo de dato, hay que resaltar que no es el nombre de una variable, sino de un dato como lo pueden ser float, int u otros.

Y ahora, con este tipo de dato definido, podemos declarar estructuras de forma muy rápida:

nombre_del_tipo estructura1, estructura2 ... ;

13.1 Declaración de estructuras y *arrays*

Las estructuras y los *arrays* son tipos de datos como los de cualquier tipo, por lo que podemos tratarlos como si fueran datos normales. Por eso es necesario ver como podemos combinar estos datos en la medida que podemos hacer *arrays* que contengan estructuras, estructura que tengan en una de sus variables un *array* ... y todas las combinaciones que queramos.

Por ejemplo, supongamos hemos declarado globalmente el siguiente tipo de dato:

typedef struct {

char fabricante[10];

int color;

float valor2;

} registro_piezas;

Luego, dentro de una función, podemos crear un vector que contenga variables del tipo registro_piezas, dicho vector es como cualquier otro, solo que en vez de contener enteros contiene estructuras del tipo declarado:

resigtro_piezas vector1[50];

El empleo de estos vectores de estructurases la potencia real del uso de estructuras en C, ya que en una sola variable tenemos un registro entero con todas las características que queramos.

13.2 Estructuras especiales: union

La union en C es muy similar a la estructura. La diferencia es que la unión se utiliza para almacenar tipos de datos diferentes en una misma zona de memoria, por lo que, la variable de tipo unión no recoge muchos subtipos como la estructura; sino que unas veces va a ser de un tipo y otras de otro distinto. La declaración de esta variable es la siguiente:

```
union etiqueta {

tipo mienbro1;

tipo mienbro2;

...

...

...

tipo mienbroN;

}nombre;
```

Descripción de los parámetros:

etiqueta − Al igual que en las estructuras, podemos identificar la declaración de un union mediante una etiqueta.

tipo mienbro1 ... tipo mienbroN; − Estas son las variables que va a contener union, pero como ya hemos explicado, esta solo va a contener una de esta variables a la vez.

nombre − Es el nombre con el que vamos a identificar al union.

El ejemplo siguiente nos explica como funciona una variable de tipo union:

main (){

```
union{

int entero;

float real;

}entero_o_real;

entero_o_real.entero = 8;

printf (%d %f, entero_o_real.entero, entero_o_real.real);

entero_o_real.real = 9.3;

printf (%d %f, entero_o_real.entero, entero_o_real.real);

}
```

La variable entero_o_real solo recoge uno de sus subtipos a la vez, al asignarle un valor a una de sus variables el resto pasa tomar ese valor (ya que en realidad el resto de los nombres solo son una referencia a la misma posición en memoria). Así que en el primer printf se escribirá 8 8, ya que entero_o_real.entero y entero_o_real.real son siempre la misma variable. En el segundo printf se escribirá 9.3 9.3 como es lógico.

14. Punteros

Los punteros son otra forma de acceder a los datos almacenados en la memoria del ordenador. Este nuevo concepto proporciona a los programadores un mecanismo muy poderoso y flexible para manejar los datos del programa. En C los punteros se tratan como cualquier otro tipo de dato, pero su contenido es la dirección de memoria de otra variable. Para declarar un puntero debemos escribir lo siguiente:

```
tipo *nombre;
```

Descripción:

tipo − Este el tipo de variable a la que va apuntar el puntero.

*nombre − Es el nombre con el que identificaremos al puntero.

Un ejemplo de la declaración de un puntero que apunta a una variable de tipo entero sería el siguiente:

```
int *puntero;
```

El siguiente programa recoge la filosofía de los punteros, vamos a ir explicando cada una de sus partes:

```
# include <stdio.h>

main{

char variable;

char *puntero;
```

valor = `a';

printf (%u => | %d | <= dirección y datos de variable \n, &variable, variable);

...

Hemos declarado dos variables, una es un carácter y otra es un puntero (que apuntará a una variable del tipo carácter). Hemos asignado a la variable de tipo carácter (variable) el valor a, a continuación vemos la posición de memoria en la que se encuentra y su contenido (para expresar la dirección de una variable le ponemos & delante, y para escribir esta dirección correctamente mediante un printf tenemos que poner que escriba una dirección con %u). variable va a ocupar un espacio en memoria tal y como se muestra en la figura.

...

puntero = &valor;

printf (%u => | %u | <= dirección y datos de puntero.
&puntero, puntero);

...

Ahora hacemos algo muy parecido, le asignamos a la variable puntero la dirección de variable. Este segundo printf va a escribir la dirección de memoria en la que está contenido el puntero (ya que como cualquier variable tiene su propia dirección) y su contenido, que es la dirección de variable. Esto queda algo como lo que se muestra en la figura.

Por lo que vemos, a un puntero le podemos asignar tres parámetros: su dirección en memoria, su contenido (que es la dirección a la que apunta) y el contenido de la variable a la que esta apuntando. Estos tres valores se pueden referenciar respectivamente de la siguiente forma:

...

printf (\n Dirección de memoria en la que está el puntero = %d \n, &puntero);

printf (\n Dirección de variable, la cual almacena el puntero = %u \n, puntero);

printf (\n Valor de la variable al que apunta el puntero = %c\n, *puntero);

}

(El primer printf va a imprimir 4562, el segundo 1024 y el tercero a)

Hemos visto que para asignarle a un puntero la dirección de una variable le ponemos a esta un & (puntero = &variable). Pero esta misma asignación, como es lógico, se puede hacer asignándole al contenido que apunta el puntero el de la variable: *puntero = variable.

Mediante los punteros podemos trabajar con una variable desde distintas funciones sin necesidad que esta sea global. Hasta ahora, al cambiar de una función a otra y pasar un valor entre ambas funciones teníamos que definir una variable local en la primera función y otra distinta en la segunda. El problema se encontraba en que si la primera función llamaba a la segunda, al terminar esta última y volver de nuevo a la función primera (la que ha llamado a la otra), todas las modificaciones echas sobre los valores que había realizado la segunda función se perdían (ya que estos estaban en variables locales de la función a la que habíamos llamado).

Pero con los punteros podemos pasar la dirección de la variable en vez de su valor, por lo que las modificaciones que hacemos sobre la variable que pasamos, se realizan en ese mismo lugar (y no en otra dirección de memoria distinta como ocurría antes) por lo que al acabar la segunda función las modificaciones. Este último método es el que llamamos **paso por referencia** (que son las variables de tipo *entrada−salida* que se han visto en algorítmica), y el primero es el **paso por datos** (para las variables de *entrada*).

Vamos a ver como se pasa por referencia mediante un ejemplo. Tenemos el mismo programa que antes, pero ahora vamos a modificar el valor de variable mediante otra función:

```
# include <stdio.h>

main{

char variable;

char *puntero;

valor = `a';

printf (%u => | %d | <= dirección y datos de variable \n, &variable, variable);

puntero = &valor;

printf (%u => | %u | <= dirección y datos de puntero.
&puntero, puntero);

printf (\n Dirección de memoria en la que está el puntero = %d \n, &puntero);

printf (\n Dirección de variable, la cual almacena el puntero = %u \n, puntero);

printf (\n Valor de la variable al que apunta el puntero = %c\n, *puntero);

cambiar(puntero);

printf (%u => | %d | <= dirección y datos de variable \n, &variable, variable);

}
```

La función cambiar recoge un valor cualquiera, solo que ahora este valor es la dirección de una variable. La función es la siguiente:

```
cambiar (char *puntero2) {

*puntero2 = `b';

}
```

Modificamos el contenido al que apunta puntero2 pero como este tiene la misma dirección de puntero, en realidad modificamos el contenido de variable.

Cuando en main se escriba el último printf, variable tendrá la misma dirección pero su contenido será ahora b.

Aquí tenemos algunos programas más que tratan punteros. El siguiente muestra como se tratan los punteros que apuntan a estructura (para referirnos a una estructura mediante un puntero escribimos: puntero −> campo_de_estructura):

```
#include <stdio.h>

tipedef struct {

char nombre[20];

int edad;

}registro;

main(){

registro r, *pr;

scanf (%s, r.monbre);

scanf (%d, &r.edad);

pr = &r;

printf (El nombre es: %s \n, pr −> nombre);

printf (La edad es: %d \n, pr −>edad);

}
```

Este otro programa incluye además como pasamos de una función a otra con este tipo de punteros:

```
#include <stdio.h>

#include <string.h>

tipedef struct {

char nombre[20];

int edad;

}registro;

main(){

registro r, *pr;

int ed = 25;

char nom[]=Juan;
```

```
pr = &r;

dar_valor (pr, nom, ed);

printf (El nombre es: %s %s\n, pr −> nombre, r.nombre);

printf (La edad es: %d %d\n, pr −>edad, r.edad);

}

dar_valor(registro *p, char *c, int e){

(p −> nombre) = c;

(p −> edad) = e;

}
```

Como se ve en el programa, la cadena de caracteres se ha pasado por referencia, ya que el nombre de los *arrays* es siempre un puntero al primer elemento que contengan. Esto quiere decir que los *arrays* solo se pueden pasa por referencia y nunca por valor. Por tanto, una cadena de caracteres se puede declarar así: char c[], o así: char *c. Al pasar las cadenas también podemos poner char c[]en la función de llegada, pero nunca char c[20]. Todo esto también le ocurre a los vectores y a las matrices y en general a todos los *arrays* **n** dimensiónales. Estos son algunos ejemplos más:

```
#include <stdio.h>

main () {

int enteros [4];

Dar_valor (enteros);

for (i=0; i<4;i++) {

printf (%d \n, enteros [i]);

}

}

Dar_valor (int *ent) {

ent [0] = 1;

ent [1] = 2;

ent [2] = 3;

ent [3] = 4;

}
```

Este programa escribiría: 1 2 3 4.

```
main(){

Int enteros [4][2], i, j;

Dar_valor(int ent[][2]);

for (i=0;i<4;i++)

{

for (j=0;j<2;j++)

printf (%d \n, enteros [i][j]);

}

}

Dar_valor (int *ent) {

ent [0] = 1;

ent [1] = 2;

ent [2] = 3;

ent [3] = 4;

}
```

15. Enumerados

Los enumerados son otra forma de expresar los datos en C. Creando un enumerado estos listando distintos miembros, estos miembros son constantes escritas como identificadores que tienen asignados unos valores numétricos. La declaración de un enumerado es la siguiente:

enum etiqueta {lista enumerados}

Descripción:

etiqueta − Es el nombre con el que nos referimos a la estructura del enumerado.

{lista enumerados} − Dentro de los corchetes pondremos cada uno de los nombres de la lista que vamos a enumerar.

Pero esto es la declaración de una estructura enumerada. Más adelante, cuando queramos utilizar una variable de enumerado, la declararemos en cualquier parte del programa para poder usarla. Esta variable se usa como un tipo int. A esta variable le vamos a asignar los valores que se declararon en su correspondiente enumerado (que en realidad son número enteros, pero que al estar definidos en el enumerado se interpretan según la lista que escribimos) Para declararla este tipo de variable tendremos que poner:

enum etiqueta nombre_variable

Descripción:

etiqueta − Es el nombre del enumerado que va a recoger esta variable

nombre_variable − Es el nombre con el que identificaremos a la variable en nuestro programa.

El siguiente programa muestra como se usa un enumerado:

```
#include <stdio.h>

enum codigo_color {negro, marron, rojo, naranja, amarillo};

main () {

enum codigo_color color;

color = marron;

switch (color) {

case negro:

printf (color negro);

break;

case marron:

printf (color marron);

break:

case rojo:

printf (color rojo);

break;

case naranja:

printf (color naranja);

break;

case amarillo:

printf (color amarillo);

break;
```

```
}

}
```

Como el nombre de los colores se evalúa como enteros podemos incluso escribir número (sabiendo que número corresponde a que elementos de la lista). De esta forma podemos sabe que números corresponden a cada color:

```
for (color = negro; color <= amarillo; color++)

printf (constante de color %d \n, color);
```

Vamos a ver otro ejemplo, aquí vemos perfectamente que los enumerados son exactamente igual que números (también se ve que podemos asignar el número que queramos a cada elemento de la lista):

```
#include <stdio.h>

enum división {dividendo=12, divisor=2};

main () {

printf (%d, dividendo/divisor);

}
```

En este caso se imprimiría 6. Otro ejemplo es el siguiente:

```
#include <stdio.h>

enum división {dividendo=12, divisor=2};

main () {

enum división div;

printf (%d, dividendo/divisor);

div = dividendo;

printf (%d, div);

}
```

Ahora se imprimiría 6 y luego 12.

16. Tratamiento de Cadenas

Recordemos que el nombre de las cadenas son punteros que apuntan al primer carácter. En el siguiente ejemplo se ve que el nombre de la cadena es exactamente igual que un puntero (tal y como se ve en el esquema del almacenamiento de memoria):

```
main () {
```

```
char cadena [ ] = Hola;

char *ptr;

ptr = cadena;

printf (la cadena es %s \n, cadena);

printf (%c \n, *ptr);

printf (%c \n, *(ptr+1));

printf (%c \n, *(ptr+2));

printf (%c \n, *(ptr+3));

}
```

El primer printf escribiría la cadena como hemos visto ya. Los siguientes printf irían escribiendo cada uno un