LENGUAJE C

Apuntes realizados por Marta Fernández de Arriba

1. Introducción al lenguaje C

El lenguaje C fue creado en 1972 por Dennis Ritchie, en los laboratorios de la Bell Telephone y ha sido estrechamente asociado con el Sistema Operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como el propio compilador C fueron escritos en C.

Este lenguaje está inspirado en el lenguaje B escrito por Ken Thompson en 1970 con intención de recodificar el UNIX, que en la fase de arranque estaba escrito en ensamblador, en vistas a su transportabilidad a otras máquinas.

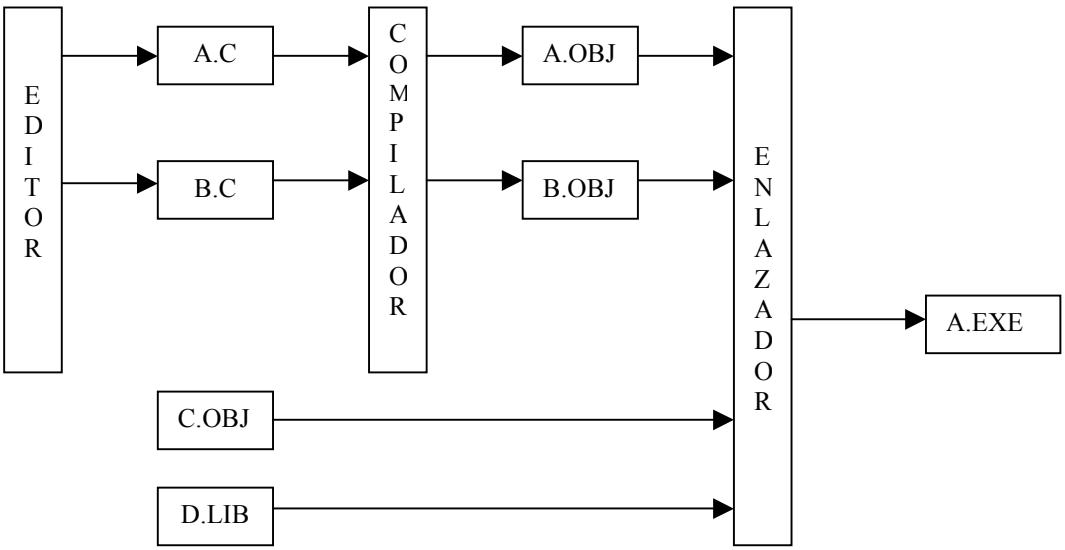
Realización de un programa en C

Para crear un programa en C se deben seguir estos pasos:

1.- Escribirlo en uno o más ficheros de texto de una forma reconocible por el compilador (código fuente, con extensión .C).

2.- Estos ficheros fuente se compilan creando ficheros de código objeto (.OBJ), código máquina con significado para el microprocesador. Cada uno de estos ficheros compilados se llaman módulos.

3.- Este código objeto no está completo, le falta toda una serie de rutinas que están en las librerías (.LIB) y a las que por ahora solo hace referencia. Además, los programas grandes suelen tener varios módulos separados. Para unirlo todo lo enlazamos con el montador o enlazador (linker) que producirá un fichero ejecutable (.EXE) por el sistema operativo



2.- Estructura de un programa en C

Todo programa en **C** está constituido a base de funciones.
Un programa comienza en la función **main ( )** , desde la cual es posible llamar a otras funciones.

Cada función estará formada por la cabecera de la función, compuesta por el nombre de la misma, así como la lista de argumentos (si los hubiese), y delimitado entre llaves la declaración de las variables a utilizar y la secuencia de sentencias a ejecutar.

```
/*comentarios  que pueden ir en cualquier parte*/
inclusión de archivos  //forman parte de las directrices del preprocesador
constantes      // también forman parte de las directrices del preprocesador
variables globales
main( ) {
    variables locales
    secuencia de sentencias // Declaración de todas las instrucciones que conforman el programa principal
 }

  una_funcion ( lista de argumentos ) {
  /* Función creada por el programador */
     variables locales
     secuencia de sentencias
  }
```

Ejemplo:

```
/* Programa que calcula el área de un círculo.
   Y muestra las partes comunes de todos los programas en C*/

# include <stdio.h>
# include <conio.h>
# define PI 3.1416
# define CUADRADO(X) ((X)*(X))
float area;
float area_de_circulo (float);
main (void)
{
  float radio;

  printf(" Programa que calcula el área de un circulo.\n");

  printf ("\t Dime el radio = ");
  scanf("%f", &radio);
  area = area_de_circulo( radio );  //llama a la función
  printf ("El área del círculo de radio %f es =%f", radio, area);
  printf("\nPulsa cualquier tecla para finalizar...");
  getch( );
  return 0;
}

float area_de_circulo( float r)
{
      return (PI * CUADRADO(r));
}
```

3. Elementos del lenguaje C

3.1 Comentarios

Secuencia de caracteres cualesquiera encerrados entre los símbolos **/*** y ***/** o que siguen hasta el carácter fin de línea a los símbolos //. Se emplean para documentar un programa. El compilador tratará los comentarios como un espacio en blanco.

```
/* Ejemplo de comentario */
/* Otro ejemplo de comentario con varias
   líneas
   bla, bla, bla
*/
```

```
// Ejemplo de comentario
// Otro ejemplo de comentario con varias
// líneas
// bla, bla, bla
```

3.2 Directrices del preprocesador

El preprocesador de C, es un programa que procesa el código fuente como primer paso en la compilación. Las directrices para el preprocesador son utilizadas para hacer programas fuente fáciles de cambiar y de compilar en diferentes situaciones. Una directriz va precedida del símbolo # e indica al preprocesador una acción específica a ejecutar.

Sus funciones son las siguientes:

Sustitución de cadenas simples (Constantes)

```
#include <stdio.h>
#define pi 3.1416
// Sustituye un nombre por un valor numérico
#define escribe printf
// o  por una cadena de caracteres
void main( )
/* Calcula el área */
{
   int r;
   float area;
   escribe("Introduce el radio: ");
   scanf("%d",&r);
   area=pi*r*r;
   escribe("El area es: %f", area);
}
```

Macros con argumentos

```
#include <stdio.h>
#define pi 3.1416
#define area(a) (pi*a*a)
void main( )  // Calcula el área
{
   int r;
   printf("Introduce el radio: ");
   scanf("%d",&r);
   printf("El area es: %f",area(r));
}
```

Inclusión de archivos

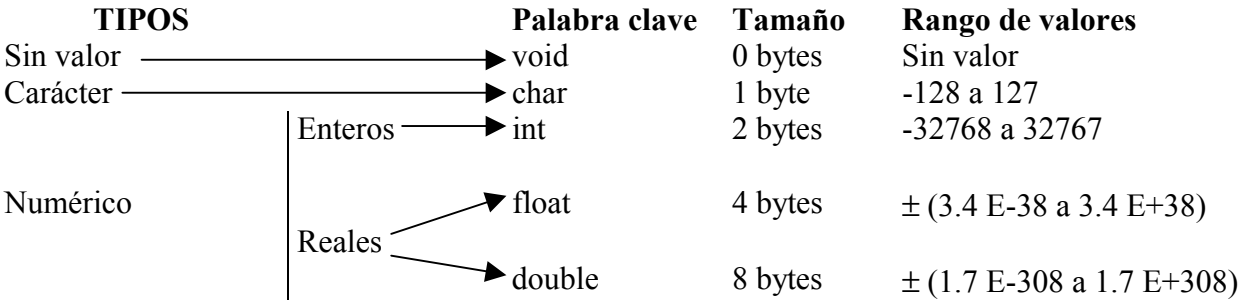
Utilizando la directiva **#include** podemos incluir librerías que se encuentran en otros ficheros.

#include "nombre_fichero" Busca en el directorio actual
#include <nombre_fichero> Busca fuera del directorio actual (donde esté indicado en el compilador)

Ejemplos:

#include "misfunc.h"
#include "c:\includes\misfunc.h" //Indicamos al compilador la ruta donde se encuentra el fichero
#include <misfunc.h> //se encuentra en el directorio por defecto del compilador

3.3 Tipos de datos



Es posible modificar el rango de valores de un determinado tipo de variable, utilizando los calificadores de tipo.

Calificador de tipo

signed (valor por defecto) — Indica que la variable va a contener signo (aplicable a char e int)

	tamaño	rango de valores
signed char	1 byte	-128 a 127
signed int	2 bytes	-32768 a 32767

unsigned — La variable no va a llevar signo (sólo valores positivos). Sólo aplicable a char e int.

	tamaño	rango de valores
unsigned char	1 byte	0 a 255
unsigned int	2 bytes	0 a 65535

short (valor por defecto) — Rango de valores en formato corto.

	tamaño	rango de valores
short char	1 byte	-128 a 127
short int	2 bytes	-32768 a 32767

long — Rango de valores en formato largo (ampliado).

	tamaño	rango de valores
long int	4 bytes	-2147483648 a 2147483647
long double	10 bytes	-3.36 E-4932 a 1.18 E+4932

Es posible combinar calificadores entre sí:

unsigned short int = unsigned int
signed short int = short int = signed int = int
signed long int = long int = long
unsigned long int = unsigned long **4 bytes** **0 a 4294967295**

3.4 Variables

Las variables se emplean para almacenar datos (numéricos o caracteres) en localidades de memoria, pudiendo variar su contenido a lo largo del programa.

Una variable es un tipo de dato, referenciado mediante un identificador (que es el nombre de la variable). Cada variable sólo podrá pertenecer a un tipo de dato.

Declaración de una variable: **[calificador de tipo] <tipo de dato> <identificador>;**

Es posible inicializar y/o declarar más de una variable del mismo tipo a la vez:

[calificador de tipo] <tipo de dato> <identificador1>,<identificador2>=<constante>,
<identificador3>=<constante>,<identificador4>;

Las variables pueden ser *globales* o *locales*, dependiendo de dónde se declaren. Una variable global se declara antes del **main( )** y puede ser utilizada en cualquier parte del programa. Una variable local se declara dentro de la función (recordar que main es una función) y sólo se puede utilizar en dicha función.

Modificador de acceso

const <tipo de dato> <identificador>=<constante>;

Las variables de tipo **const** no pueden ser cambiadas su valor durante la ejecución del programa.

3.5 Identificadores

Los identificadores son nombres dados a constantes, variables y funciones.
Sintaxis: letra o _ [letra o dígito o _]...

Un identificador consta de uno o más caracteres (letras, dígitos y el carácter de subrayado) y el primer carácter debe de ser una letra o el carácter de subrayado. Las letras pueden ser mayúsculas o minúsculas y se consideran como caracteres diferentes. (Ojo! La ñ no está permitida).

3.6 Palabras clave

Existen una serie de palabras reservadas, que tienen un significado especial para el compilador, que no se pueden utilizar como identificadores.

El lenguaje C tiene las siguientes palabras clave:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

3.7 Constantes

Una constante en C puede ser un número, un carácter o una cadena de caracteres.

El lenguaje C permite especificar un entero en base 10, 8 y 16.
⇒ Base 10: puede tener uno o más dígitos (0 a 9), de los cuales el primero es distinto a cero. Si la constante es positiva, el signo + es opcional.
⇒ Base 8: puede tener uno o más dígitos (0 a 7) precedidos por un cero.
⇒ Base 16: puede tener uno o más caracteres (0 a 9 y A a F) precedidos por 0x o por 0X

Una constante real tiene el siguiente formato: [dígitos][.dígitos][E o e [+ o -]dígitos]
donde **dígitos** representa 0 o más dígitos del 0 al 9, y **E** o **e** es el símbolo de exponente que puede ser positivo o negativo. Si la constante real es positiva no se especifica el signo y si es negativa lleva el signo menos (-).

Las constantes de un solo carácter estás formadas por un único carácter encerrado entre comillas simples. Una secuencia de escape es considerada como un único carácter.

Una constante de caracteres es una cadena de caracteres encerrados entre comillas dobles.

3.8 Secuencias de escape

Una secuencia de escape está formada por el carácter \ seguido de una letra o una combinación de dígitos. Son utilizadas para acciones como nueva línea, tabulaciones y para representar caracteres no imprimibles.

Secuencia	Nombre
\n	Nueva línea
\t	Tabulación horizontal
\v	Tabulación vertical (para impresoras)
\b	Retroceso (backspace)
\r	Retorno de carro
\f	Salto de página (para impresoras)
\a	Bell (alerta, pitido)
\'	Apóstrofe
\"	Comillas dobles
****	Backslash (muestra el carácter \ )
\0	Fin de una cadena de caracteres
\ddd	Carácter ASCII. Representación octal
\xdd	Carácter ASCII. Representación hexadecimal

3.9 Operadores

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, unarios, lógicos, relacionales, lógicos para manejo de bits y de asignación.

Pueden ser unarios o binarios:
⇒ *binarios*: **<variable1><operador><variable2>**
⇒ *unarios*: **<variable><operador>** y al revés, **<operador><variable>.**

Operadores aritméticos (binarios)

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales.
-	Resta. Los operandos pueden ser enteros o reales.
*****	Multiplicación. Los operandos pueden ser enteros o reales.
/	División. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros, el resultado es entero. En el resto de los casos el resultado es real.
%	Módulo o resto de una división entera. Los operandos tienen que ser enteros.

Operadores unarios

Operador	Operación
++	Incremento (suma uno)
--	Decremento (resta uno)
-	Cambio de signo

Operadores relacionales (binarios)

Operador	Operación
<	Primer operando menor que el segundo.
>	Primer operando mayor que el segundo
<=	Primer operando menor o igual que el segundo
>=	Primer operando mayor o igual que el segundo
==	Primer operando igual que el segundo
!=	Primer operando distinto del segundo

El resultado que devuelven estos operadores es **1** para Verdadero y **0** para Falso. Si hay más de un operador se evalúan de izquierda a derecha.

Operadores lógicos (binarios)

Operador	Operación
&&	AND. Da como resultado el valor lógico 1 (cierto) si ambos operandos son distintos de 0 (falso). Si uno de ellos es 0 (falso) el resultado es el valor lógico 0 (falso).
||	OR. El resultado es 0 (falso) si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1 (cierto).
!	NOT. EL resultado es 0 si el operando tiene un valor distinto de 0, y 1 en caso contrario.

Expresiones de Boole.

Una expresión de Boole da como resultado los valores lógicos 0 o 1. Los operadores que intervienen en una expresión de Boole pueden ser: operadores lógicos y operadores de relación.

```
int a,b;
float x=15, y=18, z=20;
a = x ==y; /*a=0 */
b = (x < y) && (y <=z); /* b = 1 */
```

Operadores lógicos para manejo de bits (binarios)

Operador	Operación	Ejemplo
&	Operador AND a nivel de bits.	**10100101 & 0111011 = 00100100**
|	Operador OR a nivel de bits.	**10100101 | 0111011 = 11110111**
^	Operador XOR a nivel de bits.	**10100101 ^ 0111011 = 11010011**
<<	Desplazamiento a la izquierda.	**10001010 << 2 = 00101000**
>>	Desplazamiento a la derecha.	**10001010 >> 2 = 00100010**
~	Complemento a uno (NOT).	**~10001010 = 01110101**

Operadores de asignación (binarios)

La mayoría de los operadores aritméticos binarios y lógicos para manejo de bits tienen su correspondiente operador de asignación:

Operador	Operación
=	Asignación simple
+=	Suma más asignación. Ejemplo: a+=3 equivale a=a+3
-=	Resta más asignación
***=**	Multiplicación más asignación
/=	División más asignación
%=	Módulo más asignación
<<=	Desplazamiento a izquierda más asignación
>>=	Desplazamiento a derecha más asignación
&=	Operador AND sobre bits más asignación
|=	Operador OR sobre bits más asignación
^=	Operador XOR sobre bits más asignación

Prioridad y orden de evaluación

La tabla que se presenta a continuación, resume las reglas de prioridad y asociatividad de todos los operadores. Los operadores escritos sobre una misma línea tienen la misma prioridad. Las líneas se han colocado de mayor a menor prioridad.

Operador	Asociatividad
() [] . ->	Izquierda a derecha
- ~ ! * & ++ -- sizeof(tipo)	Derecha a izquierda
*** / %**	Izquierda a derecha
+ -	Izquierda a derecha
<< >>	Izquierda a derecha
< <= > >=	Izquierda a derecha
== !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
|	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha
?:	Derecha a izquierda
= += -= *= /= %= <<= >>= &= |= ^=	Derecha a izquierda
,	Izquierda a derecha

Conversión de tipos

Cuando los operandos dentro de una expresión son de tipos diferentes, alguno de ellos puede sufrir una conversión de tipo de manera que estará en concordancia con el tipo de operando.

La jerarquía de los tipos básicos es la siguiente: char < int < long < float < double

⇒ Si el operando de mayor jerarquía es un double, los otros operandos se transforman en double y el resultado de la operación es un double.
⇒ Si el operando de mayor jerarquía es un float , los otros operandos se transforman en float y el resultado de la operación es un float.

.....

Si tenemos un operador de asignación, cualquiera que sea el tipo de la derecha del símbolo de asignación se convierte al tipo de la izquierda. Es decir, la variable de la izquierda siempre tiene precedencia.

CAST (tipo) expresión
Nos permiten forzar que una variable se comporte como si fuera de otro tipo en un momento dado.

variable = (int) 3.1459;

4. Funciones de E/S

Las funciones de Entrada/Salida permiten comunicarnos con el exterior. No forman parte del conjunto de sentencias de C, sino que pertenecen al conjunto de funciones de la librería *stdio.h*

4.1 Sentencia printf

Escribe con formato una serie de caracteres y/o valores de variables en la salida estándar (stdout). Su sintaxis es: **printf(cadena de control,arg1,arg2...);**

En la cadena de control se indicará el formato en que se mostrarán los argumentos posteriores (en el caso de que existan), ya que también se puede introducir una cadena de texto (sin argumentos) o combinar ambas posibilidades

⇒ Cadena de texto (sin argumentos)

```
printf("Hola mundo");
```

Con secuencias de escape: `printf("\t Dirección: C\\San José, \"Gijón\" \n");`

⇒ Si utilizamos argumentos, deberemos indicar en la cadena de control tantos modificadores como argumentos vayamos a presentar.
El modificador está compuesto por el carácter **%** seguido por un carácter de conversión, que indica de qué tipo de dato se trata.

```
printf("El %d es su valor",numero);
printf("%d + %d = %d",a,b,a+b);
printf("%d %% %d = %d",a,b,a%b); //Necesario para mostrar el %
```

Modificador	Salida
%c	Carácter
%s	Cadena de caracteres
%d %i	Entero decimal con signo
%u	Entero decimal sin signo
%o	Entero octal sin signo, sin el 0 al principio
%x	Entero hexadecimal sin signo, sin el 0x al principio
%e	Número de punto flotante en notación científica. (formato [-]d.dddE±dd)
%f	Número de punto flotante en notación decimal sin exponente. (formato [-]ddd.ddd)
%g	Usa %f o %e, el que sea más corto

Indicador	Significado
número	Indica la longitud mínima del campo. Si el dato a escribir es menor que la longitud, se rellena con blancos a la izquierda del dato. Si ocupa más de lo especificado, el ancho es incrementado en lo necesario.
0número	Si quiero rellenar con ceros a la izquierda del dato (en el caso de que sea menor)
-	El valor se ajusta a la izquierda y se rellena con blancos el espacio sobrante a la derecha.
+	Un signo precederá a cada valor numérico con signo.
nºdig.nºdec	Muestra el número de dígitos con al menos el número de decimales especificado.
# nºdig.nºdec	Muestra el punto decimal incluso si no tiene decimales

El orden es: %[-|+|#][numero][.precisión][c|d|...|g]

4.2 Sentencia scanf

Lee datos del canal de entrada estándar (stdin) y su sintaxis es: **scanf(control,arg1,arg2...);**

En la cadena de control se indican los tipos de datos que se quieren leer (utilizando los modificadores). Los datos se convierten y se almacenan en las variables que se pasan como argumentos.

Todos los argumentos son direcciones de variables (posiciones de la memoria del ordenador) en las que se almacenarán los valores leídos por teclado. Para indicarle esta posición de memoria se utiliza el símbolo **&** antes del nombre de cada variable.
EXCEPCIÓN: no se antepone el símbolo **&** antes del nombre de una variable cadena de caracteres.

```
#include <stdio.h>
main()
 {
    char nombre[10];
    int edad;
     printf("Introduce tu nombre: ");
     scanf("%s",nombre);
     printf("Introduce tu edad: ");
     scanf("%d",&edad);
     printf("\nTe llamas %s y tienes %d años", nombre, edad);
}
```

La lectura de un dato termina cuando scanf() encuentra un carácter de espacio en blanco, un tabulador o un carácter de nueva línea. Así, scanf("%d%d",&i,&j); //entrada 10 20 INTRO

Dado que scanf() lee datos delimitados por blancos, si queremos leer cadenas de caracteres que contengan espacios en blanco, se debe sustituir la especificación de formato **%s** por **%[^\n]s**, que indica que se lean caracteres hasta encontrar un carácter \n.

Un conjunto de caracteres entre [ ], indica leer caracteres hasta que se lea uno que no esté especificado en el conjunto. El efecto inverso se consigue anteponiendo al conjunto de caracteres el símbolo ^, es decir [^caracteres].

El * después de **%** y antes del código del formato, indica que lea un dato del tipo indicado y no lo almacene. Así, si introducimos 5/2, scanf("%d%*c%d",&i,&j); //almacena en i un 5 y en j un 2.

Se puede emplear un modificador de longitud que consiste en un número situado entre el signo % y el código de formato. Dicho número indica la longitud máxima que se va a leer, por ejemplo, scanf("%5s",cad); lee un máximo de 5 caracteres, si la entrada es mayor, los restantes permanecerán en el canal. Lo mismo ocurre si se trata de números, scanf("%2d%d",&i,&j);

Si en la cadena de control aparece cualquier carácter que no sea de los mencionados anteriormente, se usará para que scanf() busque el carácter en los datos de entrada, por ejemplo, con la entrada 5-2, scanf("%d-%d",&i,&j); //almacena en i un 5 y en j un 2.
En este caso si no encuentra un -, suspende la lectura.

Los espacios, tabuladores o caracteres de nueva línea los interpreta como una señal para seguir leyendo. Si se pone scanf("%d%d\n",&i,&j); después de introducir dos valores sigue esperando otro distinto de espacio, tabulador o carácter de nueva línea.

4.3 Funciones de manejo de caracteres

getchar lee un carácter introducido desde el teclado y devuelve el valor ASCII del carácter cuando se pulsa ENTER.
Devuelve EOF cuando se pulsa CTRL+Z.

getch lee un carácter desde el teclado y no lo visualiza por el monitor (no produce ECO).

getche lee un carácter desde el teclado visualizándolo por el monitor (produce ECO).

putchar escribe el carácter en la salida estándar stdout (pantalla).

```
char letra, caracter=23;

letra = getchar( );  // equivale scanf("%c",&letra);

letra = getche( ); // similar pero sin pulsar INTRO tras la pulsación de la tecla

letra = getch( );  // similar pero sin pulsar INTRO tras la pulsación de la tecla
                   // además no se ve la tecla que se ha pulsado...

putchar(letra);

putchar(caracter);
```

5. Sentencias de control

SENTENCIA if

Toma una decisión referente a la acción a ejecutar en un programa, basándose en el resultado (verdadero o falso) de una expresión.

```
if (expresión) sentencia1;
  [else sentencia2;]
```

expresión debe ser una expresión numérica, relacional o lógica. El resultado que se obtiene al evaluar la expresión es verdadero (distinto de cero) o falso (cero).

sentencia1
sentencia2 representan una sentencia simple o compuesta. Cada sentencia simple debe estar separada de la anterior por un **;**
[else sentencia2] los corchetes indican opcionalidad, puede aparecer o no.

Anidamiento de SENTENCIAS if

Las sentencias **if..else** pueden estar anidadas. Es decir, como sentencia1 o sentencia2, puede escribirse otra sentencia if.

Cuando en un programa aparecen anidadas sentencias **if...else**, la regla para diferenciar cada una de estas sentencias es que cada **else** se corresponde al **if** más próximo que no haya sido emparejado, así:

```
if (expresión1)
{
 if (expresión2)
     sentencia1;
}
else sentencia2;
```

```
if (expresión1)
 if (expresión2)
    sentencia1;
 else sentencia2;
```

ESTRUCTURA if

La estructura presentada a continuación, aparece con bastante frecuencia, como consecuencia de las sentencias **if anidadas**.

```
if (expresión1)
   sentencias1
  else if (expresión2)
         sentencias2
        else  if (expresión3)
                 sentencias3
                    .........
                  else sentenciasN
```

sentencias1, ..., sentenciasN
representan sentencias simples o compuestas.

Si se cumple la **expresión1**, se ejecutan las sentencias1 y si no se cumple, se examinan secuencialmente las expresiones siguientes hasta **else**, ejecutándose las sentencias correspondientes al primer **else if**, cuya expresión sea cierta. Si todas las expresiones son falsas, se ejecutan las **sentenciasN** correspondientes al último else.

Operador condicional ?

Sintaxis: **expresión1? expresión2: expresión3**
Si la expresión 1 es cierta se evalúa la expresión2, sino se evalúa la expresión3.

```
max = (a>b)?a:b;
```

```
if (a>b)
  max =a;
  else max=b;
```

Sentencia switch

Esta sentencia permite ejecutar una de varias acciones, en función del valor de una expresión.

```
switch (expr-test){
 case cte1: [sentencias1;]
            [break;]
case cte2: [sentencias2;]
           [break;]
        ..............
[default: sentenciasN;]
}
```

test Puede ser una variable entera o de caracteres

sentenciasi Puede incluir una o más sentencias sin necesidad de ir entre llaves, ya que se ejecutan todas hasta que se encuentra la sentencia **break**.

La sentencia switch evalúa la expresión entre paréntesis y compara su valor con las constantes de cada **case**. La ejecución de las sentencias del cuerpo de la sentencia **switch** comienza en el **case** cuya constante coincida con el valor de la **expr-test** y continúa hasta el final del cuerpo o hasta que se encuentre una sentencia que transfiera el control del cuerpo (por ejemplo **break**).

Si no existe un valor igual al valor de la **expr-test**, entonces se ejecutan las sentencias a continuación de **default**, si esta cláusula ha sido especificada.

Sentencia while

Ejecuta una sentencia, simple o compuesta, cero o más veces, dependiendo del valor de una expresión.

```
while (expresión)
  [sentencia;]
```

expresión es cualquier expresión numérica, relacional o lógica.
sentencia es una sentencia simple o compuesta.

1. Se evalúa la expresión
2. Si el resultado es falso (cero), la sentencia no se ejecuta y se pasa a ejecutar la siguiente sentencia en el programa.
3. Si es cierto (distinto de cero), se ejecuta la **sentencia** del while y el proceso se repite comenzando en el punto 1.

Sentencia do

Ejecuta una sentencia, simple o compuesta, una o más veces, dependiendo del valor de una expresión.

```
do
  [sentencia;]
while (expresión);
```

expresión es cualquier expresión numérica, relacional o lógica.
sentencia es una sentencia simple o compuesta.

1. Se ejecuta la sentencia o el cuerpo de la sentencia do.
2. Se evalúa la expresión
3. Si el resultado es falso (cero), se pasa a ejecutar la siguiente sentencia en el programa.
4. Si el resultado de la evaluación es cierto (distinto de cero), se ejecuta la **sentencia** del while y el proceso se repite comenzando en el punto 1.

Sentencia for

Cuando se desea ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido.

```
for ([inicialización];[condición];[progresión])
        sentencia;
```

inicialización	variables a las que se le asigna un valor, si hay más de una se separan por **,**
condición	es una expresión de Boole (operandos unidos por operadores relacionales y/o lógicos). Si se omite, se supone que siempre es **cierta**.
progresión	es una expresión cuyo valor evoluciona en el sentido de que se de la condición para finalizar la ejecución de la sentencia **for**.
sentencia	es una sentencia simple o compuesta.

1. Se inicializan las variables
2. Se evalúa la expresión de Boole (condición)
3. Si el resultado es falso (cero), se pasa a ejecutar la siguiente sentencia en el programa.
4. Si el resultado de la evaluación es cierto (distinto de cero), se ejecuta la **sentencia** del for, se evalúa la expresión que da lugar a la progresión de la condición y se vuelve al punto 2.

Sentencia break
Se utiliza para terminar la ejecución de un bucle o salir de una sentencia switch.

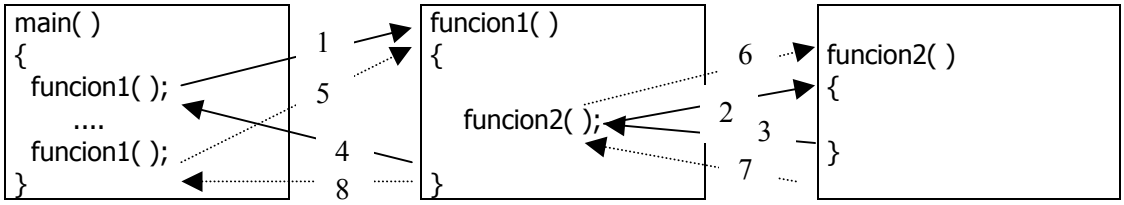
Sentencia continue
Se utiliza dentro de una sentencia do, while o for. Cuando el programa llega a una sentencia **continue** no ejecuta las líneas de código que hay a continuación y salta a la siguiente iteración del bucle.

Sentencia goto
Permite al programa saltar hacia un punto identificado con una etiqueta, pero el buen programador debe prescindir de su utilización. Es una sentencia muy mal vista en la programación en 'C'.

6. Funciones

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa en C consta al menos de una función, la función **main**. Además de ésta, puede haber otras funciones cuya finalidad es descomponer el problema general en subproblemas más fáciles de resolver y de mantener. La ejecución de un programa comienza por la función **main**.

Cuando una función se llama, el control se pasa a la misma para su ejecución, y cuando ésta finaliza, el control es devuelto al módulo que llamó, para continuar con la ejecución del mismo, a partir de la sentencia que efectúo la llamada.



La sintaxis de una función es la siguiente:

```
[clase] [tipo] nombre_función ( lista de parámetros formales)
{
  [declaraciones]
   sentencias;  // conjunto de sentencias a ejecutar cuando se realice
                //  la llamada a la función
}
```

clase Define el ámbito de la función (desde dónde puede ser llamada). Puede ser:
- ⇒ **static** es visible sólo en el fichero fuente donde está definida.
- ⇒ **extern** es visible para todos los ficheros fuente que componen el programa. Valor por defecto

tipo Indica el tipo de valor devuelto por la función. Puede ser cualquier tipo fundamental, estructura o unión. Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor de tipo entero (**int**).
Si no queremos que retorne ningún valor deberemos indicar el tipo vacío (**void**).

nombre_función Identificador que indica el nombre de la función.

Lista de parámetros formales es una secuencia de declaraciones de parámetros separados por comas, y encerrados entre paréntesis. La sintaxis es la siguiente: **tipo [identificador] [, ...]**

tipo Indica el tipo del argumento, el cual puede ser cualquier tipo fundamental, unión, estructura, puntero o array.

identificador es el nombre dado al parámetro

Los parámetros formales son variables locales que reciben un valor, y el número de ellos puede ser variable. Este valor se lo enviamos al hacer la llamada a la función.

Pueden existir funciones que no reciban argumentos, en cuyo caso la lista de parámetros formales puede ser sustituida por la palabra clave **void**.

Cuerpo de la función

El cuerpo de una función está formado por una sentencia compuesta que contiene sentencias que definen lo que hace la función, También puede contener declaraciones de variables utilizadas en dichas sentencias. Estas variables, por defecto, son locales a la función.

Se debe tener en cuenta, que tanto los argumentos de la función como sus variables locales se destruirán al finalizar la ejecución de la misma.

Sentencia return

Cada función puede devolver un valor cuyo tipo se indica en la cabecera de función. Este valor es devuelto a la sentencia de llamada a la función, por medio de la sentencia return, cuya sintaxis es: **return(expresión)**

Si la sentencia return no se especifica o se especifica sin contener una expresión, la función no devuelve ningún valor. El valor devuelto por la función debe asignarse a una variable, de lo contrario, el valor se perderá.

Llamada a una función

La llamada a una función tiene la forma: **[variable =] nombre_función([parámetros actuales])**

variable Especifica donde va a ser almacenado el valor devuelto por la función.

nombre_función Identificador que indica el nombre de la función llamada

parámetros actuales Lista de expresiones separadas por comas. Las expresiones son evaluadas y convertidas utilizando las conversiones aritméticas usuales. Los valores resultantes son pasados a la función y asignados a sus correspondientes parámetros formales. El número de expresiones de la lista, debe ser igual al número de parámetros formales.

Tipos de variables

Según el lugar donde son declaradas puede haber dos tipos de variables.

Globales: son variables que permanecen activas durante todo el programa. Se crean al iniciarse éste y se destruyen de la memoria al finalizar. Pueden ser utilizadas en cualquier función. Se declaran fuera de cualquier función (antes del main).

Locales: las variables son creadas cuando el programa llega a la función en la que están definidas. Al finalizar la función desaparecen de la memoria.

Si dos variables, una global y una local, tienen el mismo nombre, la local prevalecerá sobre la global dentro de la función en que ha sido declarada.

Dos variables locales pueden tener el mismo nombre siempre que estén declaradas en funciones diferentes.

Declaración de una función. Función prototipo.

Al igual que las variables, las funciones también han de ser declaradas.
La declaración de una función, denominada también **función prototipo**, permite conocer el nombre, el tipo del resultado, los tipos de los parámetros formales y opcionalmente sus nombres. No define el cuerpo de la función.

Una función prototipo tiene la misma sintaxis que la definición de una función, excepto que ésta, termina con un punto y coma.

Los prototipos de las funciones pueden escribirse antes de la función **main** o bien en otro fichero. En este último caso se lo indicaremos al compilador mediante la directiva #include.

Ejemplo:

```
#include <stdio.h>
#include <conio.h>
#define pi 3.14
void espera(void); //funciones prototipo
double calcular(double, int);
int parte_entera(float);
void main( )
{
   float p=3.2,r;
   int b=5;

   r = calcular (p,b);
   printf("El resultado de calcular %f+%d*pi es %.2f\n",p,b,r);
   printf("\n Lo mismo de otra forma %f+%d*pi es %.2f\n",p,b,calcular(p,b));

   b=parte_entera(p);
   printf("\n La parte entera de %f es %d",p,b);

   printf("\n Y la parte entera de %f es %d",8.7,parte_entera(8.7));

  printf("\n Introduce un número real: ");   scanf("%f",&p);
  printf("\n La parte entera de %f es %d",p,parte_entera(p));

  espera();
}

void espera(void)  //no recibe ni retorna ningún valor
{
 printf("\n\nPulsa cualquier tecla para finalizar...");
 getch();
}

double calcular (double a, int b) {   return(a+b*pi); }

int parte_entera(float p) {
   int auxiliar;
   auxiliar = (int)p;
   return(auxiliar);
}
```

Pasando parámetros por valor o por referencia

Cuando se efectúa la llamada a una función, se le pueden pasar los parámetros por valor o por referencia.

Pasar parámetros por valor, significa copiar los parámetros actuales en sus correspondientes parámetros formales, operación que se hace automáticamente, con lo cual no se modifican los parámetros actuales.

Pasar parámetros por referencia, significa que lo transferido no son los valores sino las direcciones de las variables que contienen esos valores, con lo cual los parámetros actuales pueden verse modificados.

Cuando se llama a una función, los argumentos especificados en la llamada son pasados por valor, excepto los arrays que se pasan por referencia (ya que el nombre del array indica la dirección de comienzo del mismo). Así, para pasar la dirección de una variable se utiliza el operador **&** antes del nombre de la variable, sin embargo para pasar la dirección de un array no es necesario utilizar ningún operador.

```
#include <stdio.h>
int SUMA1 (int , int);  // Todos los parámetros se pasarán por valor
void SUMA2(int, int, int *);  //Los dos primeros se pasan por valor y el último por referencia
void main()
{
   int num1, num2, suma;

  printf("\n Suma dos números enteros de distintas formas: ");
  printf("\n Introduce el primer numero: ");
  scanf("%d",&num1);
  printf("\n Introduce el segundo numero: ");
  scanf("%d",&num2);

  suma = SUMA1(num1, num2);
  printf("\n La suma de %d con %d es %d ", num1, num2, suma);
  printf("\n La suma de %d con %d es %d ", num1, num2, SUMA1(num1, num2));

  SUMA2(num1, num2,&suma);
  printf("\n La suma de %d con %d es %d ", num1, num2, suma);
}

int SUMA1(int a, int b)
{
   return (a+b);
}

void SUMA2(int a, int b, int  *s)
{
   *s = a + b;    // Al cambiar el valor de la dirección de memoria, modificará el valor de la variable suma
    a = b = 900;  // Aunque cambie el valor de a y b, no modifica el valor de las variables num1 ni num2
}
```

7. Tipos estructurados de datos

7.1 Arrays

Un array es una estructura homogénea, compuesta por varias componentes, todas del mismo tipo, y almacenadas consecutivamente en memoria. Cada componente puede ser accedido directamente por el nombre de la variable array seguido de un subíndice encerrado entre corchetes.

Un **vector** es un array *unidimensional*, es decir, sólo utiliza un índice para referenciar a cada uno de los elementos. Su sintaxis es la siguiente:

tipo nombre [tamaño];

tipo Indica el tipo de los elementos del array. Puede ser cualquier tipo excepto **void**
nombre Identificador que nombra el array
tamaño Constante que especifica el número de elementos del array. El tamaño puede omitirse cuando se inicializa el array, cuando se declara como un parámetro formal en una función o cuando se hace referencia a un array declarado en otra parte del programa.

Ejemplo: int datos[3];
Define un array unidimensional *datos* que contiene tres elementos. Cada uno de estos elementos se identifica de la siguiente forma: datos[0] datos[1] datos[2]
Los subíndices son enteros consecutivos, y el primero siempre vale 0.

Un subíndice puede ser cualquier expresión entera. Así, los siguientes códigos son equivalentes:

for (i=0;i<3;i++) datos[i]=0; // le asigna el valor 0	datos[0]=0; datos[1]=0; datos[2]=0;

Podemos *inicializar* (asignarle valores) un vector en el momento de declararlo. En este caso no es necesario indicar el tamaño. Su sintaxis es la siguiente: **tipo nombre []={ valor 1, valor 2...}**

Cadenas de caracteres.

Una particularidad son los vectores de tipo **char** (cadena de caracteres) en los que deberemos indicar el elemento en el cual se encuentra el fin de la cadena mediante el carácter nulo (**\0**). Así en un vector de n elementos de tipo char podremos rellenar un máximo de n-1, es decir, hasta vector[n-2].

Un array de caracteres puede ser inicializado asignándole un literal. Así, char cad[]="abcd"; Inicializa el array de caracteres cadena con cinco elementos (de cad[0] a cad[4]). El quinto elemento (cad[4]), es el carácter nulo (\0) con el cual C finaliza todas las cadenas de caracteres.

Si se especifica el tamaño del array de caracteres y la cadena asignada es más larga que el tamaño especificado, los caracteres en exceso se ignoran. Por ejemplo, char cadena[3]="abcd"; donde sólo los tres primeros caracteres son asignados a cadena[0], cadena[1] y cadena[2] respectivamente. Los caracteres d y \0 son ignorados.

Si la cadena asignada es más corta que el tamaño del array de caracteres, el resto de los elementos del array son inicializados con el valor nulo (\0).

Funciones específicas de manejo de cadenas: gets y puts.

gets lee una cadena de caracteres introducida desde el teclado y la almacena en la variable especificada. Finaliza la entrada introduciendo ENTER. Añade un nulo al final (\0) en lugar del retorno de carro. No hay límite para el número de caracteres que **gets** leerá, por tanto hay que asegurarse de que no se produzca desbordamiento.
A diferencia de scanf, permite la entrada de una cadena de caracteres formada por varias palabras separadas por espacios en blanco sin ningún tipo de formato.

puts escribe una cadena por pantalla y al final añade un carácter de nueva línea. El terminador nulo (\0) pasa a ser un retorno de carro.

```
char nombre[20];

printf("Introduce tu nombre: ");
gets(nombre);  //equivale a scanf("%[^\n]s",nombre);

puts("Te llamas");   //equivale a printf("Te llamas\n");
puts(nombre);        //equivale a printf("%s\n",nombre);
```

Si tenemos: **char saludo[5];** **int bonoloto[6];** **float coordenadas[3];**

<table>
<tr><td colspan="3">Se pueden inicializar a valores concretos:</td></tr>
<tr><td>saludo[0]='h';
saludo[1]='o';
saludo[2]='l';
saludo[3]='a';
saludo[4]='\0';

strcpy(saludo,"hola");</td><td>loto[0]=2;
loto[1]=5;
....
loto[5]=45;</td><td>
coordenadas[0]=2.3;
coordenadas[1]=5.67;
coordenadas[2]=12.7;</td></tr>
<tr><td colspan="3">Se pueden inicializar a un valor cuando se declaran, indicándole (o no) su tamaño</td></tr>
<tr><td>char saludo[5]="hola";
char saludo[5]={'h','o','l','a','\0'};
char saludo[ ]= "hola";
char saludo[ ]={'h','o','l','a','\0'};</td><td>int loto[6]={2,5,9,20,23,45};
int loto[ ]={2,5,9,20,23,45};</td><td>float coordenadas[3]={2.3, 5.67, 12.7};
float coordenadas[]={2.3, 5.67, 12.7};</td></tr>
<tr><td colspan="3">Solicitar un valor concreto</td></tr>
<tr><td>scanf("%c",&saludo[0]);</td><td>scanf(" %d",&loto[3]);</td><td>scanf("%f",&coordenadas[1]);</td></tr>
<tr><td colspan="3">Solicitar TODOS sus valores</td></tr>
<tr><td>scanf(" %s",saludo);
gets(saludo);</td><td>for (i=0;i<6;i++)
{
printf(" numero %d",i+1);
scanf("%d",&vector[i]);
}</td><td>for (i=0;i<3;i++)
{
printf(" coordenada %d",i+1);
scanf("%d",&coordenada[i]);
}</td></tr>
<tr><td colspan="3">Mostrar un valor concreto</td></tr>
<tr><td>printf(" %c",saludo[2]);</td><td>printf("último:%d",vector[5]);</td><td>printf("primero:%f",coordenada[0]);</td></tr>
<tr><td colspan="3">Mostrar TODOS los valores</td></tr>
<tr><td>printf(" %s",saludo);
puts(saludo);</td><td>for (i=0;i<6;i++)
printf(" %d",vector[i]);</td><td>for (i=0;i<3;i++)
printf(" %f",coordenada[i]);</td></tr>
</table>

7.2 Matrices

Una matriz es un array multidimensional. Se definen igual que los vectores excepto que se requiere un índice por cada dimensión. Su sintaxis es la siguiente:

tipo nombre [tamaño 1][tamaño 2]...[tamaño n];

Ejemplo: int a[2][3][2];
Array de 3 dimensiones, con 2*3*2=12 elementos de tipo entero. El primer elemento es el a[0][0][0] u el último es el a[1][2][1].

También se podría haber inicializado: int a[2][3][2]={1,2,3,4,5,6,7,8,9,10,11,12};

El orden en el que los valores son asignados a los elementos de la matriz sería el siguiente:
a[0][0][0]=1 a[0][0][1]=2 a[0][1][0]=3 a[0][1][1]=4 a[0][2][0]=5 a[0][2][1]=6
a[1][0][0]=7 a[1][0][1]=8 a[1][1][0]=3 a[1][1][1]=10 a[1][2][0]=11 a[1][2][1]=12
Al igual que en un array unidimensional (vector), en un array mutidimensional, se puede omitir un tamaño cuando se incializa el array: int a[][3][2]={1,2,3,4,5,6,7,8,9,10,11,12};

También se pueden inicializar cadenas de caracteres:

char dias[7][10]={"lunes","martes","miércoles","jueves","viernes","sábado","domingo"};

Para referirnos a cada palabra bastaría con el primer índice: **printf("%s",dias[i]);**

Veremos un ejemplo de cómo se rellena y visualiza una matriz *bidimensional*. Supongamos que se quiere almacenar los números de la loto de un año (52 semanas).
Se necesitan dos bucles para cada una de las operaciones. Un bucle controla las filas y otro las columnas.

```
#define NUM_SEMANAS 52
#define NUMEROS 6
#include <stdio.h>
main()
{
  int f,c,loto[NUM_SEMANAS][NUMEROS];
  for (f=0;f<NUM_SEMANAS;f++)    /* rellenamos la matriz */
  {
     printf("\n\nSemana %d",f+1);
     for (c=0;c<NUMEROS;c++)
     {
       printf("\nNumero %d: ",c+1);
       scanf("%d",&loto[f][c]);
      }
  }

  for (f=0;f<NUM_SEMANAS;f++)   /* mostramos los números */
  {
     printf("\n\nSemana %d",f+1);
     for (c=0;c<NUMEROS;c++)
        printf("\nNumero %d: %d",c+1,loto[f][c]);
   }
}
```

7.3 Estructuras

Una estructura es un conjunto de una o más variables de diferentes tipos, lógicamente relacionadas. A una estructura también se le da el nombre de registro.

Crear una estructura es definir un nuevo tipo de datos, denominado **tipo_estructura** y declarar una variable de este tipo. En la definición del tipo estructura se especifican los elementos que la componen así como sus tipos. Cada elemento de la estructura recibe el nombre de campo del registro. La sintaxis es la siguiente:

```
struct tipo_estructura
{
  tipo nombre_variable1;
  tipo nombre_variable2;
          .......
};
```

tipo_estructura Identificador que nombra el nuevo tipo de dato que hemos creado.
tipo y **nombre_variable** son las variables que formarán parte de la estructura.

Después de definir un tipo estructura, podemos declarar una variable de este tipo de la forma:

struct tipo_estructura variable1 [, variable 2, [variable3] ...];

Formas de realizar la declaración de variables	

struct ficha
//definición del tipo estructura ficha
{
  char nombre[20];
  char direccion[40];
  int edad;
};
struct ficha cliente, proveedor;
	struct
	{
	  char nombre[20];
	  char direccion[40];
	  int edad;
	 }cliente, proveedor;

En este caso declaramos la estructura, y en el momento en que necesitemos las variables, las declaramos. Para poder declarar una variable de tipo estructura en cualquier sitio, la estructura tiene que estar declarada previamente antes de la función **main**.	Las declaramos al mismo tiempo que la estructura. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa.

La forma de acceder a los campos de la estructura es la siguiente: **variable.campo;**

Ejemplo: **cliente.edad=35;**

```
strcpy(cliente.nombre,"Pipas churruca");
gets(cliente.direccion);
```

El contenido de las estructuras se puede pasar de una a otra, siempre que sean del mismo tipo:

proveedor=cliente;

También es posible inicializar variables de tipo **estructura** en el momento de su declaración:

```
struct ficha cliente={"Pipas churruca","C/San José 32", 35};
```

Si uno de los campos de la estructura es un array de números, los valores de la inicialización deberán ir entre llaves:

```
struct notas   {
            char nombre[30];
            int notas[5];
 };

 struct notas alumno={"Carlos Pérez",{8,7,9,6,10}};
```

Las estructuras pueden contener campos que a su vez son estructuras.

```
struct fecha{
 int dia, mes, anyo;
};

struct ficha
{
  char nombre[20];
  char direccion[40];
  int edad;
  struct fecha fecha_nac;
};
struct ficha persona;
```

```
persona.fecha_nac.dia=12;
printf("Mes : ");
scanf("%d",&persona.fecha_nac.mes);
```

Paso de campos de estructuras a funciones

Un campo de una estructura se puede pasar como argumento a una función de dos formas diferentes:

a) Pasando el valor del campo

Funcion(**persona.edad**);

Siendo esta función declarada como void Funcion(int n){ ... }

b) Pasando su dirección

Funcion(&**persona.edad**);

Siendo esta función declarada como void Funcion(int *n) { *n=45; ... }

Paso de estructuras a funciones

También se puede pasar toda la estructura como argumento,

a) Pasando el valor de la estructura

Funcion(**persona**);

Estando la función declarada como

void Funcion(**struct ficha** una_persona){ una_persona.edad=45; }

b) Pasando la estructura por dirección

Funcion(&**persona**);

Estando declarada como void Funcion(**struct ficha** *una_persona){ una_persona**->**edad=45; }

Arrays de estructuras: cuando los elementos de un array son de tipo estructura.

```
struct fecha{
 int dia, mes, anyo;
};

struct ficha
{
  char nombre[20];
  char direccion[40];
  int edad;
  struct fecha fecha_nac;
};
struct ficha Personal[20];
//Tenemos 20 empleados
```

```
strcpy(Personal[0].nombre, "Pepe Pérez");
printf("dirección : ");
   gets(personal[0].direccion);
Personal[0].fecha_nac.dia=12;
printf("Mes : ");
scanf("%d",&personal[0].fecha_nac.mes);
```

Paso de arrays de estructuras a funciones

Se puede pasar un array de estructuras como argumento. En este caso se pasan siempre por dirección, ya que el nombre del array es la dirección de memoria del mismo.

```
Funcion(personal);
```

Estando la función declarada como

```
void Funcion(struct ficha varias_personas[])
{
   int i;

   for (i=0; i<20; i++)
      varias_personas[i].edad=45;
}
```

7.4 Uniones

Variables que pueden contener, objetos de diferentes tipos, denominados campos, en una misma zona de memoria.

La declaración tiene la misma forma que la declaración de una estructura, excepto que en lugar de la palabra reservada **struct** se pone la palabra reservada **union**. Todo lo expuesto para las estructuras es aplicable a las uniones, excepto la forma de almacenamiento de sus campos.

Ejemplo:

```
union ficha //o guardo el nombre de la empresa, o su  CIF o bien su código
{
  char nombre_empresa[20], cif[20];
  int codigo;
};
```

Para almacenar los campos de una unión, se requiere una zona de memoria igual a la que ocupa el campo más largo de la unión. Todos los campos son almacenados en el mismo espacio de memoria y comienzan en la misma dirección. El valor almacenado es sobrescrito cada vez que se asigna un valor al mismo campo o a un campo diferente.

7.5 Enumeraciones

Lista de valores que pueden ser tomados por una variable de ese tipo. Crear una enumeración es definir un nuevo tipo de dato y declarar una variable de este tipo. La sintaxis es la siguiente:

```
enum tipo_enumerado
{
        definición de nombres de constantes
};
```

tipo_enumerado Identificador que nombra el nuevo tipo de dato que hemos creado.

Después de definir un tipo enumerado, podemos declarar una o más variables de este tipo de la forma: **enum tipo_enumerado variable1 [, variable 2, [variable3] ...];**

Ejemplo:
```
enum colores{ blanco, amarillo, azul, rojo, verde, negro};
enum colores color;
```

Se declara una variable **color** del tipo enumerado **colores**, la cual puede tomar cualquier valor de los especificados en la lista, así: color=azul;

Las variables de tipo enum son tratadas como si fuesen de tipo int. Cada identificador de la lista de constantes tiene asignado un valor. Por defecto, el primer identificador tiene asignado el valor **0**, el siguiente el valor **1**, y así sucesivamente.

color = negro; es equivalente a color = 6;

A cualquier identificador de la lista, se le puede asignar un valor inicial por medio de una expresión constante. Los identificadores sucesivos tomarán valores correlativos a partir de éste.

```
enum colores{
  blanco, amarillo, azul, rojo=0, verde, marron, negro
}color;
```

La variable color de tipo colores puede tener asociado el valor blanco = 0, amarillo =1, azul=2, rojo=0, verde=1, marron=2 y negro = 3.

A los campos de una enumeración se les aplica las siguientes reglas:

- Dos o más campos pueden tener un mismo valor (blanco tiene el mismo valor que rojo, amarillo que verde, ...)
- Un identificador no puede aparecer en más de un tipo.
- No es posible leer o escribir directamente un valor de un tipo enumerado.

7.6 Tipos definidos por el usuario

typedef permite declarar nuevos nombres de tipos de datos (sinónimos). Su sintaxis es:

typedef tipo identificador [,identificador]...;

tipo es cualquier tipo definido en C
identificador es el nuevo nombre dado a **tipo**

Ejemplos:

```
typedef int entero;     /* acabamos de crear un tipo de dato llamado entero */
entero a, b=3;          /* declaramos dos variables de este tipo */
```

Su empleo con estructuras está especialmente indicado. Se puede hacer de varias formas:

```
struct ficha
{
        char nombre[20];
        char apellidos[40];
        int edad;

};

typedef struct ficha datos;

datos fijo,temporal;
```

```
typedef struct
{
        char nombre[20];
        char apellidos[40];
        int edad;
}datos;

datos fijo,temporal;
```

FUNCIONES DE BIBLIOTECA

función	Propósito	archivo include
int isalnum(int c)	Determina si el argumento es alfanumérico (A ... Z o a...z o 0...9) Devuelve un cero (falso)si el carácter no es alfanumérico y distinto de cero (cierto) si es una letra del alfabeto.	ctype.h
int isalpha(int c)	Determina si el argumento es alfabético (A ... Z o a...z)	ctype.h
int isascii(int c)	Determina si el argumento es un carácter ASCII (0 a 127)	ctype.h
int iscntrl(int c)	Determina si el argumento es un carácter de control (0-31 y 127)	ctype.h
int isdigit(int c)	Determina si el argumento es un dígito decimal (0..9).	ctype.h
int islower(int c)	Determina si el argumento es una letra minúscula (a..z).	ctype.h
int isodigit(int c)	Determina si el argumento es un dígito octal (0..7).	ctype.h
int isprint(int c)	Determina si el argumento es un carácter ASCII imprimible(32-126)	ctype.h
int ispunct(int c)	Determina si el argumento es un carácter de puntuación.	ctype.h
int isspace(int c)	Determina si el argumento es un espacio en blanco, tabulador o carácter de nueva línea. (9-13 o 32)	ctype.h
int isupper(int c)	Determina si el argumento es una letra mayúscula (A..Z)	ctype.h
int isxdigit(int c)	Determina si el argumento es un dígito hexadecimal (0..9 y A..F).	ctype.h
int tolower(int c)	Convierte c a minúsculas si procede. Sino devuelve c sin cambios.	ctype.h
int toupper(int c)	Convierte c a mayúsculas si puede. Sino devuelve c sin cambios.	ctype.h
char * strcat(char *s1, char *s2)	Concatena s2 a s1 y añade un nulo al final. El carácter nulo que tenía s1 se sobreescribe con el primer carácter de s2.La cadena s2 no se modifica durante la operación. No realiza comprobación alguna sobre el tamaño de las cadenas, por lo que es necesario asegurarse que s1 es lo suficientemente grande para guardar su contenido original y el de s2. La función devuelve un puntero a s1.	string.h
char * strchr(char * s, int c)	Devuelve un puntero a la primera ocurrencia de c en s o un valor NULL si el carácter no es encontrado.	string.h
int strcmp(char *s1, char *s2)	Compara dos cadenas de caracteres y devuelve un entero que se interpreta como sigue: VALOR — SIGNIFICADO menor que 0 — s1 es menor que s2 0 — s1 es igual que s2 mayor que 0 — s1 es mayor que s2	string.h
char * strcpy(char *s1, char *s2)	Copia el contenido de s2 en s1. Devuelve un puntero a s1.	string.h
unsigned strcspn(char *s1, char *s2)	Devuelve la posición (subíndice) del primer carácter de s1 que pertenece al conjunto de caracteres contenidos en s2.	string.h
unsigned strlen(char *s)	Devuelve la longitud de la cadena s. La cadena debe finalizar con el carácter nulo. La función no cuenta el nulo.	string.h
char * strlwr(char * s)	Pasa a minúsculas los caracteres de la cadena s. Devuelve un puntero a la cadena s.	string.h
char * strrev(char *s)	Invierte la cadena apuntada por s. El nulo final se mantiene en la misma posición. Devuelve un puntero a la cadena s.	string.h

int abs (int x)	Retorna el valor absoluto de x.	stdlib.h
double atof(char *s)	Convierte una cadena de caracteres a un valor en doble precisión.	stdlib.h
int atoi(char *s)	Convierte una cadena de caracteres a un valor entero.	stdlib.h
long atol(char *s)	Convierte una cadena de caracteres a un valor entero largo	stdlib.h
void exit(unsigned int n)	Cierra todos los archivos y buffers y termina el programa. (El valor de n se asigna en la función para indicar el estado de terminación).	stdlib.h
int rand(void)	Retorna un valor aleatorio positivo.	stdlib.h
void srand(unsigned x)	Inicializa el generador de números aleatorios.	stdlib.h
int system(char *s)	Pasa la orden al Sistema Operativo. Retorna 0 si la orden se ejecuta correctamente; en otro caso, retorna un valor distinto de 0.	stdlib.h
double acos(double x)	Da como resultado el arco, en el rango 0 a Π, cuyo coseno es x. El valor de x debe estar entre –1 y 1, sino se produce un error de dominio.	math.h
double asin(double x)	Da como resultado el arco, en el rango -Π/2 a Π/2, cuyo seno es x. El valor de x debe estar entre –1 y 1, sino se produce un error de dominio.	math.h
double atan(double x)	Da como resultado el arco, en el rango -Π/2 a Π/2, cuyo tangente es x. El valor de x debe estar entre –1 y 1, sino se produce un error de dominio.	math.h
double ceil(double x)	Da como resultado un valor double que representa el entero más pequeño que es mayor o igual que x.	math.h
double cos(double arg)	Devuelve el coseno de arg. El valor de arg debe estar en radianes. El valor devuelto está entre –1 y 1.	math.h
double exp (double x)	Da como resultado el valor de e^x (e=2.7182818...)	math.h
double fabs (double x)	Devuelve el valor absoluto de x.	math.h
double floor(double x)	Da como resultado un valor double que representa el entero más grande que es menor o igual que x.	math.h
double fmod (double x, double y)	Devuelve el resto de x/y	math.h
double log(double x)	Devuelve el logaritmo natural de x. Se produce un error de dominio si x es negativo, y un error de rango si el argumento es cero.	math.h
double log10(double x)	Devuelve el logaritmo (en base 10) de x. Se produce un error de dominio si x es negativo, y un error de rango si el argumento es cero.	math.h
double pow (double base, double exp)	Devuelve la base elevada a la potencia exp. Se produce un error de dominio si base es 0 y exp es menor o igual que cero, o si la base es negativa y el exp no es un entero. Si hay desbordamiento se produce un error de rango.	math.h
double sin(double arg)	Devuelve el seno de arg. El valor de arg debe estar en radianes.	math.h
double sqrt(double x)	Devuelve la raíz cuadrada de x. Si el argumento es negativo se produce un error de dominio.	math.h
double tan(double arg)	Devuelve la tangente de arg. El valor de arg debe estar en radianes.	math.h

Sistemas de representación de números

Decimal	Binario	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Hay 8 dígitos octales (en el rango 0 a 7) y 16 hexadecimales (en el rango de 0 a 9 y A a F).

Cada dígito octal es equivalente a tres dígitos binarios (3 bits), y cada dígito hexadecimal es equivalente a cuatro dígitos binarios (4 bits). Por tanto, los números octales y hexadecimales proporcionan una forma conveniente y concisa para representar valores binarios.

Así, el valor binario 10110111 se representa en octal como 267 (010 110 111). Para que sea múltiplo de 3 se añade un cero a la izquierda.

Así, el valor binario 10110111 se representa en hexadecimal como B7 (1011 0111).

Tabla ASCII (American Standard Code for Information Interchange)

El tipo **char** se utiliza para almacenar un carácter, pero desde el punto de vista técnico se trata de un número entero y como tal puede guardar números, aunque realmente el ordenador almacena siempre 0 y 1.

Para manejar caracteres, el ordenador emplea un código numérico en el que se asocian unos números enteros a ciertos caracteres. El más común es el ASCII o código estándar americano para el intercambio de información.

char letra;
letra='A'= 65 = 0101 = 0x41;

Cada celda de la tabla ASCII representará:

Número en base 10	nº en binario
Carácter	

	0		1		2		3		4		5		6		7	
0	00	0000 0000	16	0001 0000	32	0010 0000	48	0011 0000	64	0100 0000	80	0101 0000	96	0110 0000	112	0111 0000
	NUL		**DLE**		**SP**		**0**		**@**		**P**		**`**		**p**	
1	01	0000 0001	17	0001 0001	33	0010 0001	49	0011 0001	65	0100 0001	81	0101 0001	97	0110 0001	113	0111 0001
	SOH		**DC1**		**!**		**1**		**A**		**Q**		**a**		**q**	
2	02	0000 0010	18	0001 0010	34	0010 0010	50	0011 0010	66	0100 0010	82	0101 0010	98	0110 0010	114	0111 0010
	STX		**DC2**		**“**		**2**		**B**		**R**		**b**		**r**	
3	03	0000 0011	19	0001 0011	35	0010 0011	51	0011 0011	67	0100 0011	83	0101 0011	99	0110 0011	115	0111 0011
	ETX		**DC3**		**#**		**3**		**C**		**S**		**c**		**s**	
4	04	0000 0100	20	0001 0100	36	0010 0100	52	0011 0100	68	0100 0100	84	0101 0100	100	0110 0100	116	0111 0100
	EOT		**DC4**		**$**		**4**		**D**		**T**		**d**		**t**	
5	05	0000 0101	21	0001 0101	37	0010 0101	53	0011 0101	69	0100 0101	85	0101 0101	101	0110 0101	117	0111 0101
	ENQ		**NAK**		**%**		**5**		**E**		**U**		**e**		**u**	
6	06	0000 0110	22	0001 0110	38	0010 0110	54	0011 0110	70	0100 0110	86	0101 0110	102	0110 0110	118	0111 0110
	ACK		**SYN**		**&**		**6**		**F**		**V**		**f**		**v**	
7	07	0000 0111	23	0001 0111	39	0010 0111	55	0011 0111	71	0100 0111	87	0101 0111	103	0110 0111	119	0111 0111
	BEL		**ETB**		**'**		**7**		**G**		**W**		**g**		**w**	
8	08	0000 1000	24	0001 1000	40	0010 1000	56	0011 1000	72	0100 1000	88	0101 1000	104	0110 1000	120	0111 1000
	BS		**CAN**		**(**		**8**		**H**		**X**		**h**		**x**	
9	09	0000 1001	25	0001 1001	41	0010 1001	57	0011 1001	73	0100 1000	89	0101 1001	105	0110 1001	121	0111 1001
	HT		**EM**		**)**		**9**		**I**		**Y**		**i**		**y**	
A	10	0000 1010	26	0001 1010	42	0010 1010	58	0011 1010	74	0100 1010	90	0101 1010	106	0110 1010	122	0111 1010
	LF		**SUB**		*****		**:**		**J**		**Z**		**j**		**z**	
B	11	0000 1011	27	0001 1011	43	0010 1011	59	0011 1011	75	0100 1011	91	0101 1011	107	0110 1011	123	0111 1011
	VT		**ESC**		**+**		**;**		**K**		**[**		**k**		**{**	
C	12	0000 1100	28	0001 1100	44	0010 1100	60	0011 1100	76	0100 1100	92	0101 1100	108	0110 1100	124	0111 1100
	FF		**FS**		**,**		**<**		**L**		****		**l**		**|**	
D	13	0000 1101	29	0001 1101	45	0010 1101	61	0011 1101	77	0100 1101	93	0101 1101	109	0110 1101	125	0111 1101
	CR		**GS**		**-**		**=**		**M**		**]**		**m**		**}**	
E	14	0000 1110	30	0001 1110	46	0010 1110	62	0011 1110	78	0100 1110	94	0101 1110	110	0110 1110	126	0111 1110
	SO		**RS**		**.**		**>**		**N**		**^**		**n**		**~**	
F	15	0000 1111	31	0001 1111	47	0010 1111	63	0011 1111	79	0100 1111	95	0101 1111	111	0110 1111	127	0111 1111
	SI		**US**		**/**		**?**		**O**		**_**		**o**		**DEL**	

NOTA: Los 32 primeros caracteres y el último son caracteres de control; no se pueden imprimir.