Técnicas de programación y lenguaje C.

Rafael Jurado Moreno
(rafa.eqtt@gmail.com)

I.E.S. María Moliner.
Segovia 2009

1.Conceptos generales.

1.1.Introducción

Un ordenador es una máquina de procesamiento de información. Dispone de rapidez, precisión y memoria para ejecutar programas elaborados por el programador, pero carece de inteligencia. (El tonto rápido).

Un programa es un conjunto de instrucciones que el ordenador debe ejecutar para realizar la tarea prevista por el programador.

1.2.Solución de problemas.

En el proceso de resolución de un problema de programación se distinguen las siguientes fases:

- **Análisis** detallado del problema
- **Descomposición** de la solución en tareas elementales o diseño del algoritmo.
- **Codificación** del algoritmo.
- **Obtención** del programa **ejecutable**.
- **Prueba**, verificación y depuración.
- **Documentación**.

1.2.1.Análisis del problema.

Al analizar el problema deben considerarse los siguientes factores:

- El equipo que se va a utilizar.
- Los datos de entrada.
- El tratamiento que debe realizarse con los datos.
- Los resultados de salida.
- La posibilidad de descomponer el problema en módulos más pequeños.

1.2.2.Diseño del algoritmo.

Es una **formula** para resolver un problema. Conjunto de acciones o secuencia de operaciones que ejecutadas en un determinado orden resuelven el problema. Existen muchas formas de resolver un problema, hay que coger la más efectiva. Características:

- Tiene que ser preciso.
- Tiene que estar bien definido.
- Tiene que ser finito.

El algoritmo se puede expresar de forma **gráfica** o mediante **pseudocódigo** (Parecido a los lenguajes de programación de alto nivel).

1.2.3.Codificación del algoritmo.

Se trata de la traducción del algoritmo al lenguaje de programación elegido.

1.2.4.Obtención del programa.

Se realiza en tres fases:

- Edición. En esta fase se crea el código fuente. Se trata de un fichero de texto escrito en el lenguaje de programación elegido.
- Compilación. Sirve para obtener el programa en lenguaje máquina (Instrucciones que la máquina entiende). El resultado son los ficheros de código objeto o código máquina (**depende del Hardware**). La compilación se realiza con programas compiladores o intérpretes, se-

gún el lenguaje elegido
- **Montaje o linkado. El** montaje o linkado (enlazado) es un proceso de enlace entre las distintas partes que componen el programa y la adecuación del código para obtener un programa ejecutable para el sistema operativo en el que estamos trabajando. (**Depende del sistema operativo**).

1.2.5.Depuración del programa.

Es la fase en la cual debe estudiarse el comportamiento del programa en todas las situaciones límite que puedan plantearse, retocando el código y el algoritmo para conseguir un resultado óptimo.

1.2.6.Documentación.

Un programa debe tener dos niveles de documentación:

- **Documentación Interna**: Comentarios del código fuente incluidos en el mismo fichero, que clarifiquen el funcionamiento del programa para que pueda ser comprendido por cualquier programador.
- **Documentación externa**: No forma parte del programa y se suele suministrar en papel. Debe incluir la descripción del problema, de los datos de entrada y salida, del algoritmo, del programa completo y los manuales de usuario y de mantenimiento.

1.3.Lenguajes de programación.

1.3.1.Lenguajes interpretados.

Se ejecuta el código fuente directamente por medio de un intérprete, que es un programa que realiza el siguiente proceso:

- Carga el código fuente (fichero de texto).
- Lee el código de una línea.
- Convierte al código en código máquina.
- Envía las instrucciones al procesador.
- Repite desde el punto 2 hasta el final del programa.

En este caso el código **fuente es independiente del sistema operativo**.

Los lenguajes interpretados más utilizados actualmente son los diversos script. (Java, c, perl, delfi).

1.3.2.Lenguajes compilados.

Son aquellos que permiten obtener un programa **ejecutable y autónomo** a partir del código fuente, que puede ser ejecutado en cualquier máquina bajo el sistema operativo para el que se compiló.

El proceso anterior se realiza por medio de un programa que se llama compilador y que realiza las siguientes tareas:

- Carga el código fuente (fichero de texto).
- Traduce el código fuente a código máquina /código objeto).
- Enlaza (Linkado) el código objeto con las librerías y funciones necesarias y crea el ejecutable.

Este proceso se realiza una única vez, quedando al final un ejecutable autónomo que se puede lanzar directamente desde el sistema operativo.

1.4.Metodología de la programación.

La calidad de un programa está determinada por las siguientes características:

- **Legibilidad**. Facilidad para ser entendido por cualquier programador.
- **Portabilidad**. Capacidad para compilarse o interpretarse en distintas máquinas, e incluso para ser traducido a otros lenguajes.
- **Fiabilidad**. Facilidad de recuperación del control tras errores de ejecución o uso inadecuado.
- **Eficiencia**. Grado de aprovechamiento de los recursos.

La metodología de programación describe las técnicas y métodos para elaborar programas que reúnan los máximos requisitos de calidad. Para ello es necesario que el programa pueda mantenerse, modificarse o actualizarse fácilmente por distintos programadores. Para conseguirlo se han definido dos criterios de programación: La **programación modular** y la **programación estructurada**. Los dos criterios son complementarios y **no excluyentes**.

1.4.1.Programación modular.

Consiste en descomponer el programa en **partes claramente diferenciadas**, llamadas módulos, que pueden ser tratados de forma independiente. Son bloques de instrucciones con nombres diferenciados que pueden ser llamados varias veces desde el módulo principal.

Los módulos pueden estar compuestos, a su vez, por otros módulos o realizar llamadas a otros módulos.

Los criterios para la división de un programa en módulos dependen del programador, aunque es importante que cumplan los siguientes requisitos:

- Tendrán un único punto de entrada y otro de salida.
- El módulo principal será reducido, para que se aprecie claramente el algoritmo.
- Deben ser independientes entre sí.
- Deben resolver completamente partes diferenciadas y definidas del problema.

1.4.2.Programación estructurada.

Es la técnica que permite realizar cualquier programa que disponga de un único punto de entrada y uno solo de salida mediante el empleo de tres tipos de **estructuras de control**:Secuenciales, condicionales y repetitivas.

- Estructura **secuencial**: Se agrupan las operaciones o sentencias de forma consecutiva
- Estructura **condicional**: Se permite la selección entre dos o más grupos de operaciones dependiendo del cumplimiento de una condición.
- Estructura **repetitiva**: Permite ejecutar repetidamente una o varias sentencias un número determinado de veces dependiendo de una condición.

1.5.Elementos en los programas.

Cuando se escribe el programa se utilizan unos elementos cuyas características están determinadas por el lenguaje utilizado. Entre estos elementos cabe destacar los identificadores, las constantes, las variables, los operadores, las expresiones y las sentencias.

1.5.1.Identificadores.

Los identificadores o **etiquetas** son palabras escogidas por el programador para designar los elementos de un programa susceptibles de ser nombrados (como: variables, constantes, subprogramas, etc.). Están formados por caracteres alfabéticos y dígitos que empiezan por un carácter alfabético. Los identificadores deben que ser significativos.

1.5.2.Datos.

Constituyen la información que será procesada por el programa. Los datos pueden ser **simples** o **estructurados**.

- **Simples**: Aquellos que no pueden descomponerse en otros más sencillos.
- **Carácter**. Es la unidad de información más pequeña, pueden ser alfabéticos (A...Z, a...z), numéricos (0...9) y especiales (=, *, +, -, etc.).
- **Numéricos**: Representan cantidades y pueden ser de tipo entero o real.
- **Lógicos** o **booleanos**: Solo pueden tomar dos valores (1 ó 0, Cierto ó falso).
- **Cadenas** de caracteres: Conjunto de caracteres alfabéticos, numéricos y especiales.
- **Estructurados**: Son conjuntos de datos simples agrupados como una única entidad con un único identificador.

1.5.3.Constantes.

Tienen un valor fijo que se le da cuando se define la constante y que ya no puede ser modificado durante la ejecución.

1.5.4.Variables.

El valor puede cambiar durante la ejecución del algoritmo, pero nunca varia su nombre y su tipo. Las características de cada variable son: **tipo de dato**(entero, carácter, booleano, etc.), **identificador**, **contenido** y memoria **ocupada**.

Dirección de memoria ->	12345	12346	12347	12348	12349	12350
Contenido de la variable	8	28	921427011			
Nombre de las Variables	Nota1	Alumnos	Tléfono			

1.5.5.Operadores.

Son símbolos que representan las distintas **operaciones** que se pueden realizar con los datos. Su cantidad y símbolo dependen del lenguaje utilizado. De acuerdo con su tipo se pueden clasificar en:

- **Aritméticos**: Suma, resta, multiplicación, división, potenciación, división entera, módulo o resto de división entera...
- **Alfanuméricos**: Concatenación.
- **De asignación**: Para establecer nuevos valores de las variables.
- **Relacionales**: Que permiten realizar comparaciones.
- **Lógicos**: AND, OR, OR exclusiva, NOT...

1.5.6.Expresiones.

Es la **combinación de operadores y datos** cuyo resultado es un valor. Es importante el orden de evaluación de los operadores y los paréntesis que es similar a los convenios matemáticos habituales.

1.5.7.Sentencias o instrucciones.

Son las acciones que forman el programa y se forman con expresiones válidas en el lenguaje de programación elegido. Las sentencias pueden ser:

- De **asignación**: Dar valor a las variables.
- De **entrada**: Recoge datos de un dispositivo de entrada (teclado, archivo, puerto, etc.) y asigna su valor a una variable.
- De **salida**: Coloca datos del programa en un dispositivo de salida (pantalla, archivo, impresora, etc.)
- **Condicionales**: Para codificar las tomas de decisiones del algoritmo.
- **Repetitivas**: Definidas para programar la repetición de bloques un determinado número de veces. Conceptos generales.

2.Representación gráfica de algoritmos.

2.1.Introducción.

Durante las fases de análisis de un problema, se hace necesaria la representación de las operaciones que el programa debe realizar y el orden en que se han de ejecutar.

Las representaciones más usadas son los **flujogramas**, los **diagramas NS** y el **pseudocódigo**.

En todo caso el algoritmo representado ha de ser **independiente del lenguaje** de programación.

Cuando la representación es gráfica, se obtienen los diagramas de flujo y pueden ser de dos tipos : Diagramas de flujo del sistema (**Organigramas**) o Diagramas de flujo del proceso (**Ordinogramas**).

2.2.Diagramas de flujo: Organigramas.

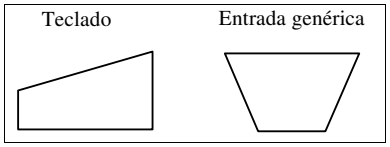
Los Organigramas describen el **flujo de datos** entre los distintos soportes que van a intervenir en el proceso. Se plantean de forma general, indicando los datos de entrada, el nombre de los programas o procesos, el soporte de los resultados y las líneas de flujo de datos a través de los procesos.

2.2.1.Símbolos.

Se utilizan una serie de símbolos que permiten representar los soportes de entrada y salida de datos, procesos y líneas de flujo.

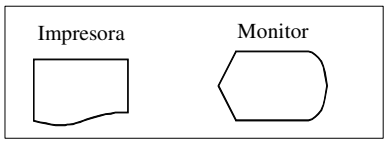
2.2.1.1.Símbolos de entrada.

Pueden ser el teclado o de tipo genérico, en cuyo caso se especificará el nombre.



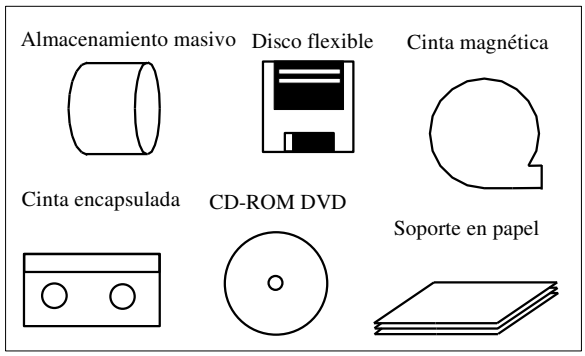
2.2.1.2.Símbolos de salida.

Los soportes de salida de datos son la impresora y el monitor.



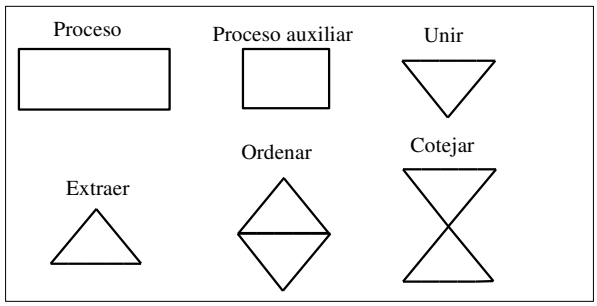
2.2.1.3.Símbolos de entrada/salida.

Hay soportes que permiten tanto la entrada como la salida de datos, en general, son soportes de **almacenamiento** de información.



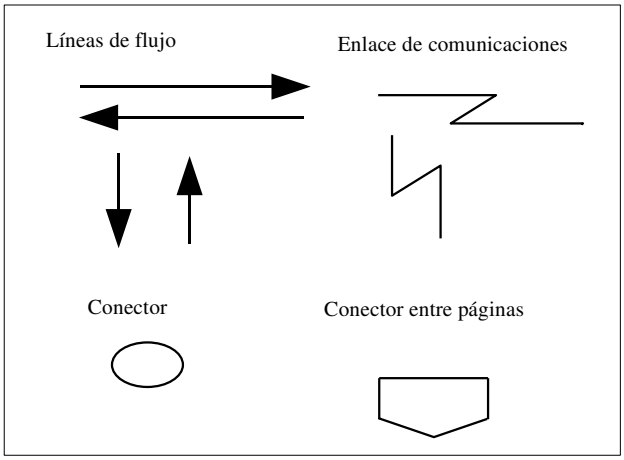
2.2.1.4.Símbolos de procesos.

Son los que representan las **operaciones** de procesado de la información.



2.2.1.5. Líneas y símbolos de flujo.

Indican el sentido de movimiento de la información.

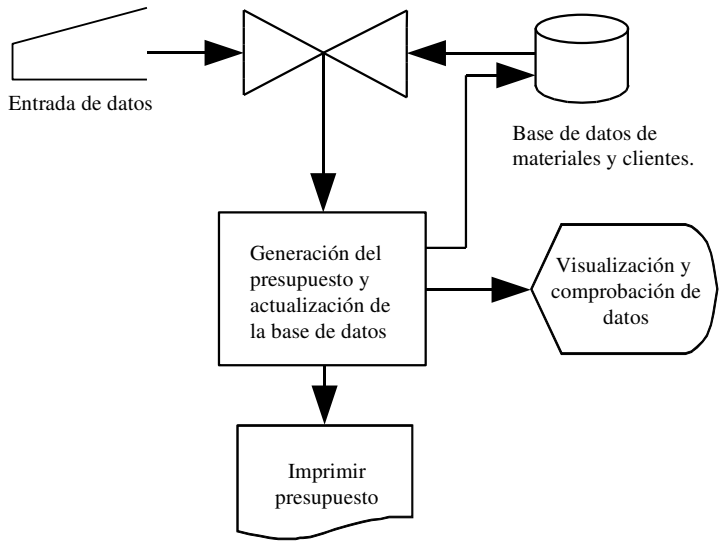


2.2.2.Normas de representación.

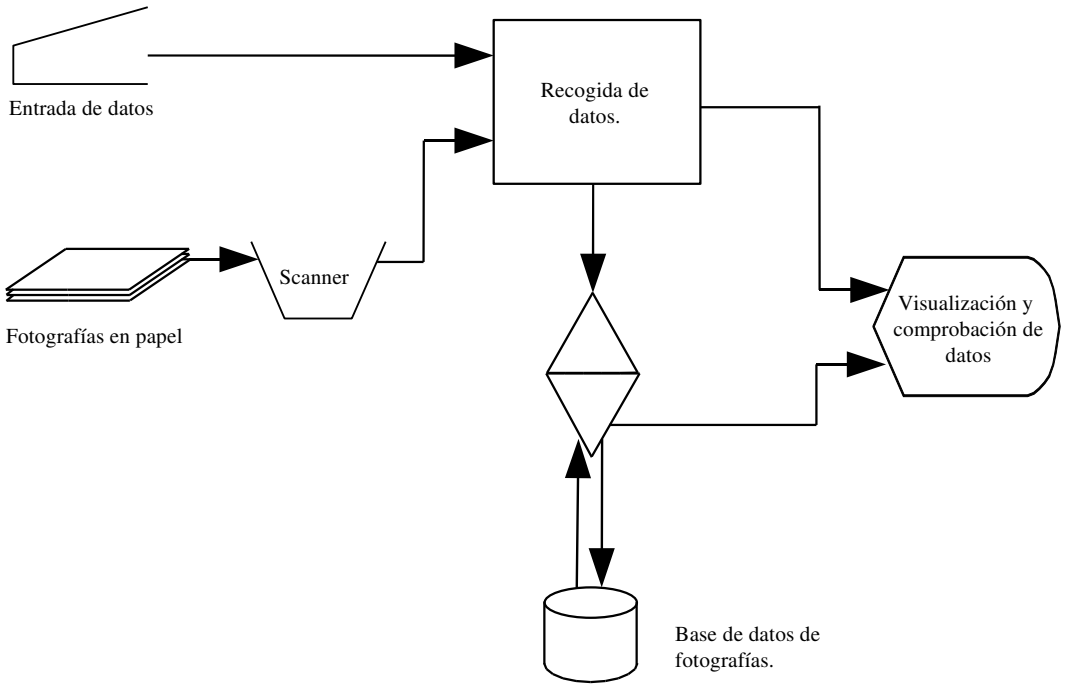
Existen unas reglas básicas para que la representación de los algoritmos de forma clara y sin ambigüedades, las más básicas son las siguientes:

- Los soportes de entrada de datos se situarán en la parte superior.
- En la parte central se situará el símbolo del proceso, con la indicación del tipo de programa.
- Los soportes de entrada/salida se situarán en ambos lados del símbolo del proceso.
- Los soportes de salida se situarán en la parte inferior.
- El flujo de datos irá de arriba abajo y de izquierda a derecha.
- Se debe procurar que el resultado sea simétrico, siempre que el mismo lo permita.

2.2.3. Ejemplos.



Presupuestos:
Representa el proceso de elaboración de presupuestos. Tomando los datos desde el teclado se **cotejan** con la base de datos. Los resultados se presentan en pantalla, se actualiza la base de datos, si es necesario, y se imprime el presupuesto.



Digitalización de fotografías:
Se utiliza un scanner para digitalizar las fotografías, y mediante el teclado de suministran los datos de cada una (nombre, tema, etc...). Se **ordenan**, se graban en disco y se visualizan los resultados.

2.3.Diagramas de flujo del proceso: Ordinogramas.

Es una notación gráfica para implementar algoritmos. Se basa en la utilización de unos símbolos gráficos que denominamos cajas, en las que escribimos las acciones (**procesos**) que tiene que realizar el algoritmo.

Las cajas están conectadas entre sí por líneas y eso nos indica el orden en el que tenemos que ejecutar las acciones.

En todo algoritmo siempre habrá una caja de **inicio** y otra de **fin**, para el principio y final del algoritmo.

2.3.1.Símbolos.

Representan los distintos tipos de **operaciones** en el programa, los puntos de **decisión**, los **comentarios** y las l**íneas de flujo** y conexión.

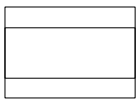
2.3.1.1.Símbolos de operación.

Permiten distinguir el tipo de operación que se realiza en el proceso en cada momento.

Principio y fin: Dentro del símbolo irá la palabra inicio o fin del algoritmo.

Símbolo de **proceso**: Indica la acción que tiene que realizar el ordenador. Dentro escribimos la acción.

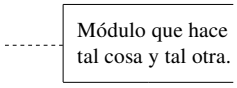
Representa las acciones de **entrada y salida**. Dentro colocaremos las acciones de lectura y escritura.



Subprograma: Dentro se coloca el nombre del subprograma al que se llama.

2.3.1.2.Símbolos de comentario.

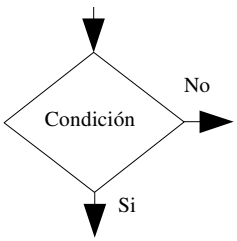
Permiten incluir informaciones y aclaraciones al algoritmo, para facilitar la comprensión del mismo. Es una aclaración para entender mejor el código, pero **no es parte del código**, no se ejecuta.



Se suele representar por medio de una línea de trazos, que parte del módulo al que se refiere el comentario.

2.3.1.3.Símbolos de decisión.

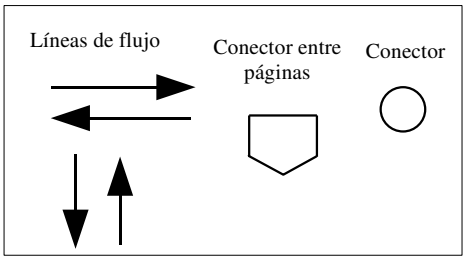
Permiten la toma de decisiones por parte del programa **bifurcando** el flujo dependiendo de una **condición**.



Las condiciones serán del tipo A>B, A=b, A<b, etc...

2.3.1.4.Símbolos de flujo y conexión.

Los símbolos de flujo son los que indican el sentido de flujo del programa y los de conexión se utilizan para conectar las líneas de flujo.



En los conectores se suele incluir un número de correspondencia para indicar la continuidad del flujo.

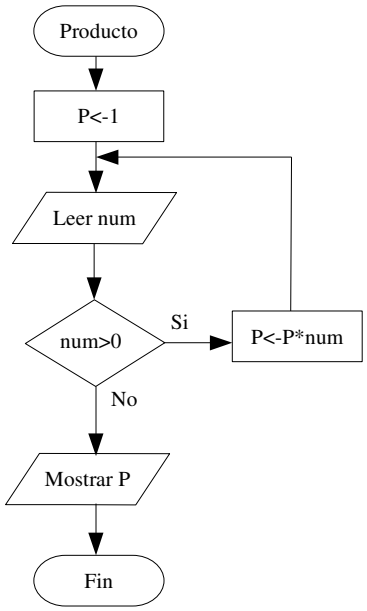
2.3.2.Normas de representación.

Las reglas más importantes son:

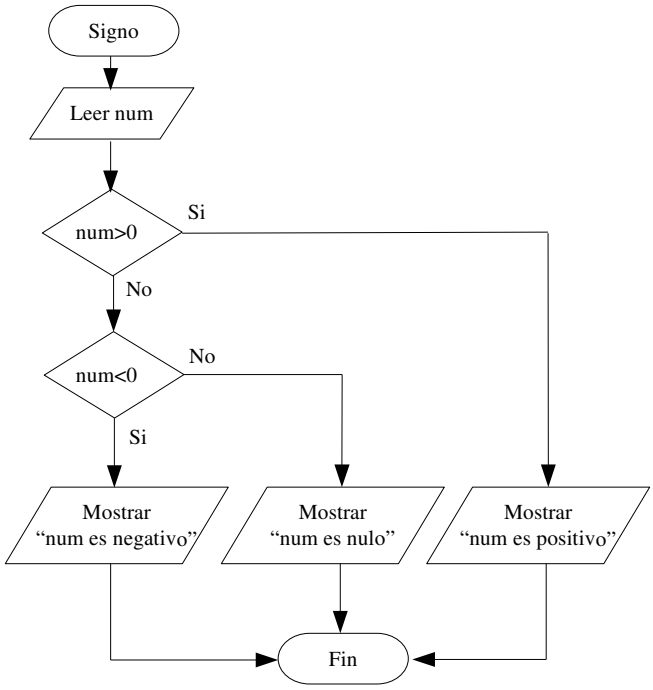
- El comienzo del programa debe situarse en la parte superior y el final en la inferior, situando los símbolos de INICIO y FIN.
- El flujo del programa irá de arriba abajo y de izquierda a derecha.
- Se procurará evitar cruces entre las líneas de flujo.
- Deben incluirse los comentarios justos para facilitar la legibilidad del ordinograma.
- Pueden emplearse varias hojas utilizando los conectores adecuados.
- Las indicaciones dentro de los símbolos deben ser independientes del lenguaje de programación.
- A cada símbolo excepto INICIO llegará una sola línea de flujo.
- De cada símbolo excepto los de decisión y FIN saldrá una única línea de flujo.
- El ordinograma debe ser claro, sin ambigüedades y lo más simétrico posible.

2.3.3.Ejemplos.

Queremos hallar el producto de varios números positivos introducidos por teclado.
El proceso termina cuando se meta un número negativo.



Hay que determinar el signo de un número introducido por teclado.

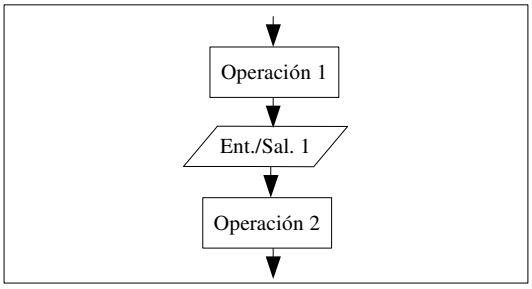


2.4.Técnicas de programación estructurada.

Se basa en escribir programas utilizando exclusivamente tres tipos de estructuras básicas: **Secuenciales**, **condicionales** y **repetitivas**.

2.4.1.Estructura secuencial.

Las acciones se ejecutan de forma encadenada y **sucesiva**.

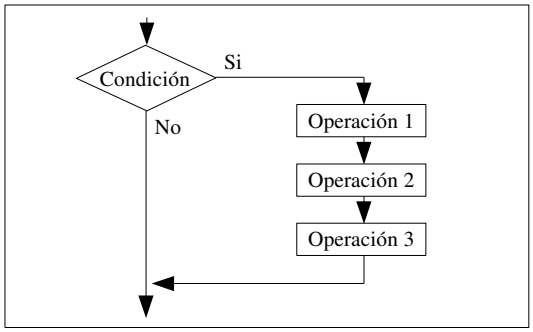


2.4.2.Estructura condicional.

Permite la **toma de decisiones** en función de una **condición** en la que intervienen las variables que forman parte del proceso. Puede ser de tres tipos: **simple**, **doble** y **múltiple**.

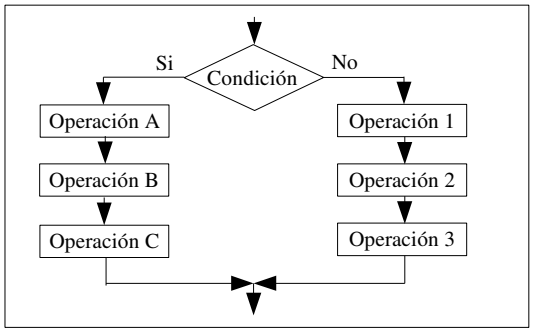
2.4.2.1.Condición simple.

Según se cumpla o no la condición se ejecutará o no una serie de sentencias.



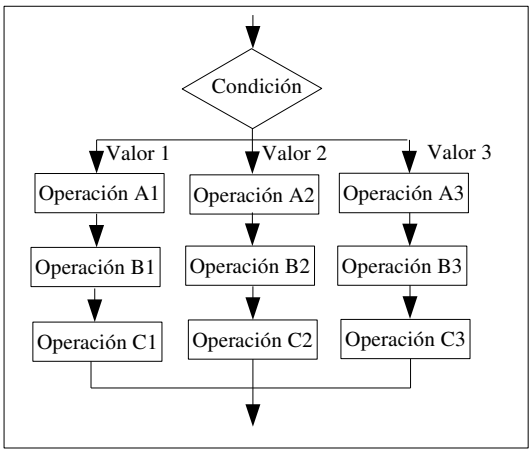
2.4.2.2.Condición doble.

Se evalúa la condición y en función del resultado se opta por la ejecución de uno de los dos conjuntos diferentes de sentencias.



2.4.2.3.Condición múltiple.

Permite programar tantos caminos distintos para el flujo de ejecución como valores puede tomar una variable o expresión del programa. (Se asocia con la sentencia **case**).



2.4.3.Estructura repetitiva

También denominada **bucle**, permite la ejecución repetida de una o varias operaciones mientras se verifique cierta condición. La condición de salida es aquella que determina la finalización de la repetición. Pueden ser de cuatro tipos: tipo **mientras**, tipo **hasta**, tipo **para** y tipo **iterar**.

2.4.3.1.Tipo mientras.

Se asocia a la sentencia **while**. Permite programar un conjunto de operaciones que se estará ejecutando **mientras** se cumpla una condición. La comprobación de la condición se realiza al **principio del bloque** de operaciones.

2.4.3.2.Tipo hasta.

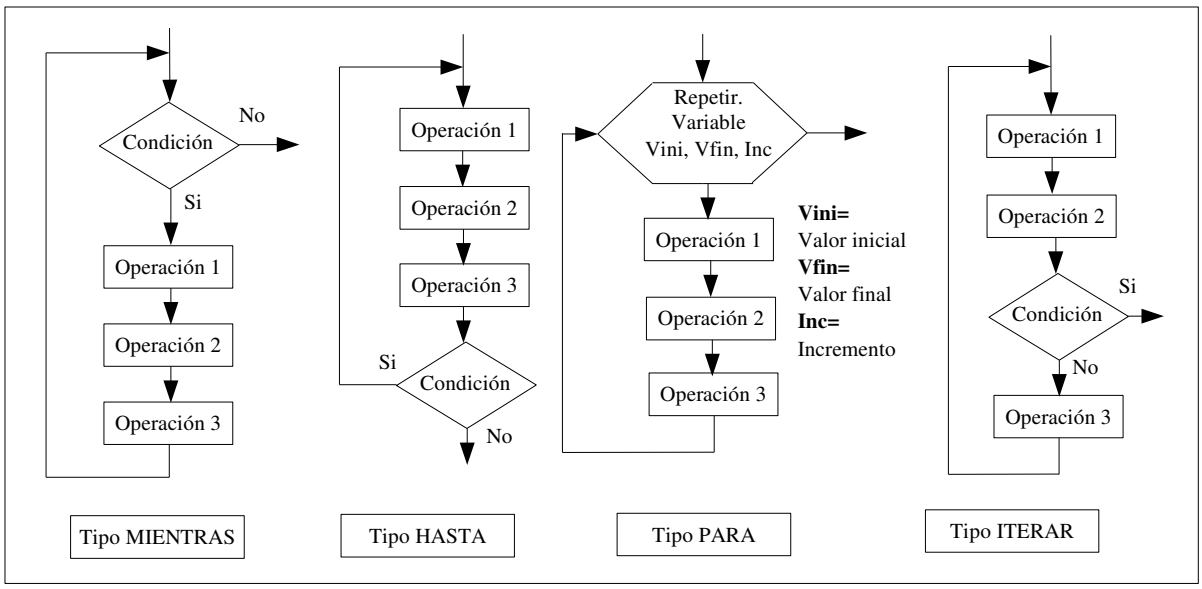
Se asocia a la sentencia **do.....while**. Permite programar un conjunto de acciones que se ejecutarán **hasta** que la condición deje de cumplirse. En este caso, la comprobación de la condición se realiza al final del bloque de operaciones.

2.4.3.3.Tipo para.

Se asocia a la sentencia **for**. Se utiliza cuando un conjunto de acciones deben ejecutarse un número específico de veces. Es similar a la estructura mientras, aunque con la diferencia de que controla el número de veces de forma automática.

2.4.3.4.Tipo iterar.

Es similar a la estructura **mientras** y **hasta**, con la particularidad de que la verificación de la condición de salida está intercalada entre las distintas operaciones.



2.5.Ejercicios.

Realizar mediante **ordinogramas** los siguientes programas:

a) Mostrar el resultado de realizar las operaciones básicas entre dos números introducidos por teclado.

b) Visualizar el mayor de dos números introducidos por teclado.

c) Presentar el factorial de un número introducido por teclado.

d) Mostrar la suma de diez números introducidos desde teclado.

e) Juego de adivinar un número generado aleatoriamente por el ordenador.

f) Presentar el resultado de elevar un número a otro introducidos por teclado, mediante multiplicaciones sucesivas.

g) Mostrar el cociente y el resto de dos números introducidos por teclado.

h) Desglosar una cantidad de dinero en el menor número de monedas de curso legal posible.

3.Pseudocódigo.

3.1.Introducción.

Es un lenguaje de especificación de algoritmos, pero muy parecido a cualquier lenguaje de programación, por lo que luego su traducción es muy sencilla, pero con la ventaja de que no se rige por las normas de un lenguaje en particular.

Está muy **próximo al lenguaje natural** y nos permite, en cierta medida, utilizar un lenguaje personalizado. Frente a las representaciones gráficas ocupa un espacio más reducido.

3.2.Normas generales.

No existe un estándar y se permite que cada programador utilice su propio lenguaje pseudocódigo, pero es recomendable tener en cuenta las siguientes normas:

- Indicar claramente el comienzo y el final del programa.
- Utilizar una línea para cada operación.
- Establecer un conjunto de palabras reservadas que pueden ser en Castellano. (inicio, fin, si, mientras, repetir, etc.).
- Utilizar tabulación para indicar los niveles de indentación, se pueden utilizar corchetes para los bloques de código.
- Cada estructura de control tendrá un único punto de entrada y otro de final.
- Se expresará en minúsculas, utilizando las mayúsculas para nombres de variables, ficheros, otros módulos, etc.
- Las referencias a otros módulos, subprogramas o subrutinas se harán en mayúsculas y entre los símbolos “<” y “>”, por ejemplo: <NUMBREMODULO>.

3.3.Elementos.

Se utilizan una serie de palabras clave que van indicando lo que significa el algoritmo.

3.3.1Elementos básicos.

- Nombre del algoritmo: **algoritmo**. Ej. algoritmo [NOMBRE_DEL_ALGORITMO]. Se utiliza al comienzo del programa para indicar el nombre del algoritmo.
- Inicio del algoritmo: **inicio**. Determina donde comienza el código.
- Fin del algoritmo: **fin**. Determina el final del código.
- Asignación: Se utiliza el símbolo “**<-**” o “**=**”.

```
Ej. : VARIABLE = EXPRESIÓN.
```

Indica la actualización de una variable con el resultado de una expresión.

- Entrada: **leer**. Ej. :

```
leer VARIABLE
```

Determina la actualización de una variable con un valor leído.

- Salida: **mostrar**. Ej.:

```
mostrar VARIABLE
mostrar EXPRESIÓN
```

Indica que el valor de la variable o expresión debe enviarse al dispositivo de salida.

- Declaración de variables: **var**. Ej.:

```
var [NOMBRES]: [tipo]
```

Se suele utilizar en bloque para declarar todas las variables del algoritmo.

- Declaración de constantes: **const**. EJ:

```
const [NOMBRES]:[tipo]
```

Se suele utilizar como en el caso anterior.

Con esto el diagrama general de un algoritmo será:

```
Algoritmo <nombre del algoritmo>

Var
    <nombre>: <tipo>

Inicio

    <Instrucciones>
Fin
```

3.3.2Estructuras en pseudocódigo.

Se pueden realizar las mismas estructuras que con los ordinogramas: **secuencial**, **condicionales** y **repetitivas**. No todas se pueden implementar en todos los lenguajes de programación y en alguna de ellas hay ligeras diferencias.

3.3.2.1.Estructura secuencial.

Las distintas operaciones se realizan una a continuación de la otra y en orden descendente.

```
...
acción_1
...
acción_n
...
```

3.3.2.2.Estructura condicional simple.

Si se cumple la condición se ejecutan las acciones 1, 2,..., n y <mas_cosas> y si no se cumple, salta directamente a <mas_cosas>.

```
...
si <condición> entonces
    <acción_1>
    <acción_2>
    ...
    <acción_n>
fin_si
<mas_cosas>
...
```

3.3.2.3.Estructura condicional doble.

Si se cumple la condición se ejecutan las acciones1, ..., n y <mas_cosas> y si no se cumple las acciones A, ..., Z y <mas_cosas>.

```
...
si <condición> Entonces
    <acción_1>
    ...
    <acción_n>
si_no
    <acción_A>
    ...
    <acción_Z>
fin_si
<mas_cosas>
...
```

3.3.2.4.Estructura condicional múltiple.

Dependiendo del valor de la VARIABLE ejecutará las diferentes acciones.

```
...
Según VARIABLE
    =VALOR1 hacer
           <acción_A1>
           ...
           <acción_An>
    =VALOR2 hacer
           <acción_B1>
           ...
           <acción_Bn>
           ...
    =VALORN hacer
           <acción_Z1>
           ...
           <acción_Zn>
fin_según
...
```

3.3.2.5.Estructura de tipo mientras.

Mientras se cumple la condición se ejecutan las acciones. La evaluación de la condición se realiza al comienzo del bucle.

```
...
mientras <condición>
    <acción_1>
    <acción_2>
    ...
    <acción_n>
fin_mientras
...
```

3.3.2.6.Estructura de tipo hasta.

Se ejecutan las acciones hasta alcanzar la condición. Las acciones se ejecutan al menos una vez realizándose la evaluación de la condición al final del bucle.

```
...
repetir
    <acción_1>
    <acción_2>
    ...
    <acción_n>
hasta <condición>
...
```

3.3.2.7.Estructura de tipo para.

Nos permite ejecutar las acciones un determinado número de veces controlado por una variable. La variable se inicia con el valor Vini y se incrementa con el valor Vinc hasta conseguir Vfin.

```
...
para <variable>  desde Vini
                 Hasta Vfin
                 Incrementando Vinc

    <acción_1>
    <acción_2>
    ...
    <acción_n>
fin_para
...
```

3.3.2.8.Estructura de tipo iterar.

El conjunto de acciones se repite hasta que se alcanza la condición, que está situada dentro del bucle.

```
iterar
    <acción_1>
    <accion_2>
    ...
    <acción_n>
salir_si <condición>
    <acción_a>
    <acción_b>
    ...
    <acción_z>
fin_iterar
```

3.4.Ejemplos.

Programa que realiza las operaciones matemáticas básicas con dos números introducidos por teclado.

```
algoritmo operaciones
var
    a,b,suma,resta,producto,
    cociente : entero
inicio
    leer a
    leer b
    suma=a+b
    resta=a-b
    producto=a*b
    cociente=a/b
    mostrar “El resultado es:”
    mostrar suma, resta, producto,
                cociente
fin
```

Programa que muestra el mayor de dos números introducidos por teclado.

```
algoritmo mayor
var
    a, b : entero
inicio
    leer a y b
    si a>b entonces
           mostrar a
    si_no
           si a=b entonces
                  mostrar "Los números
                         son iguales"
           si_no
                  mostrar b
           fin_si
    fin_si
fin
```

Programa que muestra el factorial de un número introducido por teclado.

```
algoritmo factorial
var
    n, factorial: entero
Inicio
    leer n
    factorial=1
    mientras n>0
           factorial=factorial*n
           n=n-1
    fin_mientras
    mostrar "El factorial de ",
            n, "es ", factorial
fin
```

Programa que muestra la suma de diez números desde el teclado.

```
algoritmo suma_diez
var
    dato[10]: array entero
    suma: entero
inicio
    n=0
    resultado=0
    repetir
           introducir dato[n]
           resultado=resultado+dato[n]
           n=n+1
    hasta n=10
fin
```

3.5.Ejercicios.

Realizar mediante pseudocódigo los siguientes programas:

a) Mostrar la suma, por separado, de los números pares e impares comprendidos entre dos números introducidos por teclado.

b) Juego de adivinar un número generado aleatoriamente por el ordenador.

c) Resolución de una ecuación de segundo grado.

d) Presentar el resultado de elevar un número a otro introducidos por teclado, mediante multiplicaciones sucesivas.

e) Mostrar el cociente y el resto de dos números introducidos por teclado.

f) Desglosar una cantidad de dinero en el menor número de monedas de curso legal posible.

g) Mostrar n números leídos desde teclado en el orden inverso al de su introducción.

4.Elementos de programación.

4.1..Introducción.

Al realizar el algoritmo para la solución de problemas, es necesario emplear elementos sin los cuales la solución sería más compleja. Se trata de elementos básicos que nos ayudan a or**ganizar los datos** que se manejan en el algoritmo.

4.2.Elementos auxiliares de programación.

Son elementos que **contienen datos que no forman parte del problema**, pero que son creados por el programador para facilitar la resolución del problema. A este tipo corresponden los **contadores**, **acumuladores** y **banderas**.

4.2.1.Contadores.

No son más que **variables** cuyo valor se **incrementa o decrementa**, siempre en la misma cantidad mediante una instrucción del tipo:

```
contador=contador +- incremento.
```

Normalmente va asociado a un bucle para controlar el número de interacciones.

4.2.2.Acumuladores.

Es similar al contador, pero en este caso la operación realizada con el incremento no se limita a suma y resta y las variaciones no son fijas, si no que vienen determinadas con el programa. Se realiza con una instrucción del tipo:

```
acumul= acumul <operación> expresión.
```

4.2.3.Banderas.

También llamadas **flags**, **interruptores** o **switch**. Es una variable que solo puede tomar **dos valores posibles**: (verdadero/falso), (1/0), (si/no), (on/off), etc. Se utilizan como indicadores de condiciones y permiten la toma de decisiones.

4.3.Estructuras de datos.

Son constantes o variables creadas por el programador para ordenar los datos y reunirlos de forma que puedan ser manejados más fácilmente.

4.3.1.Tablas o arrays.

Es la típica , constituida por un número fijo de elementos del mismo tipo. Están almacenados de forma **consecutiva** en memoria formando un conjunto compacto, pudiendo ser tratados como variables independientes.

Según el número de dimensiones, los arrays pueden ser: **unidimensionales**, **bidimensionales** o **multidimensionales**.

4.3.1.1Array unidimensional.

En las tablas unidimensionales los elementos son referenciados por el nombre dado a la tabla seguido de **un índice**, que hace referencia a la posi-

ción que ocupa el elemento.

	Lista[0]	Lista[1]	Lista[2]	.....	Lista[n]
Tabla Lista	valor 1	valor 2	valor 3		valor n

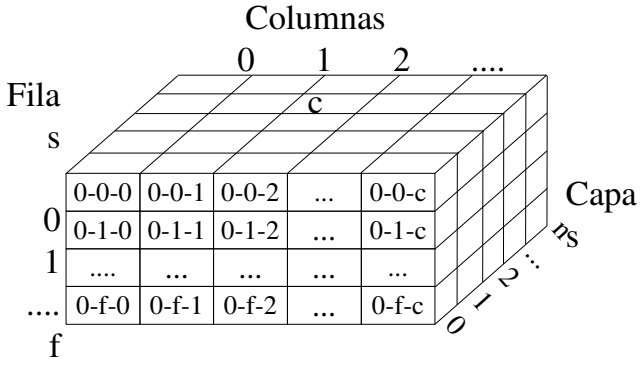
4.3.1.2Array bidimensional.

Las tablas bidimensionales o matrices son un conjunto de elementos referenciados por **dos índices**, uno correspondiente a la **fila** y otro a la **columna** que ocupa el elemento.

	Columna 0	Columna 1	Columna 2	...	Columna n
Fila 0	[0-0]	[0-1]	[0-2]	[0-...]	[0-n]
Fila 1	[1-0]	[1-1]	[1-2]	[1-...]	[1-n]
...	[...-0]	[...-1]	[,..-2]	[..,-,..]	[...-n]
Fila n	[n-0]	[n-1]	[n-2]	[n-...]	[n-n]

4.3.1.3.Array multidimensional.

Las tablas multidimensionales o poliedros son difíciles de manejar por lo que las de tres dimensiones son casi las únicas que se utilizan. Tienen **tres índices** que son, por orden: profundidad o **capa**, **fila** y **columna**.



4.3.2.Archivos

Se utilizan para el almacenamiento masivo y **permanente** de información.

Los archivos se componen de **registros**, que equivalen a fichas y son las unidades de acceso y tratamiento de la información.

En las operaciones de lectura o escritura en un archivo el sistema hace uso de los **buffers de memoria**, lugar de almacenamiento intermedio de la información. El buffer no tiene el mismo tamaño de los registros,

Los registros están compuestos por **campos** que pueden contener datos de diferente tipo y pueden, a su vez, contener subcampos.

De acuerdo con el uso y desde el punto de vista del programa, los archivos pueden ser:

- **Permanentes**: Contienen información principal, que no cambia.
- **De movimiento**: Se crean y actualizan con cierta frecuencia.
- **Temporales**: Almacenamiento auxilia. Tienen el tiempo de vida que dure su cometido.

Según la forma en que se organice el almacenamiento y acceso a la información pueden ser:

- **Secuenciales**: Los registros se colocan de forma consecutiva en el archivo.
- **Aleatorios**: Los registros se colocan de forma aleatoria, sin orden determinado.
- **Indexada**: Se establecen dos áreas en cada archivo: Una de índices, que permite localizar los registro, otra de datos, que contiene los registros.

4.4.Ejercicios.

a) Utilizando contadores, realiza el ordinograma y pseudocódigo de un programa que lea 10 números del teclado y cuente los que son positivos.

b) Empleando acumuladores, realiza el ordinograma y pseudocódigo de un programa que calcule la suma y el producto de todos los números comprendidos entre dos introducidos por teclado.

c) Mediante un swich, representa el ordinograma y pseudocódigo de un programa que sume separadamente los números pares y los impares, comprendidos entre 1 y 100.

d) Obtener el ordinograma y pseudocódigo de un programa que lea las calificaciones de un alumno, en las distintas materias, las guarde en un array de n elementos y calcule la nota media.

e) Realizar el ordinograma y pseudocódigo de un programa que localice el mayor y el menor de dos números, en un array desordenado.

f) Lograr el ordinograma y pseudocódigo de un programa que presente la tabla de multiplicar del 1 al 10.

5.Introducción al lenguaje C.

El lenguaje de programación C está caracterizado por ser de **uso general**, con una sintaxis sumamente compacta y de alta portabilidad.

Es común leer que se lo caracteriza como un lenguaje de "bajo nivel". No debe confundirse el término "bajo" con "poco", ya que el significado del mismo es en realidad "profundo", en el sentido que C maneja los elementos básicos presentes en todas las computadoras: caracteres, números y direcciones.

Esta particularidad, junto con el hecho de no poseer operaciones de entrada-salida, manejo de caracteres, de asignación de memoria, etc. , puede al principio parecer un grave defecto; sin embargo el hecho de que estas operaciones se realicen por medio de llamadas a **Funciones** contenidas en **Librerías** externas al lenguaje en sí, es el que confiere al mismo su alto grado de portabilidad, independizándolo del "Hardware" sobre el cual corren los programas, como se irá viendo a lo largo de los siguientes capítulos.

La descripción del lenguaje se realiza siguiendo las normas del **ANSI C (American National Standars Institute)**, por lo tanto, todo lo expresado será utilizable con cualquier compilador que se adopte; sin embargo en algunos casos particulares se utilizan funciones dependientes del Compilador ó Sistema Operativo, expresándose en estos casos la singularidad de las mismas.

5.1.Características.

Este lenguaje ha tenido mucho éxito, lo que ha hecho que se haya extendido rápidamente y que sea fácil encontrar compiladores para distintas máquinas y entornos. Esto se debe a sus características ya que es **compacto**, **estructurado**, **portable**, **flexible** y de **tipo medio**.

5.1.1.Compacto.

Se dice que es compacto por su reducido número de palabras reservadas, sólo 32 de acuerdo con el estándar ANSI C:

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**	

Además se utilizan símbolos para representar las **operaciones algebraicas**, **lógicas** y **relacionales** de las matemáticas convencionales y propias de la programación.

Se puede decir que cualquier programa se puede escribir utilizando exclusivamente las palabras reservadas y operadores, pero sería muy laborioso y requeriría un profundo conocimiento de la máquina con la que se está trabajando.

5.1.2.Estructurado.

Se basa en el empleo de **funciones**, bloques programados para realizar tareas simples y que, unidas entre sí, consiguen programas más complejos y de alto nivel.

Las funciones pueden verse como programas reducidos con datos **privados e independientes** del programa del que forman parte y que pueden utilizarse múltiples veces, realizando operaciones

simples.

Además, en C, se pueden crear bloques de código formando una unidad lógica.

En lenguaje C se pueden emplear sentencias de **decisión** e **iteración** propias de lenguajes de alto nivel. Incluso posee la instrucción **goto, aunque se desaconseja su utilización** (está mal visto entre los programadores de C).

5.1.3.Portable.

El código fuente no depende del compilador ni de la máquina en que se ha escrito, por lo que el mismo código fuente puede ser compilado en distintas máquinas y distintos entornos, utilizando las librerías adecuadas.

5.1.4.Flexible.

El programador de C tiene pocas restricciones y de él dependen prácticamente todas las decisiones de control. Esto hace que el lenguaje C sea muy flexible. El inconveniente es que a veces no se producen mensajes de error cuando el programa no funciona correctamente.

5.1.5.Lenguaje de tipo medio.

El lenguaje C combina elementos y características de alto nivel (sentencias muy potentes) y características de lenguajes de bajo nivel como ensamblador (permite el control de la CPU a nivel de registros, e incluso de bits), con plena disponibilidad de memoria y puertos.

5.2.Funciones.

La función es la unidad primaria de programación, es decir, los programas se construyen a partir de funciones. Son bloques de código independientes que se comunican con la función principal recibiendo y devolviendo datos.

La forma genérica de una función en C es:

```
tipo_devuelto nombre_función
(lista_de_parámetros)
{
    cuerpo de la función:
    declaraciones, sentencias,
    otras funciones, etc.
}
```

Siendo:

tipo_devuelto el tipo de dato que la función devuelve al punto en que fue llamada,

nombre_función el nombre del bloque de código

y **lista_de_parámetros** es una relación de los argumentos con los que se llama a la función para que realice las operaciones necesarias.

Dentro de la función se incluyen las declaraciones, sentencias y proposiciones necesarias para desarrollar las actividades para las que ha sido diseñada.

5.3.Variables y constantes.

No son más que **posiciones de memoria** que se identifican por un nombre. Las variables pueden cambiar de valor a lo largo de la ejecución de un programa, o bien en ejecuciones distintas de un mismo programa. Además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos. Un ejemplo típico es el número ▪PI, que vale 3.141592654. Este valor,

con más o menos cifras significativas, puede aparecer muchas veces en las sentencias de un programa.

En C existen distintos tipos de variables y constantes:

- **Numéricas**. Son valores numéricos, **enteros** o **reales**. Se permiten también **octales** (números enteros en base 8) y **hexadecimales** (base 16).
- **Caracteres**. Cualquier carácter individual encerrado entre apóstrofos (tal como 'a', 'Y', ')', '+', etc.) es considerado por C como una constante carácter, o en realidad como un número entero pequeño (entre 0 y 255, o entre -128 y 127, según los sistemas). Existe un código, llamado código ASCII, que establece una equivalencia entre cada carácter y un valor numérico correspondiente.
- **Cadenas de caracteres**. Un conjunto de caracteres alfanuméricos encerrados entre comillas es también un tipo del lenguaje C, como por ejemplo: "espacio", "Esto es una cadena de caracteres", etc. En realidad son tratados como arrays de caracteres, finalizados con un carácter nulo.
- **Constantes simbólicas**. Las constantes simbólicas tienen un nombre (identificador) y en esto se parecen a las variables. Sin embargo, no pueden cambiar de valor a lo largo de la ejecución del programa.

5.4.Identificadores.

Los identificadores o **etiquetas** son los nombres con el que se hace referencia a una variable, constante o función. Cada lenguaje tiene sus propias reglas respecto a las posibilidades de elección de nombres para las funciones y variables. En ANSI C estas reglas son las siguientes:

- Un identificador se forma con una secuencia de letras (minúsculas de la a a la z; mayúsculas de la **A a la Z**) y dígitos (del **0 al 9**).
- El carácter subrayado o underscore (_) se considera como una letra más.
- Un identificador **no puede contener espacios en blanco**, ni otros caracteres distintos de los citados, como por ejemplo (*,;.:-+, etc.).
- El primer carácter de un identificador debe ser siempre una letra o un (_), es decir, **no puede ser un dígito**.
- Se hace distinción entre letras mayúsculas y minúsculas. Así, Masa es considerado como un identificador distinto de masa y de MASA.
- No son válidas las palabras reservadas, pero estas están en minúsculas.
- ANSI C permite definir identificadores de hasta 31 caracteres de longitud.

Ejemplos de identificadores válidos son los siguientes:

```
tiempo, distancia1, caso_A, PI,
velocidad_de_la_luz.
```

Por el contrario, los siguientes nombres **no son válidos**:

```
1_valor, tiempo-total, dolares$, %final.
```

Además de lo anterior hay una serie de recomendaciones que simplifican enormemente la tarea de programación y, sobre todo, de corrección y mantenimiento de los programas:

- En general es muy aconsejable elegir los nombres de las funciones y las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan, utilizando para ello tantos caracteres como sean necesarios. Es cierto

que los nombres largos son más laboriosos de teclear, pero en general resulta rentable tomarse esa pequeña molestia.

- Las funciones creadas por el programador pueden comenzar por una letra mayúscula, para distinguirlas de las definidas en el compilador.
- Los identificadores de constantes suelen escribirse en mayúsculas.

5.5.Sentencias.

Una sentencia es un conjunto de identificadores símbolos y palabras reservadas que se escriben en una línea. En C las sentencias terminan con un punto y coma ';'.

Pueden ir en cualquier posición de la línea, seguidas o en líneas diferentes, es decir, **no se tienen en cuenta los espacios vacíos ni los finales de línea**.

5.6.Indentación.

Es el sangrado de las líneas del programa, su misión es, exclusivamente, facilitar la lectura y escritura de los programas, ya que es ignorada por el compilador.

A la hora de establecer la indentación, el programador se atiene a criterios personales, pero estos deben ser fijos y estables para que den una idea clara de la subordinación entre sentencias.

5.7.Bloques de sentencias.

Son conjuntos de sentencias agrupadas entre llaves "**{ }**" que forman una unidad lógica inseparable y que, a todos los efectos, son como una **única sentencia**.

5.8.Comentarios.

El lenguaje C permite que el programador introduzca comentarios en los ficheros fuente. La misión de los comentarios es servir de explicación o aclaración sobre cómo está hecho el programa, de forma que pueda ser entendido por una persona diferente (o por el propio programador algún tiempo después).

Tienen las siguientes características:

- Los caracteres (**/***) se emplean para iniciar un comentario; el comentario termina con los caracteres (***/**).
- No se puede introducir un comentario dentro de otro.
- Todo texto introducido entre los símbolos de comienzo y final de comentario son ignorados por el compilador.

Por ejemplo:

```
var_1 = var_2;    /* En esta línea se
                  asigna a var_1 el
                  valor contenido en
                  var_2 */
```

Una fuente frecuente de errores al programar en C, es el olvidarse de cerrar un comentario que se ha abierto previamente. Sería un **error** como el siguiente:

```
var_1=var_2              /*Asigno valor
                         a var_1
variable_3=55     /*Asigno valor a
                  variable_3*/
```

El lenguaje ANSI C permite también otro tipo de comentarios, tomado del C++. Todo lo que va en cualquier línea del código detrás de la doble barra (**//**) y hasta el final de la línea, se considera

como un comentario y es ignorado por el compilador. Para comentarios cortos, esta forma es más cómoda que la anterior, pues no hay que preocuparse de cerrar el comentario (el fin de línea actúa como cierre). Como contrapartida, si un comentario ocupa varias líneas hay que repetir la doble barra (//) en cada una de las líneas.

Con este segundo procedimiento de introducir comentarios, el último ejemplo podría ponerse en la forma:

```
var_1 = var_2;    // En esta línea se
                  // asigna a var_1 el
                  //valor contenido en
                  //variable_2
```

5.9.El preprocesador.

El preprocesador es un componente característico de C, que no existe en otros lenguajes de programación. El preprocesador actúa sobre el programa fuente, **antes de que empiece la compilación** propiamente dicha, para realizar ciertas operaciones. Una de estas operaciones es, por ejemplo, la **sustitución de constantes simbólicas**. Así, es posible que un programa haga uso repetidas veces del valor 3.141592654, correspondiente al número PI. Es posible definir una constante simbólica llamada PI que se define como 3.141592654 al comienzo del programa y se introduce luego en el código cada vez que hace falta. En realidad PI no es una variable con un determinado valor: el preprocesador chequea todo el programa antes de comenzar la compilación y sustituye el texto PI por el texto 3.141592654 cada vez que lo encuentra. Las constantes simbólicas suelen escribirse completamente con mayúsculas, para distinguirlas de las variables.

El preprocesador realiza muchas otras funciones que se irán viendo a medida que se vaya explicando el lenguaje. Lo importante es recordar que actúa siempre por delante del compilador (de ahí su nombre), facilitando su tarea y la del programador.

5.10.Las librerías.

Con objeto de mantener el lenguaje lo más sencillo posible, muchas sentencias que existen en otros lenguajes, no tienen su correspondiente contrapartida en C. Por ejemplo, en C no hay sentencias para entrada y salida de datos. Es evidente, sin embargo, que ésta es una funcionalidad que hay que cubrir de alguna manera. El lenguaje C lo hace por medio de funciones preprogramadas.

Estas funciones están agrupadas en un conjunto de librerías de código objeto, que constituyen la llamada librería estándar del lenguaje. La llamada a dichas funciones se hace como a otras funciones cualesquiera, y deben ser declaradas antes de ser llamadas por el programa (más adelante se verá cómo se hace esto por medio de la directiva del preprocesador #include).

El estándar ANSI C especifica un conjunto mínimo de funciones que debe proporcionar todo compilador, con un nombre y características idénticas. Los compiladores suelen incluir muchas más pero que no son estándar.

5.11.El compilado.

El compilador es el elemento más característico del lenguaje C. Como ya se ha dicho anteriormente, su misión consiste en traducir a lenguaje de máquina (**código objeto**) el programa C contenido en uno o más ficheros fuente. El compilador es capaz de **detectar ciertos errores** durante el proceso de compilación, enviando al usuario el correspondiente mensaje de error.

5.12.El linkado.

Es la segunda etapa se realiza para la obtención del programa. Es el proceso de montaje (linkado) del programa, consistente en producir un programa **ejecutable** en lenguaje de máquina, en el que están ya incorporados todos los otros módulos que aporta el sistema sin intervención explícita del programador: funciones de librería (archivos *.LIB), recursos del sistema operativo, etc.).

En un PC con sistema operativo Windows (DOS) el programa ejecutable se guarda en un fichero con extensión (*.exe). En otros sistemas operativos existe un atributo del fichero que lo hace ejecutable. Este fichero es cargado por el sistema operativo en la memoria RAM cuando el programa va a ser ejecutado.

El código máquina de los ficheros objeto y de las librerías están en formato reubicable, es decir, no tienen asignadas direcciones de memoria reales, es el proceso de linkado el que las asigna, para que el programa ejecutable se sitúe correctamente en la memoria del ordenador.

5.13.Generación del código ejecutable.

Pasos a seguir para obtener el fichero ejecutable de un programa en C:

- Creación del **fichero fuente** con un editor de texto.
- Compilación del programa para obtener los **ficheros objeto** (*.obj u .o).
- Enlazado o linkado de los ficheros objeto y las funciones de biblioteca para obtener el **fichero ejecutable**.

En la mayoría de los compiladores, la compilación y el linkado, se realizan en una sola operación, puesto que utilizan entornos de programación que integran todos los recursos y utilidades necesarias para: editar, compilar, linkar y depurar el programa.

Cuando se trata de un proyecto grande, el programa se divide en varios archivos, que pueden compilarse por separado y enlazarse todos ellos juntos para obtener el ejecutable final.

5.14.Ejemplos.

A continuación se muestra uno de los programas más sencillos:

```
/* Primer programa*/
/*Fichero hola.c*/
#include <stdio.h>
main()           /*Función principal*/
{
    printf("Hola Caracola.\n");
}
```

La primera línea es una directiva al preprocesador que añade el fichero **stdio.h** al fichero fuente. En este fichero se encuentran las **declaraciones** y **definiciones** de las funciones estándar de retrada salida, entre ellas **printf()**.

La **función principal** (**main()**), debe existir siempre en todos nuestros programas. En este caso incluye una única sentencia, finalizada por '**;**'. Esta sentencia es una llamada a la función **printf()**, a la que pasa como **argumento** una cadena de texto para que sea mostrada en pantalla.

La secuencia que comienza por la barra invertida sirve para representar caracteres no imprimibles, en este caso '**\n**' es igual a retorno de carro y avance de línea (**RETURN**).

Se ha incluido un comentario entre los símbolos '**/***' y '***/**'.

Programa que permite convertir grados Fahrenheit en grados Celsius:

```
/* Conversión de Fahrenheit a Celsius*/
/* Fichero fahren1.c */

#include <stdio.h>

void main(void)
{
    int fahren, celsius;                          /*Variables enteras*/

    printf("Conversión de ºF a ºC:\n");           /*Mensaje de presentación*/
    fahren=100;
    celsius=5*(fahren-32)/9;               /*Fórmula de conversión*/

    printf("%d ºF = %d ºC\n",fahren, celsius);  /*Mostrar el resultado*/
}
```

Se puede observar la **declaración** de dos variables de tipo **entero** antes de su utilización.

En este caso se utiliza la función **printf()**, no solo para mostrar mensajes de texto en pantalla, sino también para mostrar los valores de las **variables** fharen y celsius. Se utiliza para ello el símbolo '**%**' seguido de un carácter que indica el tipo de dato que se desea representar, en este caso '**d**' (entero).

Ampliaremos el programa para poder introducir la temperatura por teclado:

```
/* Conversión de Fahrenheit a Celsius*/
/* Fichero fahren2.c */

#include <stdio.h>

void main(void)
{
    int fahren, celsius;                    /*Variables enteras*/

    printf("Conversión de ºF a ºC:\n");
    printf("Introduce la temperatura fahrenheit:");
    scanf("%d",&fahren);                    /*Introducción de datos
                                            por teclado*/
    celsius=5*(fahren-32)/9;         /*Fórmula de conversión*/

    printf("%d ºF = %d ºC\n",fahren, celsius);  /*Resultado*/
}
```

La lectura se realiza por medio de la función **scanf()**, que se encarga de leer del teclado un número entero ("**%d**") y almacenarlo en la variable fahren. Es **necesario** el símbolo '**&**' delante de la variable para que la función realice correctamente la lectura. Esto se aclarará en más adelante, cuando se estudie la función scanf().

La conversión no se realiza con precisión debido a que estamos utilizando variables enteras, el programa mejoraría si utilizásemos números **reales**.

```
/* Conversión de Fahrenheit a Celsius*/
/* Fichero fahren3.c */

#include <stdio.h>
```

```
void main(void)
{
    float fahren, celsius;           /*Variables reales*/

    printf("Conversión de ºF a ºC:\n");
    printf("Introduce la temperatura fahrenheit:");
    scanf("%f",&fahren);             /*Introducción de datos
                              por teclado*/
    celsius=5*(fahren-32)/9;         /*Fórmula de conversión*/

    printf("%f ºF = %f ºC\n",fahren, celsius);  /*Resultado*/
}
```

Hemos tenido que cambiar la declaración de las variables, así como los especificadores de formato utilizados por printf() y scanf() (**%f** para números reales.

5.15.Ejercicios.

a) Escribir los programas mostrados como ejemplo añadiendo los comentarios convenientes para entender su funcionamiento. Obtener los ejecutables y comprobar su funcionamiento utilizando un entorno de programación. Observa el proceso de compilación y linkado.

b) Dentro del entorno de programación elegido, hacer diferentes pruebas y modificaciones en los ficheros anteriores, para aclarar las dudas que puedan surgir en la utilización del entorno. Probar las opciones de ejecución paso a paso, depuración, puntos de ruptura, visualización de variables, etc.

c) Realizar un programa que convierta de pesetas a euros.

d) Construir un programa que calcule la longitud de una circunferencia a partir de su radio.

e) Realizar un programa que tome dos números desde el teclado y muestre su suma, diferencia, producto y cociente.

6.Datos en C.

6.1.Introducción.

El C, como cualquier otro lenguaje de programación, tiene posibilidad de trabajar con datos de distinta naturaleza: texto formado por ca**racteres** alfanuméricos, números **enteros**, números **reales** con parte entera y parte fraccionaria, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras (rango y/o precisión), posibilidad de ser sólo positivos o de ser positivos y negativos, etc. En este apartado se verán los tipos fundamentales de datos admitidos por el C. Más adelante se verá que hay otros tipos de datos, derivados de los fundamentales. Tipos de datos fundamentales del C:

Tipos de datos fundamentales (notación completa)			
Enteros	char	signed char	unsigned char
	signed short int	signed int	signed long int
	unsigned short int	unsigned int	unsigned long int
Reales	float	double	long double

La palabra **char** hace referencia a que se trata de un **carácter** (una letra mayúscula o minúscula, un dígito, un carácter especial, etc.).

La palabra **int** indica que se trata de un número **entero**.

La palabra **float** se refiere a un número **real** (también llamado de punto o coma flotante).

Los números enteros pueden ser positivos o negativos (**signed**), o bien no negativos (**unsigned**). Los caracteres tienen un tratamiento muy similar a los enteros y admiten estos mismos cualificadores. En los datos enteros, las palabras **short** y **long** hacen referencia al número de cifras o rango de dichos números.

En los datos reales las palabras **double** y **long** apuntan en esta misma dirección, aunque con un significado ligeramente diferente, como más adelante se verá.

Esta nomenclatura puede simplificarse: las palabras signed e int son las o**pciones por defecto** para los números enteros y pueden omitirse, resultando la siguiente tabla, que indica la nomenclatura más habitual para los tipos fundamentales del C.

Tipos de datos fundamentales (notación abreviada).			
Enteros	Char	signed char	unsigned char
	Short	int	Long
	unsigned short	unsigned	unsigned long
Reales	Float	double	long double

A continuación se va a explicar cómo puede ser y cómo se almacena en C un dato de cada tipo fundamental.

6.2.Tipos de datos fundamentales. Variables.

Recuérdese que en C es **necesario declarar todas las variables que se vayan a utilizar**. Una variable no declarada produce un mensaje de error en la compilación.

Cuando una variable es declarada se le reserva memoria de acuerdo con el tipo incluido en la declaración.

Es posible inicializar (dar un valor inicial) las variables en el momento de la declaración; ya se verá que en ciertas ocasiones el compilador da un valor inicial por defecto, mientras que en otros casos no se realiza esta inicialización y la memoria asociada con la variable correspondiente contiene basura informática (combinaciones sin sentido de unos y ceros, resultado de operaciones anteriores con esa zona de la memoria, para otros fines).

Según la máquina y el compilador que se utilice los tipos primitivos pueden ocupar un determinado tamaño en memoria. Lo que trataremos a continuación se aplica a la arquitectura x86.

Otras arquitecturas pueden requerir distintos tamaños de tipos de datos primitivos. ANSIC no dice nada acerca de cuál es el número de bits en un byte, ni del tamaño de estos tipos; más bien, ofrece solamente las siguientes "**garantías de tipos**":

- Un tipo **char** tiene el tamaño mínimo en bytes asignable por la máquina, y todos los bits de este espacio deben ser "accesibles".
- El **tamaño reconocido de char es de 1**. Es decir, sizeof(char) siempre devuelve 1.
- Un tipo **short** tiene al menos el mismo tamaño que un tipo char.
- Un tipo **long** tiene al menos el doble tamaño en bytes que un tipo short.
- Un tipo **int** tiene un tamaño entre el de short y el de long, ambos inclusive, preferentemente el tamaño de un apuntador de memoria de la máquina.
- Un tipo **unsigned** tiene el mismo tamaño que su versión signed.

6.2.1.Caracteres (tipo char).

Las variables **carácter** (tipo char) contienen un único carácter y se almacenan en **un byte** de memoria (8 bits, 256 valores diferentes entre 0 y 255 para variables sin signo, y de -128 a 127 para variables con signo, que es la opción por defecto.

La declaración de variables tipo carácter puede tener la forma:

```
char nombre;
char nombre1, nombre2, nombre3;
```

Se puede declarar más de una variable de un tipo determinado en una sola sentencia.

Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter *letra* y asignarle el valor -*a*-, se puede escribir:

```
char letra = 'a';
```

A partir de ese momento queda definida la variable *letra* con el valor correspondiente a la letra **a**. Recuérdese que el valor '*a*' utilizado para inicializar la variable letra es una **constante** carácter. En realidad, letra se guarda en un solo byte como un número entero, el correspondiente a la letra a en el código **ASCII**.

En la variable *letra*, su contenido puede ser variado cuando se desee por medio de una sentencia que le asigne otro valor, por ejemplo:

```
letra = 'z';
```

También puede utilizarse una variable char para dar valor a otra variable de tipo char:

```
caracter = letra;       /* Ahora
carácter es
                igual a 'z' */
```

Como una variable tipo char es un número entero pequeño (entre 0 y 255), se puede utilizar el contenido de una variable char **de la misma forma que se utiliza un entero**, por lo que están permitidas operaciones como:

```
letra = letra + 1;
letra_minus = letra_mayus + ('a' - 'A');
```

En el primer ejemplo, si el contenido de *letra* era una *a*, al **incrementarse** en una unidad pasa a contener una *b*. El segundo ejemplo es interesante: puesto que la diferencia numérica entre las letras minúsculas y mayúsculas es **siempre la misma** (según el código ASCII), la segunda sentencia pasa una letra mayúscula a la correspondiente letra minúscula sumándole dicha diferencia numérica.

Recuérdese para concluir que las variables tipo char son y se almacenan como **números enteros pequeños**. Ya se verá más adelante que se pueden escribir como caracteres o como números según que formato de conversión se utilice en la llamada a la función de escritura.

6.2.2.Números enteros (tipo int).

Una variable tipo **int** se almacena en 4 bytes (32 bits).

Con 32 bits se pueden almacenar 4,294967,296 números enteros diferentes: de 0 a 4,294,967,295 para variables sin signo, y de -2,147,483,648 a 2,147,483,647 para variables con signo, que es la opción por defecto.

Una variable entera (tipo int) se declara, o se declara y se **inicializa** en la forma:

```
unsigned int numero;
int nota = 10;
```

En este caso la variable *numero* podrá estar entre 0 y 4,294,967,295, mientras que *nota* deberá estar comprendida entre -2,147,483,648 y 2,147,483,647 .

Cuando a una variable int se le asigna en tiempo de ejecución un valor que queda fuera del rango permitido (situación de **overflow** o valor excesivo), se produce un **error** en el resultado de consecuencias tanto más imprevisibles cuanto que de ordinario el programa **no avisa** al usuario de dicha circunstancia.

Cuando el ahorro de memoria es muy importante puede asegurarse que el ordenador utiliza 2 bytes para cada entero declarándolo en una de las formas siguientes:

```
short numero;
short int numero;
```

En este caso la variable *numero* podrá estar entre -32.768 y 32.767.

6.2.3.Números enteros (tipo long).

Existe la posibilidad de utilizar enteros con un **rango mayor** si se especifica como tipo **long**:

```
long int numero_grande; /*o, ya que la
                  palabra clave int
                  puede omitirse en
                  este caso: */
long numero_grande;
```

El rango de un entero long puede variar según el ordenador o el compilador que se utilice, en nuestro caso coincide con el tamaño de un entero, por lo que se pueden representar 4.294.967.296 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre -2.147.483.648 y 2.147.483.647.

También se pueden declarar enteros long que sean siempre positivos con la palabra **unsigned**:

```
unsigned long positivo_muy_grande;
```

En algunos ordenadores una variable int ocupa 2 bytes (coincidiendo con short) y en otros 4 bytes (coincidiendo con long). Lo que garantiza el ANSI C es que el rango de int no es nunca menor que el de short ni mayor que el de long.

6.2.4.Números reales (tipo float).

En muchas aplicaciones hacen falta variables **reales**, capaces de representar magnitudes que contengan una parte entera y una parte fraccionaria o decimal. Estas variables se llaman también de punto flotante o coma flotante.

De ordinario, en **base 10** y con notación científica, estas variables se representan por medio de la **mantisa**, que es un número mayor o igual que 0.1 y menor que 1.0, y un **exponente** que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número considerado. Por ejemplo, **pi** se representa como $0.3141592654 \cdot 10^{1}$.

Tanto la mantisa como el exponente pueden

ser positivos y negativos.

Los ordenadores trabajan en base 2. Por eso un número de tipo float se almacena en 4 bytes (32 bits), utilizando **24 bits** para la **mantisa** (1 para el signo y 23 para el valor) y **8 bits** para el **exponente** (1 para el signo y 7 para el valor).

Es interesante ver qué clase de números de coma flotante pueden representarse de esta forma. En este caso hay que distinguir el **rango** de la **precisión**.

La precisión hace referencia al **número de cifras** con las que se representa la mantisa: con 23 bits el número más grande que se puede representar es 8.388.608 lo cual quiere decir que se pueden representar todos los números decimales de **6 cifras** y la mayor parte (aunque no todos) de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo float tienen entre **6 y 7 cifras** decimales equivalentes de precisión.

Respecto al exponente por el que hay que multiplicar la mantisa, 7 bits, el número más grande que se puede representar es 127 que será el exponente en base dos (2^{127}=1,7014e+38).

El rango vendrá definido por la combinación de mantisa y exponente. En la mayoría de los compiladores el rango de los números representables es de –3.4E+38 a –1.17E-38 y de 1.17E-38 a 3.4E+38.

Las variables tipo float se declaran de la forma:

```
float numero_real;
```

Las variables tipo float pueden ser inicializadas en el momento de la declaración, de forma análoga a las variables tipo int.

6.2.5.Números reales (tipo double).

Las variables tipo float tienen un rango y (sobre todo) una precisión muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo double, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan **53 bits** para la mantisa (1 para el signo y 52 para el valor) y **11 para el exponente** (1 para el signo y 10 para el valor). La precisión es en este caso, 2^{52} = 4.503.599.627.370.496 lo cual representa entre **15 y 16** cifras decimales equivalentes.

Con respecto al rango, con un exponente de 10 bits el número más grande que se puede representar será $1.7977 \cdot 10^{308}$.

Las variables tipo double se declaran de forma análoga a las anteriores:

```
double real_grande;
```

Por último, existe la posibilidad de declarar una variable como **long double**, aunque el ANSI C no garantiza un rango y una precisión mayores que las de double. Eso depende del compilador y del tipo de ordenador.

Estas variables se declaran en la forma:

```
long double real_pero_que_muy_grande;
```

Cuyo rango y precisión no está normalizado. En nuestro caso se utilizan 12 bytes.

6.2.6.Duración y visibilidad de las variables: Modos de almacenamiento.

El tipo de una variable se refiere a la naturaleza de la información que contiene (ya se han visto los tipos char, int, long, float, double, etc.).

El modo de almacenamiento (storage class) es otra característica de las variables de C que determina cuándo se crea una variable, cuándo deja

de existir y desde dónde se puede acceder a ella, es decir, desde dónde es visible.

En C existen 4 modos de almacenamiento fundamentales: **auto**, **extern**, **static** y **register**. Seguidamente se exponen las características de cada uno de estos modos.

6.2.6.1.auto (automático).

Es la opción por defecto para las variables que se **declaran dentro de un bloque {...}**, incluido el bloque que contiene el código de las funciones. En C la declaración debe estar siempre al **comienzo del bloque**. En C++ la declaración puede estar en cualquier lugar y hay autores que aconsejan ponerla justo antes del primer uso de la variable. No es necesario poner la palabra *auto*.

Cada variable *auto* es creada al **comenzar** a ejecutarse el bloque y deja de existir cuando el bloque se **termina** de ejecutar. Cada vez que se ejecuta el bloque, las variables *auto* se **crean y se destruyen** de nuevo.

Las variables *auto* son variables **locales**, es decir, sólo son visibles en el bloque en el que están definidas y en otros bloques anidados en él, aunque pueden ser ocultadas por una nueva declaración de una nueva variable con el mismo nombre en un bloque anidado.

No son inicializadas por defecto, y (antes de que el programa les asigne un valor) pueden contener basura informática (conjuntos aleatorios de unos y ceros, consecuencia de un uso anterior de esa zona de la memoria). A continuación se muestra un ejemplo de uso de variables de modo *auto*:

```
{
int i=1, j=2;     /* se declaran e
                  inicializan i y j */
    {
          float a=7., j=3.;
                  /* se declara una
                  nueva variable j */
          ...
          j=j+a;
                  /* aqui j es float
                  la variable int j es
                  invisible
                  la variable i=1 es
                  visible */
    }
          /* fuera del bloque, a ya no
          existe, la variable j=2
          existe y es entera */
}
/*fuera de los dos bloques no existen i,
j y a */
```

6.2.6.2.Extern.

Son variables **globales**, que se definen **fuera de cualquier bloque o función**, por ejemplo antes de definir la función *main()*. Estas variables existen durante toda la ejecución del programa. Las variables *extern* son visibles por todas las funciones que están entre la definición y el fin del fichero. Para verlas desde otras funciones definidas anteriormente o desde otros ficheros, deben ser declaradas en ellos como variables **extern**. Por defecto, **son inicializadas a cero**.

Una variable *extern* es definida o creada (una variable se crea en el momento en el que se le reserva memoria y se le asigna un valor) una sola vez, pero puede ser declarada (es decir, reconocida para poder ser utilizada) varias veces, con objeto de hacerla accesible desde diversas funciones o ficheros.

También estas variables **pueden ocultarse** mediante la declaración de otra variable con el mismo nombre en el interior de un bloque.

Las variables *extern* permiten transmitir valores entre distintas funciones, pero ésta es una práctica considerada como **peligrosa**. A continuación se presenta un ejemplo de uso de variables *extern*:

```
int i=1, j, k;    /* se declaran antes
                  de main() */
main()
{
int i=3;          /* i=1 se hace
                  invisible */
int func1(int, int);    /* j, k son
....                          visibles
*/
....
```

```
}

int func1(int n, int m)
{
1 int k=3; // k=0 se hace invisible
... // i=3 es invisible
}
```

6.2.6.3.Static.

Cuando ciertas variables son declaradas como *static* dentro de un bloque, conservan su valor entre distintas ejecuciones de ese bloque. Dicho de otra forma, las variables *static* se declaran dentro de un bloque como las *auto*, pero permanecen en memoria durante toda la ejecución del programa como las *extern*.

Cuando se llama varias veces sucesivas a una función (o se ejecuta varias veces un bloque) que tiene declaradas variables *static*, los valores de dichas variables se conservan entre dichas llamadas. La inicialización sólo se realiza la **primera vez**. Por defecto, son inicializadas a **cero**.

Las variables definidas como *static extern* son visibles sólo para las funciones y bloques comprendidos desde su definición hasta el fin del fichero. **No son visibles** desde otras funciones ni aunque se declaren como *extern*. Ésta es una forma de **restringir la visibilidad** de las variables.

Por defecto, y por lo que respecta a su visibilidad, las funciones tienen modo *extern*. Una función puede también ser definida como *static*, y entonces sólo es visible para las funciones que están definidas después de dicha función y en el mismo fichero. Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.

6.2.6.4.Register.

Este modo es una recomendación para el compilador, con objeto de que (si es posible) ciertas variables sean almacenadas en los registros de la CPU y los cálculos con ellas sean más rápidos. No existen los modos *auto* y *register* para las funciones.

6.2.7.Conversiones de tipo implícitas y explícitas(casting).

Más adelante se comentarán las conversiones implícitas de tipo que tienen lugar cuando en una expresión se mezclan variables de distintos tipos. Por ejemplo, para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una es *int* y otra *float*, la primera se convierte a *float* (**la variable del tipo de menor rango se convierte al tipo de mayor rango**), antes de realizar la operación. A esta conversión automática e **implícita** de tipo (el programador no necesita intervenir, aunque sí conocer sus reglas), se le denomina **promoción**, pues la variable de menor rango es promocionada al rango de la otra.

Así pues, cuando dos tipos diferentes de constantes y/o variables aparecen en una misma expresión relacionadas por un operador, el compilador convierte los dos operandos al mismo tipo de acuerdo con los rangos, que de mayor a menor se ordenan del siguiente modo:

```
long double > double > float > unsigned
long > long > unsigned int > int > char
```

Otra clase de conversión implícita tiene lugar cuando el **resultado de una expresión** es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa). Por ejemplo, si i y j son variables enteras y x es *double*:

```
x = i*j – j + 1;
```

En C existe también la posibilidad de realizar conversiones explícitas de tipo (llamadas **casting**, en la literatura inglesa). El **casting** es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se

desea convertir, encerrado **entre paréntesis**. En el siguiente ejemplo:

```
k = (int) 1.7 + (int) masa;
```

La variable *masa* es convertida a tipo *int*, y la constante 1.7 (que es de tipo *double*) también. El casting se aplica con frecuencia a los valores de retorno de las funciones.

6.3.Tipos de datos fundamentales. Constantes.

Se entiende por constantes aquel tipo de información numérica o alfanumérica que **no puede cambiar** más que con una nueva compilación del programa. En el código de un programa en C pueden aparecer diversos tipos de constantes que se van a explicar a continuación.

6.3.1.Constantes numéricas.

6.3.1.1.Constantes enteras.

Una constante entera decimal está formada por una secuencia de **dígitos del 0 al 9**. Las constantes enteras decimales están sujetas a las mismas restricciones de rango que las variables tipo *int* y *long*, pudiendo también ser *unsigned*.

El tipo de una constante se puede determinar **automáticamente** según su magnitud, o de modo **explícito** posponiendo ciertos caracteres, como en los ejemplos que siguen:

```
23484 constante tipo int
45815 constante tipo long (es mayor
                      que 32767)
25u ó 25U constante tipo unsigned int
739l ó 739L constante tipo long
58ul ó 58UL constante tipo unsigned
                      long
```

En C se pueden definir también constantes enteras **octales**, esto es, expresadas en base 8 con dígitos del **0 al 7**. Se considera que una constante está expresada en base 8 si **el primer dígito por la izquierda es un cero (0)**. Análogamente, una secuencia de dígitos (del **0 al 9**) y de letras (**A, B, C, D, E, F**) **precedida por 0x o por 0X**, se interpreta como una constante entera **hexadecimal**, esto es, una constante numérica expresada en base 16. Por ejemplo:

```
011 constante octal     (igual a 9 en
                        base 10)
11 constante entera decimal (no es
                        igual a 011)
0xA constante hexadecimal (igual a 10
                        en base 10)
0xFF constante hexadecimal (igual a
                  255 en base 10)
```

La ventaja de los números expresados en base 8 y base 16 proviene de su estrecha relación con la base 2, que es la forma en la que el ordenador almacena la información.

6.3.1.2.Constantes de coma flotante.

Como es natural, existen también constantes de coma flotante, que pueden ser de tipo *float*, *double* y *long double*. Una constante de punto flotante se almacena de la misma forma que la variable correspondiente del mismo tipo. Por defecto (si no se indica otra cosa) **las constantes de punto flotante son de tipo *double***.

Para indicar que una constante es de tipo ***float*** se le añade una **f** o una **F**; para indicar que es de tipo ***long double***, se le añade una **l** o una **L**. En cualquier caso, **el punto decimal siempre debe estar presente** si se trata de representar un número real.

Estas constantes se pueden expresar de varias

formas. La más sencilla es un conjunto de dígitos del **0 al 9**, **incluyendo un punto decimal**. Para constantes muy grandes o muy pequeñas puede utilizarse la **notación científica**; en este caso la constante tiene una parte entera, un punto decimal, una parte fraccionaria, una **e** o **E**, y un exponente entero (afectando a la base 10), con un signo opcional. Se puede omitir la parte entera o la fraccionaria, pero no ambas a la vez. Las constantes de punto flotante son siempre positivas. Puede anteponerse un signo (-), pero no forma parte de la constante, sino que con ésta constituye una expresión, como se verá más adelante. A continuación se presentan algunos ejemplos válidos:

```
1.23 constante tipo double (opción por
                        defecto)
23.963f constante tipo float
.00874 constante tipo double
23e2 constante tipo double (igual a
                                2300.0)
.874e-2 constante tipo double en
                  notación científica
                  (=.00874)
.874e-2f constante tipo float en
                  notación científica
```

Seguidos de otros que **NO SON CORRECTOS:**

```
1,23 ERROR: la coma no esta permitida
23963f ERROR: no hay punto decimal ni
carácter e ó E
.e4 ERROR: no hay ni parte entera ni
fraccionaria
-3.14 ERROR: sólo el exponente puede
llevar signo
```

6.3.2.Constantes carácter.

Una constante carácter es un carácter cualquiera encerrado entre apóstrofos (tal como ***'x'*** o ***'t'***). El valor de una constante carácter es el **valor numérico asignado a ese carácter según el código ASCII**. Conviene indicar que en C no existen constantes tipo char; lo que se llama aquí constantes carácter son en realidad constantes enteras.

Hay que señalar que el valor ASCII de los números del 0 al 9 **no coincide** con el propio valor numérico. Por ejemplo, el valor ASCII de la constante carácter '7' es 55.

Ciertos caracteres no representables gráficamente, el apóstrofo (') y la barra invertida (\) y otros caracteres, se representan mediante la siguiente tabla de secuencias de escape, con ayuda de la barra invertida (\).

Nombre completo	Constante	en C	ASCII
Sonido de alerta	BEL	\a	7
Nueva línea	NL	\n	10
Tabulador horizontal	HT	\t	9
Retroceso	BS	\b	8
Retorno de carro	CR	\r	13
Salto de página	FF	\f	12
Barra invertida	\	\\	92
Apóstrofo	'	\'	39
Comillas	"	\"	34
Carácter nulo	NULL	\0	0

6.3.3.Cadenas de caracteres.

Una cadena de caracteres es una secuencia de caracteres **delimitada por comillas (")**, como por ejemplo: *"Esto es una cadena de caracteres"*. Dentro de la cadena, pueden aparecer caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter. Por ejemplo, las comillas (") deben estar precedidas por (\), para no ser interpretadas como fin de la cadena; también la propia barra invertida (\).

Es muy importante señalar que el compilador

sitúa siempre un byte **nulo (\0)** adicional **al final de cada cadena** de caracteres para señalar el final de la misma. Así, la cadena *"mesa"* no ocupa 4 bytes, sino 5 bytes. A continuación se muestran algunos ejemplos de cadenas de caracteres:

```
"Informática"
"'A'"
" cadena con espacios en blanco "
"Es una \"cadena de caracteres\".\n"
```

6.3.4. Constantes de tipo Enumeración.

En C existe una clase especial de constantes, llamadas **constantes enumeración**. Estas constantes se utilizan para definir los posibles valores de ciertos identificadores o variables que sólo deben poder tomar unos pocos valores. Por ejemplo, se puede pensar en una variable llamada *dia_de_la_semana* que sólo pueda tomar los 7 valores siguientes: *lunes, martes, miercoles, jueves, viernes, sabado y domingo*. Es muy fácil imaginar otros tipos de variables análogas, una de las cuales podría ser una variable booleana con sólo dos posibles valores: SI y NO, o TRUE y FALSE, u ON y OFF. El uso de este tipo de variables hace más claros y legibles los programas, a la par que disminuye la probabilidad de introducir errores.

En realidad, las constantes enumeración son los posibles valores de ciertas variables definidas como de ese tipo concreto. Considérese como ejemplo la siguiente declaración:

```
enum dia   {lunes, martes, miercoles,
           jueves, viernes, sabado,
           domingo};
```

Esta declaración **crea un nuevo tipo de variable** (el tipo de variable *dia*) que sólo puede tomar uno de los 7 valores encerrados entre las llaves. Estos valores son en realidad **constantes** tipo int: *lunes* es un *0*, *martes* es un *1*, *miercoles* es un *2*, etc. Ahora, es posible definir variables, llamadas *dia1* y *dia2*, que sean de tipo *dia*, en la forma:

```
enum dia dia1, dia2;    /* esto es C */
dia dia1, dia 2;              /*esto es
C++*/
```

A estas variables se les pueden asignar valores en la forma *dia1 = martes*; o aparecer en diversos tipos de expresiones y de sentencias que se explicarán más adelante.

Los valores enteros que se asocian con cada constante tipo enumeración pueden ser controlados por el programador. Por ejemplo, la declaración,

```
enum dia   {lunes=1, martes, miercoles,
           jueves, viernes, sabado,
           domingo};
```

Asocia un valor 1 a *lunes*, 2 a *martes*, 3 a *miercoles*, etc., mientras que la declaración,

```
 enum dia  {lunes=1, martes, miercoles,
           jueves=7, viernes, sabado,
           domingo};
```

Asocia un valor 1 a *lunes*, 2 a *martes*, 3 a *miercoles*, un 7 a *jueves*, un 8 a *viernes*, un 9 a *sabado* y un 10 a *domingo*.

Se puede también hacer la definición del tipo *enum* y la declaración de las variables en una única sentencia, en la forma:

```
enum palo  {oros, copas, espadas,
           bastos} carta1, carta2,
           carta3;
```

Donde *carta1, carta2 y carta3* son variables que sólo pueden tomar los valores *oros, copas, espadas y bastos* (equivalentes respectivamente a 0, 1, 2 y 3).

6.4.Cualificador const.

Se puede utilizar el cualificador *const* para indicar que esa variable **no puede cambiar de valor**. Si se utiliza con un array, los elementos del array no pueden cambiar de valor. Por ejemplo:

```
const int i=10;
const double x[] = {1, 2, 3, 4};
```

El lenguaje C no define lo que ocurre si en otra parte del programa o en tiempo de ejecución se intenta modificar una variable declarada como const. De ordinario se obtendrá un mensaje de error en la compilación si una variable const figura a la izquierda de un operador de asignación.

El cualificador *const* se suele utilizar cuando, por motivos de eficiencia, se pasan argumentos por referencia a funciones y no se desea que dichos argumentos sean modificados por éstas.

6.5.Tamaño de los datos en C.

Tipo de dato	Bytes	Rango
char	1	-128 a 127
unsigned char	1	0 a 255
short	2	-32768 a 32767
unsigned short	2	0 a 65535
int	4	--2147483648 a 2147483647
unsigned int	4	0 a 4294967295
long	4	-2147483648 a 2147483647
unsigned long	4	0 a 4294967296

Tipo de dato	Bytes	Rango
float	4	(de 6 a 7 dígitos) -3.4E+38 a –1.17E-38 1.17E-38 a 3.4E+38
double	8	(de 15 a 16 dígitos) -1.79E+308 a -2.22E-308 2.22E-308 a 1.79E+308
long double	12	(No estandar, unos 19 dígitos) -1.18E+4932 a –3.36E-4932 3.36E-4932 a 1.18E+4932

7.Estructuras de control de flujo.

7.1.Introducción.

En principio, las sentencias de un programa en C se ejecutan secuencialmente, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. El lenguaje C dispone de varias sentencias para modificar este flujo secuencial de la ejecución. Las más utilizadas se agrupan en dos familias: las **bifurcaciones**, que permiten elegir entre dos o más opciones según ciertas condiciones, y los **bucles**, que permiten ejecutar repetidamente un conjunto de instrucciones tantas veces como se desee, cambiando o actualizando ciertos valores.

7.2.Bifurcaciones.

7.2.1.Bifurcación simple (if).

Esta sentencia de control permite ejecutar o no una sentencia simple o compuesta según se cumpla o no una determinada condición. Esta sentencia tiene la siguiente forma general:

```
...
if (expresion)
          sentencia;
...
```

Explicación: Se evalúa *expresion*. Si el resultado es **verdadero** (true, ≠0), se ejecuta *sentencia*; si el resultado es **falso** (false , =0), se salta sentencia y se prosigue en la línea siguiente. Hay que recordar que sentencia puede ser una sentencia simple o compuesta (**bloque { ... }**).

Ejemplos:

```
...
/*Imprime el mayor de dos números.*/
mayor=b;
if (a>b)
    mayor=a;
printf("el mayor es %d",mayor);
...
```

```
...
/*Convierte un número de negativo a
positivo y añade 1 a la cantidad de
números negativos.*/
...
if(a<0)
{
    b=-a;
    negativo=negativo+1;
}
printf("%d sin signo es %d y llevamos"
          " %d números negativos", a,
          b, negativo);
...
```

7.2.2.Bifurcación doble (if....else).

Esta sentencia permite realizar una **bifurcación**, ejecutando una parte u otra del programa según se cumpla o no una cierta condición. La forma general es la siguiente:

```
if (expresion)
    sentencia_1;
else
    sentencia_2;
…
```

Explicación: Se evalúa *expresion*. Si el resul-

tado es **verdadero**(true, ▪ ≠0), se ejecuta *sentencia_1* y se prosigue en la línea siguiente a *sentencia_2*; si el resultado es **falso** (false, =0), se **salta** *sentencia_1*, se ejecuta *sentencia_2* y se prosigue en la línea siguiente. Hay que indicar aquí también que *sentencia_1* y *sentencia_2* pueden ser sentencias simples o compuestas (bloques { ... }).

Ejemplo:

```
...
if (a<1000)
    printf("el número introducido "
           "es pequeño);
else
    printf("el número introducido "
           "es grande");
...
```

7.2.3.Bifurcación múltiple (if...else if....else).

Esta sentencia permite realizar una **ramificación múltiple**, ejecutando una entre varias partes del programa según se cumpla una entre n condiciones.

La forma general es la siguiente:

```
...
if (expresion_1)
    sentencia_1;
else if (expresion_2)
           sentencia_2;
    else if (expresion_3)
           sentencia_3;
           else if (...)
...
                 else
                       sentencia_n;
...
```

Explicación: Se evalúa *expresion_1*. Si el resultado es **verdadero**, se ejecuta *sentencia_1*. Si el resultado es **falso**, se salta *sentencia_1* y se evalúa *expresion_2*. Si el resultado es **verdadero** se ejecuta *sentencia_2*, mientras que si es **falso** se evalúa *expresion_3* y así sucesivamente.

Si ninguna de las expresiones o condiciones es **verdadera** se ejecuta **expresion_n** que es la opción por defecto (puede ser la sentencia vacía, y en ese caso puede eliminarse junto con la palabra else). Todas las sentencias pueden ser simples o compuestas.

7.2.4.Sentencia switch.

La sentencia que se va a describir a continuación desarrolla una función similar a la de la sentencia if ... else con múltiples ramificaciones, aunque como se puede ver presenta también importantes diferencias.

La forma general de la sentencia switch es la siguiente:

```
...
switch (expresion)
{
    case expresion_cte_1:
           sentencia_1;
    case expresion_cte_2:
           sentencia_2;
...
    case expresion_cte_n:
           sentencia_n;
    default:
           sentencia;
}
...
```

Explicación: Se evalúa *expresion* y se considera el resultado de dicha evaluación. Si dicho resultado coincide con el **valor constante** *expresion_cte_1*, se ejecuta *sentencia_1* **seguida** de *sentencia_2*, *sentencia_3*, ..., *sentencia_n*. Si el resultado coincide con el valor constante *expresion_cte_2*, se ejecuta *sentencia_2* **seguida** de *sentencia_3*, ..., *sentencia_n*. En general, se ejecutan todas aquellas sentencias que están a **continuación** de la *expresion_cte* cuyo valor coincide con el

resultado calculado al principio. Si ninguna **expresion_cte** coincide se ejecuta la sentencia que está a continuación de *default*.

Si se desea ejecutar únicamente una *sentencia_i* (y no todo un conjunto de ellas), basta poner una sentencia ***break*** a continuación (en algunos casos puede utilizarse la sentencia *return* o la función *exit()*). El efecto de la sentencia *break* es dar por terminada la ejecución de la sentencia *switch*.

Existe también la posibilidad de ejecutar la misma *sentencia_i* para varios valores del resultado de *expresion*, poniendo varios *case expresion_cte* seguidos.

El siguiente ejemplo ilustra las posibilidades citadas:

```
...
switch (expresion)
{
    case expresion_cte_1:
           sentencia_1;
           break;
    case expresion_cte_2:
    case expresion_cte_3:
           sentencia_2;
           break;
    default:
           sentencia_3;
}
...
```

7.2.5.Bifurcaciones anidadas.

Una sentencia *if* puede incluir otros *if* dentro de la parte correspondiente a su sentencia, A estas sentencias se les llama **sentencias anidadas** (una dentro de otra), por ejemplo:

```
...
if (a >= b)
    if (b != 0.0)
           c = a/b;
...
```

En ocasiones pueden aparecer dificultades de interpretación con sentencias *if...else* anidadas, como en el caso siguiente:

```
...
if (a >= b)
    if (b != 0.0)
           c = a/b;
    else
           c = 0.0;
...
```

En principio se podría plantear la duda de a cuál de los dos *if* corresponde la parte *else* del programa. Los espacios en blanco (las **indentaciones** de las líneas) parecen indicar que la sentencia que sigue a *else* corresponde al segundo de los *if*, y así es en realidad, pues la regla es que **el *else* pertenece al *if* más cercano**. Sin embargo, no se olvide que el compilador de C no considera los espacios en blanco (aunque sea muy conveniente introducirlos para hacer más claro y legible el programa), y que si se quisiera que el *else* perteneciera al primero de los *if* no bastaría cambiar los espacios en blanco, sino que habría que utilizar llaves, en la forma:

```
...
if (a >= b)
{
    if (b != 0.0)
           c = a/b;
}
else
    c = 0.0;
...
```

7.2.6.Operador condicional (¿... : ...).

El operador condicional es un operador con tres operandos (**ternario**) que realiza una función parecida a *if.....else* y que tiene la siguiente forma general:

```
expresion_1 ? expresion_2: expresion_3;
```

Explicación: Se evalúa *expresion_1*. Si el resultado de dicha evaluación es **verdadero** se ejecuta *expresion_2*; si el resultado es **falso**, se ejecuta *expresion_3*.

El siguiente ejemplo aclara el uso de este operador:

	Equivale a:

mayor=(a>b)?a:b;	if(a>b)
	mayor=a;
	else
	mayor=b;

7.3.Bucles.

Además de bifurcaciones, en el lenguaje C existen también varias sentencias que permiten **repetir** una serie de veces la ejecución de unas líneas de código. Esta repetición se realiza, bien un **número determinado de veces**, bien hasta que se cumpla una determinada **condición** de tipo lógico o aritmético.

De modo genérico, a estas sentencias se les denomina **bucles**. Las tres construcciones del lenguaje C para realizar bucles son el *while*, el *for* y el *do...while*.

7.3.1.Bucle mientras (while).

Esta sentencia permite ejecutar repetidamente una sentencia o bloque de sentencias, **mientras** se cumpla una determinada condición. La forma general es como sigue:

```
...
while ( expresion_de_control)
    sentencia;
...
```

Explicación: Se evalúa *expresion_de_control* y si el resultado es **falso** se salta sentencia y se prosigue la ejecución. Si el resultado es **verdadero** se ejecuta *sentencia* y se **vuelve** a evaluar *expresion_de_control*.

Evidentemente alguna variable de las que intervienen en *expresion_de_control* habrá tenido que ser modificada, pues si no el bucle continuaría indefinidamente.

En otras palabras, *sentencia* se ejecuta repetidamente **mientras** *expresion_de_control* sea verdadera, y se deja de ejecutar cuando *expresion_de_control* se hace falsa.

Obsérvese que en este caso el control para decidir si se sale o no del bucle está *antes* de *sentencia*, por lo que es posible que sentencia no se llegue a ejecutar ni una sola vez.

7.3.2.Bucle mientras (do.....while). (parecido a hasta)

Esta sentencia funciona de modo análogo a *while*, con la diferencia de que la **evaluación** de *expresion_de_control* se **realiza al final** del bucle, después de haber ejecutado **al menos una vez** la sentencia; éstas se vuelven a ejecutar **mientras** *expresion_de_control* sea verdadera. La forma general de esta sentencia es:

```
...
do
    sentencia;
while( expresion_de_control);
...
```

Donde *sentencia* puede ser una única sentencia o un bloque, y en la que debe observarse que hay que poner (**;**) a continuación del paréntesis que encierra a *expresion_de_control*, entre otros motivos para que esa línea se distinga de una sentencia *while* ordinaria.

7.3.3.Bucle para (for).

Este es quizás el tipo de bucle más versátil y utilizado del lenguaje C. Su forma general es la siguiente:

```
...
for (inicia;control;actualizacion)
    sentencia;
...
```

Posiblemente la forma más sencilla de explicar la sentencia *for* sea utilizando la construcción *while* que sería equivalente. Dicha construcción es la siguiente:

```
...
inicia;
while (control)
 {
    sentencia;
    actualizacion;
}
...
```

Donde *sentencia* puede ser una única sentencia terminada con (N), otra sentencia de control ocupando varias líneas (*if, while, for*, ...), o una sentencia compuesta o un bloque encerrado entre llaves {...}.

Antes de iniciarse el bucle se ejecuta **inicia**, que es una o más sentencias que **asignan valores iniciales** a ciertas variables o contadores. A continuación se evalúa control y si es **falsa** se prosigue en la sentencia siguiente a la construcción *for*; si es **verdadera** se ejecutan *sentencia* y *actualizacion*, y se **vuelve a evaluar** expresion_de_control.

El proceso prosigue hasta que control sea falsa.

La parte de *actualizacion* sirve para actualizar variables o incrementar contadores.

Un ejemplo típico puede ser la recogida de 10 números introducidos por teclado e impresión de la suma de todos ellos:

```
...
for (i=0, suma=0; i<10; i++)
{
    scanf("%d",&numero);
    suma=suma+numero;
}
printf("La suma es %d",suma);
...
```

Primeramente se **inicializan** las variables *i* y *suma* a cero; el ciclo se repetirá **mientras** que *i* sea menor que 10, y al **final** de cada ciclo el valor de *i* se **incrementará** en una unidad. En total, el bucle se repetirá 10 veces.

La ventaja de la construcción *for* sobre la construcción *while* equivalente está en que en la cabecera de la construcción *for* se tiene toda la información sobre como se inicializan, controlan y actualizan las variables del bucle.

Obsérvese que la inicializacion consta de dos sentencias separadas por el operador (,).

7.4.Sentencias de control de flujo.

7.4.1.Sentencia break.

La instrucción *break* **interrumpe** la ejecución del bucle donde se ha incluido, haciendo al programa salir de él aunque la expresión de control correspondiente a ese bucle sea verdadera.

7.4.2.Sentencia continue.

La sentencia *continue* hace que el programa **comience el siguiente ciclo** del bucle donde se halla, aunque no haya llegado al final de la sentencia compuesta o bloque.

7.4.3.Sentencia goto.

La sentencia *goto* hace saltar **incindicionalmente,** al programa, a la sentencia donde se haya escrito la etiqueta correspondiente. Por ejemplo:

```
...
if (condicion)
    goto otro_lugar; /* salto al lugar
                indicado por la
                etiqueta*/
sentencia_1;
sentencia_2;
...
otro_lugar:      /* esta es la
                sentencia a la que
                se salta*/
sentencia_3;
...
```

Obsérvese que la etiqueta termina con el **carácter (:)**. La sentencia *goto* no es una sentencia muy prestigiada en el mundo de los programadores de C, pues **disminuye la claridad** y legibilidad del código. Fue introducida en el lenguaje por motivos de compatibilidad con antiguos hábitos de programación, y **siempre puede ser sustituida** por otras construcciones más claras y estructuradas.

7.5Ejemplos.

Sentencia *if*:

```
/* hacefrio.c -- porcentaje de días bajo cero */
#include <stdio.h>
#define ESCALA "Celsius"
#define CONGELA 0
int main(void)
{
    float temperatura;
    int congelado = 0;
    int dias = 0;

    printf("Introduzca la lista de temperaturas mínimas diarias.\n");
    printf("Use grados %s, y pulse s para terminar.\n", ESCALA);
    while (scanf("%f", &temperatura) == 1)
    {
           dias++;
           if (temperatura < CONGELA)
                  congelado++;
    }
    if (dias != 0)
           printf("%d días en total: %.1f%% bajo cero.\n",
                  dias, 100.0 * (float) congelado / dias);
    if (dias == 0)
           printf("¡No se han introducido datos!\n");
    return 0;       /* programa terminado correctamente */
}
```

Parte de lo anterior con *if.....else*:

```
...
if (dias != 0)
    printf("%d días en total: %.1f%% bajo cero.\n",
                        dias, 100.0 * (float) congelado / dias);
else printf("¡No se han introducido datos!\n");
return 0;       /* programa terminado correctamente */
...
```

Más *if*.

```
/* divisores.c -- calcula divisores con if anidados */
#include <stdio.h>
#define NO 0
#define SI 1
int main(void)
{
    long num;           /* número a analizar     */
    long div;           /* divisores potenciales */
    int primo;

    printf("Cálculo de los divisores de un número.\n");
    printf("Introduzca el número deseado; ");
    printf("pulse s para salir.\n");
    while (scanf("%ld", &num) == 1)
    {
           for (div = 2, primo = SI; (div * div) <= num; div++)
           {
                  if (num % div == 0)
                  {
                         if ((div * div) != num)
                                printf("%ld es divisible por %ld y %ld.\n",
                                              num, div, num/div);
                         else
                                printf("%ld es divisible por %ld.\n", num, div);
                         primo = NO;/*el número no es primo*/
                  }
           }
           if (primo == SI)
                  printf("%ld es primo.\n", num);
           printf("Introduzca otro número para analizar; ");
           printf("pulse s para salir.\n");
    }
    return 0;
}
```

Elección múltiple *else if*.

```
/*/* animales-else-if.c -- ejemplo de sentencia else if */
#include <stdio.h>
int main(void)
{
    char ch;
    printf("Déme una letra y responderé, con ");
    printf("un nombre de animal\nque comience por ella.\n");
    printf("Pulse una letra; para terminar pulse #.\n");
    while((ch = getchar()) != '#')
    {
          if (ch >= 'a' && ch <= 'z') /* sólo minúsculas */
                if (ch=='a')
                      printf("aranillo, oveja salvaje del Caribe\n");
                else if (ch=='b')
                      printf("babirusa, cerdo arlequinado de Malasia\n");
                     else if (ch=='c')
                            printf("chascalote, ballena gigante del Nepal\n");
                      else if (ch=='d')
                            printf("destemplat, pingüino rojo de Kenia\n");
                            else if (ch=='e')
                                  printf("equigobo, camello siberiano\n");
                                  else
                                        printf("Humm....,ése no me lo sé.\n");
          else
                printf("Sólo me trato con letras minúsculas.\n");
          while (getchar() != '\n')
                continue;           /* salta carácter nueva línea */
          printf("Introduzca otra letra o un #.\n");
    }           /* fin del while */
    return 0;
}
```

Este sistema no es muy “elegante” para las bifurcaciones múltiples es mucho más apropiada la sentencia switch, como se muestra a continuación.

Elección múltiple *swich*:

```
/* animales.c -- ejemplo de sentencia switch */
#include <stdio.h>
int main(void)
{
    char ch;
    printf("Déme una letra y responderé, con ");
    printf("un nombre de animal\nque comience por ella.\n");
    printf("Pulse una letra; para terminar pulse #.\n");
    while((ch = getchar()) != '#')
    {
          if (ch >= 'a' && ch <= 'z') /* sólo minúsculas */
                switch (ch)
                {
                      case 'a' :
                            printf("aranillo, oveja salvaje del Caribe\n");
                            break;
                      case 'b' :
                            printf("babirusa, cerdo arlequinado de Malasia\n");
                            break;
                      case 'c' :
                            printf("chascalote, ballena gigante del Nepal\n");
                            break;
                      case 'd' :
                            printf("destemplat, pingüino rojo de Kenia\n");
                            break;
                      case 'e' :
                            printf("equigobo, camello siberiano\n");
                            break;
                      default :
                      printf("Humm....,ése no me lo sé.\n");
                }           /* fin del switch */
          else
                printf("Sólo me trato con letras minúsculas.\n");
          while (getchar() != '\n')
                continue;          /* salta carácter nueva línea */
          printf("Introduzca otra letra o un #.\n");
    }           /* fin del while */
    return 0;
}
```

Bucle *while*.

```
/* sumas.c -- suma enteros de forma interactiva */
#include <stdio.h>
int main(void)
{
    long num;
    long suma = 0L;           /* inicializa suma a cero     */
    int estado;

    printf("Introduzca un entero a sumar. ");
    printf("Pulse s para salir.\n");
    estado = scanf("%ld", &num);     /*scanf() vale 1 si recoje un número*/
    while (estado == 1)  /* == significa "son iguales" */
    {
           suma = suma + num;
           printf("Introduzca el siguiente. ");
           printf("Pulse s para salir.\n");
           estado = scanf("%ld", &num);
    }
    printf("La suma de estos enteros es %ld.\n", suma);
    return 0;
}
```

Bucle más “C”: Nos permite sustituir las dos llamadas a la función *scanf()* por una sola, y a su vez dejar al código más estructurado.

```
...
while(scanf(“%ld”,&num)==1)   /*mientras recojamos correctamente el número*/
    {
           /*acciones del bucle*/
    }
...
```

Cuando un bucle finaliza:

```
/* cuando.c -- cuando se termina un bucle */
#include <stdio.h>
int main(void)
{
    int n = 5;
    while (n < 7)
    {
           printf("n = %d\n", n);
           n++;
           printf("Ahora n = %d\n", n);
    }
    return 0;
}
```

Salida del programa anterior:

```
n = 5
Ahora n = 6
n = 6
Ahora n = 7
```

Detalles de sintaxis:

```
/* while1.c -- ¿donde están las llaves? */
#include <stdio.h>
int main(void)
{
    int n = 0;
    while (n < 3)
           printf("n es %d\n", n);
           n++;
    printf("Eso es todo lo que sabemos hacer\n");
    return 0;
}
```

Otro:

```
/* while2.c -- ¿y los puntos y coma? */
#include <stdio.h>
int main(void)
{
    int n = 0;
    while (n++ < 3);
           printf("n es %d\n", n);
    printf("Eso es todo lo que sabemos hacer\n");
    return 0;
}
```

¿Qué es más verdad?

```
/* verdad.c -- ¿què valores son ciertos? */
#include <stdio.h>
int main(void)
{
    int n = 3;
    while (n)
           printf("%d\n", n--);
    n = -3;
    while (n)
           printf("%2d\n", n++);
    return 0;
}
```

Problemas con la verdad:

```
/* problema.c -- uso erróneo de = */
#include <stdio.h>
int main(void)
{
    long num;
    long suma = 0L;
    int estado;
    printf("Introduzca un entero a sumar. ");
    printf("Pulse s para salir.\n");
    estado = scanf("%ld", &num);
    while (estado = 1)
    {
           suma = suma + num;
           printf("Introduzca el siguiente. ");
           printf("Pulse s para salir.\n");
           estado = scanf("%ld", &num);
    }
    printf("La suma de estos enteros es %ld.\n", suma);
    return 0;
}
```

Bucle con contador:

```
/* suerte1.c -- bucle contador */
#include <stdio.h>
#define NUMERO 22
int main(void)
{
    int cont = 1;              /* inicialización  */
    while (cont <= NUMERO)           /* test            */
    {
          printf("¡Buena suerte!\n");            /* acción          */
          cont++;                           /* incremento cont */
    }
    return 0;
}
```

Bucle con condición de salida *do....while*:

```
/* dowhile.c -- bucle con condición de salida */
#include <stdio.h>
int main(void)
{
    char ch;
    do
    {
          scanf("%c", &ch);
          printf("%c", ch);

    } while (ch != '#');
    return 0;
}
```

Bucle *for*:

```
/* suerte2.c -- bucle contador que utiliza for */
#include <stdio.h>
#define NUMERO 22
int main(void)
{
    int cont;
                        /*Todos los datos están en esta línea*/
    for (cont = 1; cont <= NUMERO; cont++)
          printf("¡Buena suerte!\n");
    return 0;
}
```

Flexibilidad del bucle *for*:

```
1) Sentido descendente:              for (seg=5; seg>0; seg--){.....}
2) Contar con incrementos:           for (n=5; n<60; n+=5){.....}
3) Contar con caracteres:            for (ch='a'; ch<=z; ch++){.....}
4) Condiciones distintas:            for (n=1; p*n<234; n++){.....}
5) Incremento geométrico:            for (deuda=100.0; deuda<150.0; deuda*=1.1){.....}
6) Cualquier expresión válida en "c":      for (x=1; y<=75; y=(++x*5)+50){.....}
7) Expresiones en blanco:            for (n=5; ans<=25;){....}        for(;;){.....}
8) No es necesario iniciar una variable:
                                     for(printf("Empiece a meter números\n"); num!=6;)
                                           scanf("%d", &num);
                                     printf("Este es el que yo quería");
9) Se puede alterar los parámetros dentro del bucle:
                                     for (n=1; n<minimo; n+=incremento){.....}
etc.
```

Zenón encuentra el bucle *for*:

```
/* zenon.c -- suma de una serie */
#include <stdio.h>
#define LIMITE 15
int main(void)
{
    int cont;
    float suma, x;
    for (suma=0.0, x=1.0/2.0, cont=1; cont <= LIMITE; cont++, x /= 2.0)
    {
           suma += x;
           printf("suma = %f en la etapa %d.\n", suma, cont);
    }
    return 0;
}
```

break:

```
/* break.c -- usa break para salir del bucle */
#include <stdio.h>
int main(void)
{
    float largo, ancho;
    printf("Calculo el área de un rectángulo.\n");
    printf("Para salir introducir una letra en la anchura.\n");
    printf("Longitud del rectángulo:\n");
    while (scanf("%f", &largo) == 1)
    {
           printf("Longitud = %0.2f:\n", largo);
           printf("Anchura del rectángulo\n");
           if (scanf("%f", &ancho) != 1)
                  break;
           printf("Anchura = %0.2f:\n", ancho);
           printf("Area = %0.2f:\n", largo * ancho);
           printf("Longitud del rectángulo:\n");
    }
    return 0;
}
```

continue:

```
/* salta.c -- usa continue para saltar parte del bucle */
#include <stdio.h>
#define MIN 0.0
#define MAX 100.0

int main(void)
{
    float puntos;
    float total = 0.0;
    int n = 0;
    float min = MAX;
    float max = MIN;

    printf("Calcula el valor máximo, mínimo y la media"
                " de una serie de números comprendidos entre 0 y 100.\n"
                "Introduce una letra para terminar.\n");
    printf("Introduzca puntuaciones:\n");
    while (scanf("%f", &puntos) == 1)
    {
           if (puntos < MIN || puntos > MAX)
           {
                  printf("%0.1f no es un valor válido.\n", puntos);
                  continue;
           }
           printf("Se acepta %0.1f:\n", puntos);
           min = (puntos < min)? puntos: min;
           max = (puntos > max)? puntos: max;
           total += puntos;
           n++;
    }
    if (n > 0)
    {
           printf("El promedio de %d valores es %0.1f.\n", n, total/n);
           printf("Mínimo = %0.1f, máximo = %0.1f\n", min, max);
    }
    else
           printf("No se han indicado valores válidos.\n");
    return 0;
 }
```

7.6.Ejercicios.

a) Realiza un programa que lea dos números del teclado y, mediante la sentencia *if...else*, muestre el mayor de dos números.

b) Haz un programa que, mediante la sentencia *switch*, presente en pantalla un sencillo menú con al menos cinco opciones.

c) Diseña un programa que muestre los números del 1 al 100 utilizando un bucle *while*.

d) Haz el mismo programa anterior, pero con un bucle *for*.

e) Construye un programa que muestre los números pares, comprendidos entre 1 y 100 y que, mediante *break* pare el bucle en un número previamente introducido por el teclado.

f) Realizar un programa con un bucle infinito para presentar permanentemente un menú que incluirá una opción de salida haciendo uso de la sentencia *return*.

g) Haz un programa que calcule la raíz cuadrada de un número por el método interactivo de Newton. (ri=((n/ri)+ri)/2, test=ri*ri, repetir mientras n>test y n-test>=precisión o n<test y test-n>=precisión).

h) Realizar un programa que mediante la sentencia *switch*, indique el número de días que tiene el mes introducido por teclado, representado por los números entre 1 y 12.

i) Desarrollar un programa que realice la suma, resta y producto de dos números enteros y permita la selección de la operación mediante un menú.

j) Elabora un programa que muestre en pantalla un "triángulo mágico" como el que se muestra a continuación, y que permita seleccionar el número de líneas.

```
    1
   121
  12321
 1234321
123454321
```

k) Haz un programa que muestre en pantalla una onda senoidal, una onda triangular y una cuadrada.

l) Juego de adivinar el número generado por el ordenador. Es necesario utilizar las funciones *srand()* y *rand()*, cuyo prototipo se encuentra en *stdlib.h*, la función *time()* que se define en *time.h*.

8.Entrada y salida estándar.

A diferencia de otros lenguajes, C no dispone de sentencias de entrada/salida. En su lugar se utilizan funciones contenidas en la **librería** estándar y que forman parte integrante del lenguaje.

Las funciones de entrada/salida (Input/Output) son un conjunto de **funciones**, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior.

Trabajan con el dispositivo estándar de E/S, que es la consola: el **teclado** para entradas y el **monitor** para salidas. En realidad las E/S están asociadas a tres ficheros:

- Fichero estándar de entrada *stdin* asociado al teclado.
- Fichero estándar de salida *stdout* asociado al monitor.
- Fichero estándar de error *stderr* asociado también al monitor.

Para su utilización es necesario **incluir**, al comienzo del programa, el archivo *stdio.h* en el que están definidos sus prototipos:

```
#include <stdio.h>
```

Donde *stdio* proviene de **standard-input-output**.

8.1.Salida con formato (printf()).

La función *printf()* imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. La forma general de esta función se puede estudiar viendo su prototipo:

```
int printf ("cadena_de_control", arg1,
           arg2, ...)
```

La función *printf()* imprime el texto contenido en *cadena_de_control* junto con el **valor** de los otros argumentos, de acuerdo con los formatos incluidos en *cadena_de_control*. Los puntos suspensivos (...) indican que puede haber un número **variable** de argumentos. Cada formato comienza con el carácter (**%**) y termina con un carácter de conversión. Considérese el ejemplo siguiente:

```
...
int i;
double tiempo;
float masa;
printf("Resultado nº: %d. En el"
    " instante %lf la masa vale %f\n",
    i, tiempo, masa);
...
```

En el que se imprimen tres **variables** (*i*, *tiempo* y *masa*) con los formatos (*%d*, *%lf* y *%f*), correspondientes a los tipos (*int*, *double* y *float*), respectivamente. La cadena de control se imprime con el valor de cada variable *intercalado* en el lugar del formato correspondiente.

Especificadores de formato:

%d	Enteros con signo (*int*)
%u	Enteros sin signo (*unsigned int*)
%o	Enteros sin signo en octal (*unsigned int*)
%x	Enteros sin signo en hexadecimal (*unsigned int*) con caracteres **0...f** .
%X	Enteros sin signo en hexadecimal (*unsigned int*) con caracteres **0...F** .
%f	Real (*float*, *double*) en formato **[-]ddd.ddd**
%e	Real (*float*, *double*) en formato **[-]ddd.ddde[+/-]ddd**
%E	Real (*float*, *double*) en formato **[-]ddd.dddE[+/-]ddd**
%g	Real (*float*, *double*) %f o %e el que sea más corto.

%G	Real (*float*, *double*) %f o %E el que sea más corto.
%c	Carácter (*char*)
%s	Cadena de caracteres
%p	Puntero

Lo importante es considerar que debe haber correspondencia uno a uno entre los **formatos** que aparecen en la *cadena_de_control* y los otros **argumentos** (constantes, variables o expresiones).

Entre el carácter % y el carácter de conversión puede haber, por el siguiente orden, uno o varios de los elementos que a continuación se indican:

% [flags] [anchura] [.precisión] [prefijo_tipo] formato

Las opciones más usuales son las siguientes (Hay más):

- **flags**: permite cambiar la **justificación** de la representación y actuar o no sobre la presentación del **signo**.
  - **Un signo (-)**, que indica alineamiento por la **izquierda** (el defecto es por la dcha),
  - E**l signo (+)** antepone el signo al número.
  - **Un cero (0)** completa el número, con ceros a la izquierda hasta el ancho total.
  - **Un espacio en blanco** pone un espacio al principio si el número es positivo.
- **anchura**: permite fijar el número de posiciones que se **reservarán** en pantalla para la presentación del dato de salida.
- **.precisión**: depende de el dato representado:
  - En los números **enteros** determina el número de dígitos mínimos de la representación.
  - En una **cadena** de caracteres, el nº máximo de caracteres a imprimir.
  - En **reales** el nº de decimales.
- **prefijo_tipo**: es un cualificador del tipo a representar.
  - una (h) para *short*.
  - una (l) para *long* en enteros o *double* en los reales.
  - una (L) para *long double*.
- **formato**: es un campo obligatorio que especifica el tipo de dato según la tabla anterior.

Además de lo anterior se pueden utilizar secuencias especiales (**secuencias de escape**) para representar códigos ASCII sin representación simbólica. Los más utilizados son:

\a	Alarma
\’	Comilla simple
\b	Espacio atrás
\”	Comilla doble
\f	Salto de página
\\	Barra invertida
\n	Nueva línea
\00	Carácter ASCII en octal
\r	Retorno de carro
\xHH	Carácter ASCII en hexadecimal
\t	Tabulador horizontal
\0	Carácter nulo

A continuación se incluyen algunos ejemplos de uso de la función *printf()*. El primer ejemplo contiene sólo texto, por lo que basta con considerar la cadena_de_control.

```
printf("Con cien cañones por”
    “ banda,\nviento en popa a toda”
    “ vela,\n");
```

El resultado serán **dos líneas** con las dos primeras estrofas de la famosa poesía. No es posible partir *cadena_de_control* en varias líneas con caracteres intro, por lo que en este ejemplo se observa que se ha cortado la cadena con **comillas dobles (“)**. Una forma alternativa, muy sencilla, clara y ordenada, de escribir la poesía sería la siguiente:

```
printf("%s\n%s\n%s\n%s\n",
"Con cien cañones por banda,",
"viento en popa a toda vela,",
"no cruza el mar sino vuela,",
"un velero bergantín.");
```

En este caso se están escribiendo **cuatro cadenas** constantes de caracteres que se introducen como argumentos, con formato ***%s*** y con los correspondientes saltos de línea.

Un ejemplo que contiene una constante y una variable como argumentos es el siguiente:

```
printf("En el año %s ganó %ld ptas.\n",
          "1993", beneficios);
```

Donde el texto **1993** se imprime como cadena de caracteres (***%s***), mientras que beneficios se imprime con formato de variable *long* (***%ld***).

Con el siguiente ejemplo se puede apreciar el efecto que producen los modificadores del formato:

```
printf("/%010.2f/\n", 1.234560e+003);
```

La salida en este caso sería:

```
 /0001234.56/.
```

Es importante hacer corresponder bien los formatos con el tipo de los argumentos, pues si no los resultados pueden ser muy diferentes de lo esperado.

La función *printf()* tiene un **valor de retorno** de tipo *int*, que representa el número de caracteres escritos en esa llamada.

8.2.Entrada con formato (scanf()).

La función *scanf()* es análoga en muchos aspectos a *printf()*, y se utiliza para **leer datos** de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

```
int scanf("%x1%x2...", &arg1,
&arg2, ...);
```

Donde x1, x2, ... son los caracteres de conversión, mostrados en la tabla siguiente, que representan los formatos con los que se espera encontrar los datos.

%d o %i	Enteros con signo en decimal (*int*)
%u	Enteros sin signo en decimal (*unsigned int*)
%o	Enteros sin signo en octal (*unsigned int*)
%x o %X	Enteros sin signo en hexadecimal (*unsigned int*) 0-f
%f, %e, %E, %g, %G	Real (*float, double*)
%c	Carácter (*char*)
%s	Cadena de caracteres
h, l	Modificador para *short*, *long* y *double*
L	modificador para *long double*

La función *scanf()* devuelve como **valor de retorno** el número de conversiones de formato realizadas con éxito.

La cadena de control de *scanf()* puede contener caracteres además de formatos. Dichos caracteres se utilizan para tratar de **detectar la presencia de caracteres** idénticos en la entrada por teclado.

Si lo que se desea es leer variables numéricas, esta posibilidad tiene escaso interés.

En la función *scanf()* los argumentos que siguen a la *cadena_de_control* deben ser pasados por **referencia**, ya que la función los lee y tiene que trasmitirlos al programa que la ha llamado. Para ello, dichos argumentos deben estar constituidos por las **direcciones de las variables** en las que hay que depositar los datos, y no por las propias variables. Una **excepción** son las cadenas de caracteres, cuyo nombre es ya de por sí una **dirección** (un puntero), y por tanto no debe ir precedido por el **operador (&)** en la llamada.

Por ejemplo, para leer los valores de dos variables *int* y *double* y de una cadena de caracteres, se utilizarían la sentencia:

```
int n;
double distancia;
char nombre[20];
scanf("%d%lf%s", &n, &distancia,
nombre);
```

En la que se establece una correspondencia entre *n* y *%d*, entre *distancia* y *%lf*, y entre *nombre* y *%s*. Obsérvese que *nombre* no va precedido por el operador (*&*). La lectura de cadenas de caracteres se **detiene en cuanto se encuentra un espacio en blanco**, por lo que para leer una línea completa con varias palabras hay que utilizar otras técnicas diferentes.

En los formatos de la cadena de control de *scanf()* pueden introducirse corchetes *[...]*, que se utilizan como sigue.

```
scanf("%[AB \t]", s);   /* se leen solo
                        los caracteres
                        indicados*/
```

Lee caracteres hasta que encuentra uno **diferente** de ('*A*','*B*',' ','\t'). En otras palabras, se leen sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se detiene la lectura y se devuelve el control al programa que llamó a *scanf()*.

Si los corchetes contienen un carácter **(^)**, se leen todos los caracteres **distintos** de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Por ejemplo:

```
scanf(" %[^\n]", s);
```

Lee todos los caracteres que encuentra hasta que llega al carácter nueva línea '\n'. Esta sentencia puede utilizarse por tanto para leer líneas completas, con **blancos incluidos**.

Recuérdese que con el formato *%s* la lectura se detiene al llegar al primer delimitador (carácter blanco, tabulador o nueva línea).

8.2.Macros de entrada/salida de caracteres.

Una macro representa una sustitución de texto que se realiza antes de la compilación por medio del preprocesador. Para casi todos los efectos, estas macros pueden ser consideradas como funciones. Más adelante se explicarán las macros con algunos ejemplos.

Las macros *getchar()* y *putchar()* permiten respectivamente leer e imprimir un sólo carácter cada vez, en la entrada o en la salida estándar.

La macro *getchar()* recoge un carácter introducido por teclado y lo deja disponible como valor de retorno. La macro *putchar()* escribe en la pantalla el carácter que se le pasa como argumento.

```
/*Escribe el carácter a.*/

putchar('a');

/*Equivale a: printf("a");*/
/*Lee un carácter del teclado*/

c = getchar();

/*Equivale a: scanf("%c", &c);*/
```

Estas macros están definidas en el fichero *stdio.h*, y su código es sustituido en el programa por

el preprocesador antes de la compilación.

Por ejemplo, se puede leer una línea de texto completa utilizando *getchar()*:

```
...
int i=0, c;
char name[100];
while((c = getchar()) != '\n') /* se
leen
          caracteres hasta el '\n'*/
    name[i++] = c; /* se almacena el
          carácter en Name[]*/
name[i]='\0'; /* se añade el carácter
          fin de cadena*/
...
```

8.3.Lectura y escritura de cadenas.

Una **cadena** de caracteres es una secuencia de caracteres delimitada por comillas ("), como por ejemplo: *"Esto es una cadena de caracteres"*.

Dentro de la cadena, pueden aparecer caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter. Por ejemplo, las comillas (") deben estar precedidas por (\), para no ser interpretadas como fin de la cadena; también la propia barra invertida (\).

8.3.1.Lectura de cadenas.

Con la función *scanf()* es posible leer cadenas con el especificador de formato *%s*, con la particularidad de que la función leerá todos los caracteres hasta el primer carácter separador (**espacio, tabulador o INTRO**). La expresión sería:

```
scanf("%s",cadena);
```

CUIDADO leerá todos los caracteres hasta el primer espacio, que no es leído y **se quedará** en *stdin*, y será leído por la siguiente llamada a *scanf()*.

Más apropiada para la lectura de cadenas es la función ***gets()***, que lee del teclado caracteres hasta la pulsación de la tecla INTRO y los almacena en el array de caracteres cuyo nombre se pasa como argumento, sustituyendo el INTRO por '\0'. Se utiliza normalmente de la siguiente forma:

```
gets(nombre_de_la_cadena);
```

No se hace ninguna comprobación de desbordamiento del búfer, por lo que hay que tener la precaución de que el tamaño de la cadena sea suficiente para contener los datos, más el carácter nulo del final.

8.3.2.Escritura de cadenas.

La función *printf()* permite la escritura de cadenas de caracteres por distintos métodos:

Puede incluir la cadena entre comillas dobles como cadena de control:

```
printf("\nEsto es una cadena mostrada"
          " con printf().");
```

Mediante el especificador de formato *%s* y una constante:

```
printf("%s", "\nesto tambien está"
          " mostrado por printf()");
```

Mediante el especificador *%s* y un nombre de variable:

```
char mensa[]="\Otro más con printf()";
printf("%s",mensa);
```

No obstante es más apropiado utilizar la función ***puts()*** que recibe como argumento el nombre del array y muestra en pantalla todos los elementos de la cadena, cambiando el carácter nulo de terminación de la cadena por '**\n**'. Se utiliza de la forma siguiente:

```
puts(nombre_de_la_cadena);
puts("\nEsto también funciona");
```

8.4.Ejemplos de printf().

```
/* imprime.c -- ejemplo de especificadores de conversión */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
    int numero = 5;
    float ron = 13.5;
    int coste = 3100;
    printf("Las %d mujeres se bebieron %f vasos de ron.\n",numero, ron);
    printf("El valor de pi es %f.\n", PI);
    printf("Fazer non quiso que tal malandrín fablara.\n");
    printf("%c%d\n",'$', 2 * coste);
    return 0;
}
```

```
/* anchura.c -- anchuras de los campos
*/
#include <stdio.h>
#define PAGINAS 336
int main(void)
{
    printf("/%d/\n", PAGINAS);
    printf("/%2d/\n", PAGINAS);
    printf("/%10d/\n", PAGINAS);
    printf("/%-10d/\n", PAGINAS);
    return 0;
}
```

La salida será:

```
/336/
/336/
/       336/
/336       /
```

```
/* flotante.c -- combinaciones de punto
flotante */
#include <stdio.h>
#define RENTA 1234.56
int main(void)
{
    printf("/%f/\n", RENTA);
    printf("/%e/\n", RENTA);
    printf("/%4.2f/\n", RENTA);
    printf("/%3.1f/\n", RENTA);
    printf("/%10.3f/\n", RENTA);
    printf("/%10.3e/\n", RENTA);
    printf("/%+4.2f/\n", RENTA);
    printf("/%010.2f/\n", RENTA);
    return 0;
}
```

La salida es:

```
/1234.560000/
/1.234560e+03/
/1234.56/
/1234.6/
/  1234.560/
/ 1.235e+03/
/+1234.56/
/0001234.56/
```

```
/* simbolos.c -- estudia controles de
formatos */
#include <stdio.h>
int main(void)
{
printf("%x %X %#x\n", 31, 31, 31);
printf("**%d**% d**% d**\n", 42, 42,
-42);
printf("**%5d**%5.3d**%05d**%05.3d**\n",
           6, 6, 6, 6);
return 0;
}
```

El resultado es:

```
1f 1F 0x1f
**42** 42**-42**
**    6**  006**00006**  006**
```

```
/* tiras.c -- formatos de tiras */
#include <stdio.h>
#define PINTA "¡Alucinante acción!"
int main(void)
{
    printf("/%2s/\n", PINTA);
    printf("/%22s/\n", PINTA);
    printf("/%22.5s/\n", PINTA);
    printf("/%-22.5s/\n", PINTA);
    return 0;
    }
```

El resultado será:

```
/¡Alucinante acción!/
/ ¡Alucinante acción!/
/                  ¡Alu/
/¡Alu                  /
```

```
/* prntval.c -- valor de retorno de printf() */
#include <stdio.h>
int main(void)
{
    int n = 100;
    int rv;

    rv = printf("El punto de ebullición del agua es %d grados.\n", n);
    printf("La función printf() ha impreso %d caracteres.\n", rv);
    return 0;
}
```

La salida será:

```
El punto de ebullición del agua es 100 grados.
La función printf() ha impreso 48 caracteres.
```

```
/* largo.c -- impresión de tiras largas
*/
#include <stdio.h>
int main(void)
{
printf("Una forma de imprimir una ");
printf("tira larga.\n");
printf("Otra forma de imprimir una \
tira larga.\n");
printf("La nueva forma de imprimir una"
            " tira larga.\n");/* ANSI C
*/
return 0;
}
```

El resultado será:

```
Una forma de imprimir una tira larga.
Otra forma de imprimir una tira larga.
La nueva forma de imprimir una tira larga.
```

```
/* secretos.c -- programa informativo totalmente inútil */
#include <stdio.h>
#define DENSIDAD 0.97    /* densidad del hombre en kg por litro */
int main(void)
{
    float peso, volumen;
    int sitio, letras;
    char nombre[40];
    printf("¡Hola! ¿Cómo te llamas?\n");
    scanf("%s", nombre);
    printf("%s, ¿cuál es tu peso en kg?\n", nombre);
    scanf("%f", &peso);
    sitio = sizeof nombre;
    letras = strlen(nombre);
    volumen = peso/DENSIDAD;
    printf("Bien, %s, tu volumen es %2.2f litros.\n", nombre, volumen);
    printf("Además, tu nombre tiene %d letras,\n", letras);
    printf("y disponemos de %d bytes para guardarlo.\n", sitio);
    return 0;
}
```

```
/* elogio2.c */
#include <stdio.h>
#define ELOGIO "¡Por Júpiter, qué gran nombre!"
int main(void)
{
    char nombre[50];
    printf("¿Cómo te llamas?\n");
    scanf("%s", nombre);
    printf("Hola, %s. %s\n", nombre, ELOGIO);
    printf("Tu nombre de %d letras ocupa %d celdas de memoria.\n",
    strlen(nombre), sizeof nombre);
    printf("La frase de elogio tiene %d letras ", strlen(ELOGIO));
    printf("y ocupa %d celdas de memoria.\n", sizeof ELOGIO);
    return 0;
}
```

8.5.Ejemplos de scanf().

```
/* entradas.c -- cuándo se debe usar & */
#include <stdio.h>
int main(void)
{
    int edad;
    float sueldo;
    char cachorro[30];        /* tira de caracteres */
    printf("Confiese su edad, sueldo y mascota favorita.\n");
    scanf("%d %f", &edad, &sueldo);        /* use & aquí */
    scanf("%s", cachorro);    /* en cadena de caracteres no se usa & */
    printf("Edad: %d, Sueldo: %.0f euros, Mascota: %s\n ",
                        edad, sueldo, cachorro);
    return 0;
}
```

El resultado será:

```
Confiese su edad, sueldo y mascota favorita.
98
800 Gallipato
Edad: 98, Sueldo: 800 euros, Mascota: Gallipato
```

```
/* salta2.c -- salta los dos primeros enteros de la entrada */
#include <stdio.h>
int main(void)
{
    int n;

    printf("Introduzca tres enteros:\n");
    scanf("%*d %*d %d", &n);         //El asterisco significa siguiente
    printf("El último entero leído es %d\n", n);
    return 0;
}
```

La salida será:

```
Introduzca tres enteros:
25 45 22
El último entero leído es 22
```

IMPORTANTE

Los caracteres no leídos se quedan en el Buffer y pueden producir falsas lecturas. Para limpiar el Buffer utilizar fflush(stdin) o while(getchar()!='\n');

8.6.Ejemplos entrada/salida de caracteres.

```
/* cifrado1.c -- modifica la entrada conservando los espacios */
#include <stdio.h>
#define ESPACIO ' '           /* apóstrofo espacio apóstrofo */
int main(void)
{
    char ch;
    ch = getchar();                       /* lee un carácter */
    while (ch != '\n')               /* mientras no sea fin de línea */
    {
          if (ch == ESPACIO)         /* deja carácter espacio */
                putchar(ch);         /* sin modificar */
          else
                putchar(ch + 1);     /* cambia los demás caracteres */
          ch = getchar();                 /* toma siguiente carácter */
    }
    return 0;
}
```

Resultado:

```
LLAMAME HAL
MMBNBNF IBM
```

```
/* cifrado2.c -- modifica la entrada conservado los espacios */
/* CON UN ESTILO MÁS “C” */
#include <stdio.h>
#define ESPACIO ' '                  /* apóstrofo espacio apóstrofo */
int main(void)
{
    char ch;
    while ((ch = getchar()) != '\n')
    {
          if (ch == ESPACIO)         /* deja carácter espacio */
                putchar(ch);         /* sin modificar */
          else
                putchar(ch + 1);     /* cambia los demás caracteres */
    }
    return 0;
}
```

```
/* nombre1.c -- lee un nombre */
#include <stdio.h>
#define MAX 80
int main(void)
{
    char nombre[81];           /* reserva espacio */
    printf("Hola, ¿cómo te llamas?\n");
    gets(nombre);                     /* introduce entrada en tira "nombre" */
    printf("Bonito nombre, %s.\n", nombre);
    return 0;
}
```

```
/* nombre2.c -- lee un nombre */
/*Otra manera de usar getchar*/
#include <stdio.h>
#define MAX 81
int main(void)
{
    char nombre[MAX];
    char *ptr;                //char *gets(char *)
    printf("Hola, ¿cómo te llamas?\n");
    ptr = gets(nombre);
    printf("¿%s? ¡Ah! ¡%s!\n", nombre, ptr);
    return 0;
}
```

8.7.Ejercicios.

a) Elabora un programa que convierta de pesetas a euros y viceversa, mediante un menú que disponga de la opción salir.

b) Realiza un programa que imprima el calendario de un mes, pidiendo el día de la semana de comienzo y el nombre del mes en cuestión.

c) Elabora un programa que lea un carácter del teclado y muestre en pantalla el valor numérico de su código ASCII en decimal y en hexadecimal.

d) Crea un programa que lea una cadena del teclado y a continuación la presente en pantalla.

e)Realiza un programa que presente la tabla de caracteres ASCII con su equivalente decimal. Procura que la tabla quede ordenada.

f)Diseña un programa que tome tres palabras del teclado y luego las imprima separadas por guiones.

9.Operadores, expresiones y sentencias.

9.2.Operadores.

Un operador es un carácter o grupo de caracteres **que actúa** sobre una, dos o más variables para realizar una determinada operación con un determinado resultado.

Ejemplos típicos de operadores son la suma (**+**), la diferencia (**-**), el producto (*****), etc.

Los operadores pueden ser **unarios**, **binarios** y **ternarios**, según actúen sobre uno, dos o tres operandos, respectivamente.

En C existen muchos operadores de diversos tipos (éste es uno de los puntos fuertes del lenguaje), que se verán a continuación.

9.2.1.Operadores aritméticos.

Los operadores aritméticos son los más sencillos de entender y de utilizar. Todos ellos son operadores binarios.

En C se utilizan los cinco operadores siguientes:

- Suma: **+**
- Resta: **-–**
- Multiplicación: *****
- División: **/**
- Módulo o Resto: **%**

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.

El único operador que requiere una explicación adicional es el operador **resto %**. En realidad su nombre completo es **resto de la división entera**. Este operador se aplica **solamente a constantes, variables o expresiones de tipo int**. Aclarado esto, su significado es evidente: **23%4** es **3**, puesto que **el resto** de dividir 23 por 4 es 3.

Si a%b es cero, a es múltiplo de b.

Como se verá más adelante, una expresión es un conjunto de variables y constantes (y también otras expresiones más sencillas) relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es el siguiente polinomio en la variable x:

```
5.0 + 3.0*x – x*x/2.0
```

Las expresiones pueden contener paréntesis **(...)** que agrupan a algunos de sus términos. Puede haber paréntesis contenidos dentro de otros paréntesis. El significado de los paréntesis coincide con el habitual en las expresiones matemáticas, con algunas características importantes que se verán más adelante. En ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones.

9.2.2.Operadores de asignación.

Los **operadores de asignación** atribuyen a una variable (es decir, depositan en la zona de memoria correspondiente a dicha variable) el resultado de una expresión o el valor de otra variable.

El operador de asignación más utilizado es el **operador de igualdad (=)**, que no debe ser confundido con la igualdad matemática. Su forma general es:

```
nombre_de_variable = expresion;
```

Cuyo funcionamiento es como sigue: se evalúa *expresion* y el resultado se deposita en *nombre_de_variable*, **sustituyendo cualquier otro**

valor que hubiera en esa posición de memoria anteriormente. Una posible utilización de este operador es como sigue:

```
variable = variable + 1;
```

Desde el punto de vista matemático este ejemplo no tiene sentido (¡Equivale a 0 = 1!), pero sí lo tiene considerando que en realidad el operador de asignación (=) representa una sustitución; en efecto, se toma el valor de variable contenido en la memoria, se le suma una unidad y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador variable, sustituyendo al valor que había anteriormente. El resultado ha sido incrementar el valor de variable en una unidad.

Así pues, **una variable puede aparecer a la izquierda y a la derecha del operador (=)**. Sin embargo, a la izquierda del operador de asignación (=) no puede haber nunca una expresión: tiene que ser necesariamente el nombre de una variable. Es incorrecto, por tanto, escribir algo así como:

```
a + b = c; // INCORRECTO
```

Existen otros cuatro operadores de asignación **(+=, -=, *= y /=)** formados por los 4 operadores aritméticos seguidos por el carácter de igualdad. Estos operadores simplifican algunas operaciones recurrentes sobre una misma variable. Su forma general es:

```
variable op= expresion;
```

Donde **op** representa cualquiera de los operadores (**+ - * /**).

La expresión anterior es equivalente a:

```
variable = variable op expresion;
```

A continuación se presentan algunos ejemplos con estos operadores de asignación:

```
                    //equivale a:
peso += 1;          peso = peso+1;
rango /= 2.0;       rango = rango /2.0
x *= 3.0 * y – 1.0;     x = x * (3.0 * y
– 1.0)
```

9.2.3.Operadores incrementales.

Los **operadores incrementales** (**++**) y (**--**) son **operadores unarios** que incrementan o disminuyen en una unidad el valor de la variable a la que afectan. Estos operadores pueden ir inmediatamente **delante o detrás de la variable**.

Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece.

Si es la variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión.

A continuación se presenta un ejemplo de estos operadores:

```
i = 2;
j = 2;
m = i++; /* despues de ejecutarse esta
            sentencia m=2 e i=3*/
n = ++j; /* despues de ejecutarse esta
            sentencia n=3 y j=3*/
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados m y n del ejemplo anterior son diferentes.

9.2.4.Operadores relacionales.

Una característica imprescindible de cualquier lenguaje de programación es la de **considerar alternativas**, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones.

Los **operadores relacionales** permiten estudiar si se **cumplen o no esas condiciones**. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (**yes, no**), (**on, off**), (**true, false**), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (**true, false**). Si una condición se cumple, el resultado es **true**; en caso contrario, el resultado es **false**.

En C un 0 representa la condición de false, y cualquier número distinto de 0 equivale a la condición true. Cuando el resultado de una expresión es true y hay que asignar un valor concreto distinto de cero, **por defecto se toma un valor unidad**. Los operadores relacionales de C son los siguientes:

- Igual que: ==
- Menor que: <
- Mayor que: >
- Menor o igual que: <=
- Mayor o igual que: >=
- Distinto que: !=

Todos los operadores relacionales son **operadores binarios** (tienen dos operandos), y su forma general es la siguiente:

```
expresion1 op expresion2
```

Donde **op** es uno de los operadores (**==, <, >, <=, >=, !=**). El funcionamiento de estos operadores es el siguiente: se evalúan expresion1 y expresion2, y se comparan los valores resultantes. Si la condición representada por el operador relacional **se cumple, el resultado es 1**; si la condición **no se cumple, el resultado es 0**.

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1)     /* resultado=0 porque la
              condición no se cumple*/
(3<=3)     /* resultado=1 porque la
              condición se cumple*/
(3<3)      /* resultado=0 porque la
              condición no se cumple*/
(1!=1)     /* resultado=0 porque la
              condición no se cumple*/
```

9.2.5.Operadores lógicos.

Los **operadores lógicos** son operadores **binarios** que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc.

El lenguaje C tiene dos operadores lógicos: **el operador Y (&&)** y el **operador O (||)**. En inglés son los operadores **and y or**. Su forma general es la siguiente:

```
expresion1 || expresion2
expresion1 && expresion2
```

El operador **&&** devuelve un 1 si tanto *expresion1* como *expresion2* son verdaderas (**distintas de 0**), y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0); por otra parte, el operador || devuelve 1 si al menos una de las expresiones es cierta.

Es importante tener en cuenta que los compiladores de C tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no deseados. Por ejemplo: para que el resultado del operador && sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa expresion1 y es falsa, ya no hace falta evaluar expresion2, y de hecho no se evalúa. Algo parecido pasa con el operador ||: si expresion1 es verdadera, ya no hace falta evaluar expresion2.

Los operadores && y || se pueden combinar entre sí (quizás agrupados entre paréntesis), dando a veces un código de más difícil interpretación. Por ejemplo:

```
(2==1) || (-1==-1) // el resultado es 1
(2==2) && (3==-1) // el resultado es 0
((2==2) && (3==3)) || (4==0)
                   // el resultado es 1
((6==6) || (8==0)) && ((5==5) &&(3==2))
                   // el resultado es 0
```

9.2.6.Operadores de manejo de bits.

Los **operadores de manejo de bits** permiten actuar sobre los operandos a nivel de bits. Los operandos sólo pueden ser de tipo **entero**, pues es el único caso en que tendrán sentido estas operaciones.

En C los operadores de bits son:

- **&** (AND)
- **|** (OR)
- **^** (XOR)
- **~** (NOT)
- **<<** (desplazamiento hacia la izquierda)
- **>>** (desplazamiento hacia la derecha).

Estas operaciones se **realizan bit a bit** de los operandos. Por ejemplo:

```
4&12       // 0100 & 1100 = 0100 =4
4|8        // 0100 | 1000 = 1100 =12
8>>3       //desplaza 3 posiciones a la
           // derecha los bits del
// número 8 en binario
// 0000000000001000 resulta:
// 0000000000000001
// los tres bits de la derecha
// se pierden
```

9.2.7.Otros operadores.

Además de los operadores vistos hasta ahora, el lenguaje C dispone de otros operadores. En esta sección se describen algunos operadores **unarios** adicionales.

9.2.7.1.Operador menos (–).

El efecto de este operador en una expresión es **cambiar el signo de la variable** o expresión que le sigue. Recuérdese que en C no hay constantes numéricas negativas. La forma general de este operador es: -expresion.

9.2.7.2.Operador más (+).

Este es un nuevo operador unario introducido en el ANSI C, y que tiene como finalidad la de servir de complemento al operador (–) visto anteriormente. Se puede anteponer a una variable o expresión como operador unario, pero **en realidad no hace nada**.

9.2.7.3.Operador sizeof().

Este es el operador de C con el nombre más largo. Puede parecer una función, pero en realidad es un operador. La finalidad del operador sizeof() es devolver **el tamaño**, en bytes, del tipo de variable introducida entre los paréntesis. Recuérdese que este tamaño depende del compilador y del tipo de ordenador que se está utilizando, por lo que es

necesario disponer de este operador para producir código portable. Por ejemplo:

```
var_1 = sizeof(double) // var_1
          // contiene el tamaño
          // de una variable double
```

9.2.7.4.Operador negación lógica (!).

Este operador devuelve un cero (false) si se aplica a un valor distinto de cero (true), y devuelve un 1 (true) si se aplica a un valor cero (false). Su forma general es: !expresion.

9.2.7.5.Operador coma (,).

Los operandos de este operador son expresiones, y tiene la forma general:

```
expresion = expresion_1, expresion_2
```

En este caso, *expresion_1* se evalúa primero, y luego se evalúa *expresion_2*. **El resultado global es el valor de la segunda expresión**, es decir de *expresion_2*. Este es el operador de menos precedencia de todos los operadores de C. Como se explicará más adelante, su uso más frecuente es para introducir expresiones múltiples en la sentencia ***for***.

9.2.7.6.Operadores dirección (&) e indirección (*).

Aunque estos operadores se introduzcan aquí de modo circunstancial, su importancia en el lenguaje C es **absolutamente esencial**, resultando uno de los puntos más fuertes (y quizás **más difíciles de dominar**) de este lenguaje. La forma general de estos operadores es la siguiente:

```
*expresion;
&variable;
```

El operador **dirección & devuelve la dirección de memoria de la variable** que le sigue.

Por ejemplo:

```
variable_1 = &variable_2;
```

Después de ejecutarse esta instrucción *variable_1* contiene la **dirección de memoria** donde se guarda el contenido de *variable_2*. Las variables que almacenan direcciones de otras variables se denominan **punteros** (o apuntadores), deben ser declaradas como tales, y tienen su propia aritmética y modo de funcionar. Se verán con detalle un poco más adelante.

No se puede modificar la dirección de una variable, por lo que no están permitidas operaciones en las que el operador & figura a la izda del operador (=), al estilo de:

```
// No es correcto
&variable_1 = nueva_direccion;
```

El operador indirección ***** es el operador **complementario** del **&**. Aplicado a una expresión que represente una dirección de memoria (puntero) **permite hallar el contenido** o valor almacenado en esa dirección. Por ejemplo:

```
variable_3 = *variable_1;
```

El **contenido** de la dirección de memoria representada por la variable de tipo puntero *variable_1* se recupera y se asigna a la variable *variable_3*.

Como ya se ha indicado, las variables puntero y los operadores dirección (&) e indirección (*) **serán explicados con mucho más detalle en una sección posterior**.

9.3.Expresiones.

Ya han aparecido algunos ejemplos de expresiones del lenguaje C en las secciones precedentes. Una expresión es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos. Por ejemplo, **1+5** es una expresión formada por dos operandos (1 y 5) y un operador (el +); esta expresión **es equivalente al valor 6**, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado.

Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos.

En C existen distintos tipos de expresiones.

9.3.1.Expresiones aritméticas.

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (**+, -, *, /, %, ++, --**). Como se ha dicho, también se pueden emplear **paréntesis** de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, **la solución de la ecuación de segundo grado:**

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Se escribe, en C en la forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

Donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El **conjunto** de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una **sentencia**. En la expresión anterior aparece la llamada a la función de librería ***sqrt()***, que tiene como valor de retorno la **raíz cuadrada** de su único argumento.

En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/
(2 * a);
```

9.3.2.Expresiones lógicas.

Los elementos con los que se forman estas expresiones son **valores lógicos**; **verdaderos** (true, o distintos de 0) y **falsos** (false, o iguales a 0), y los **operadores lógicos ||, && y !**. También se pueden emplear los **operadores relacionales (<, >, <=, >=, ==, !=)** para producir estos valores lógicos a partir de valores numéricos. Estas expresiones equivalen siempre a un **valor 1 (true) o a un valor 0 (false)**. Por ejemplo:

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

Donde de nuevo la expresión lógica es lo que está entre el operador de asignación (=) y el (;). **La variable *a* valdrá 1 si *b* es mayor que *c* y *c* mayor que *d*, ó si *c* es igual a *e* ó *e* es igual a *b*.**

9.3.3.Expresiones generales.

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C es su **flexibilidad para combinar expresiones y operadores de distintos tipos** en una expresión que se

podría llamar general, aunque es una expresión absolutamente ordinaria de C.

> Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un 0).

Esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico:

> true si es distinto de 0 y false si es igual a 0.

Esto permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:

```
(a - b*2.0) && (c != d)
```

A su vez, el operador de asignación (**=**), además de introducir un nuevo valor en la variable que figura a su izda, **deja también este valor disponible para ser utilizado en una expresión** más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables a, b y c:

```
a = b = c = 1;
```

Que equivale a: *a = (b = (c = 1))*; En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a *c*; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a *b*; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable *a*.

9.4.Reglas de precedencia y asociatividad.

El resultado de una expresión depende del **orden en que se ejecutan las operaciones**. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

```
3 + 4 * 2
```

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11.

Con objeto de que el resultado de cada expresión quede **claro e inequívoco**, es necesario definir las reglas que definen el **orden** con el que se ejecutan las expresiones de C.

Existen dos tipos de reglas para determinar este orden de evaluación:

- Reglas de precedencia.
- Asociatividad.

Además, el orden de evaluación puede **modificarse por medio de paréntesis**, pues siempre se realizan primero las operaciones encerradas en los paréntesis más interiores.

Los distintos operadores de C se ordenan según su distinta **precedencia o prioridad**; para operadores de la misma precedencia o prioridad, en algunos el **orden de ejecución** es de izquierda a derecha, y otros de derecha a izquierda (se dice que se asocian de izda a dcha, o de dcha a izda). A este orden se le llama **asociatividad**.

En la tabla se muestra la precedencia (**disminuyendo de arriba abajo**) y la asociatividad de los operadores de C. En dicha Tabla se incluyen también algunos operadores que no han sido vistos hasta ahora.

Precedencia	*Asociatividad*
() [] -> .	*izda a dcha*
++ -- ! sizeof (tipo) *+(unario) -(unario) *(indir.) &(dirección)*	*dcha a izda*
** / %*	*izda a dcha*
+ -	*izda a dcha*
< <= > >=	*izda a dcha*
== !=	*izda a dcha*
&&	*izda a dcha*
||	*izda a dcha*
?:	*dcha a izda*
*= += -= *= /=*	*dcha a izda*
, (operador coma)	*izda a dcha*

En la Tabla anterior se indica que el operador (*) tiene precedencia sobre el operador (+). Esto quiere decir que, en ausencia de paréntesis, el resultado de la expresión *3+4*2* es *11* y no *14*. Los operadores binarios (+) y (-) tienen igual precedencia, y asociatividad de izda a dcha. Eso quiere decir que en la siguiente expresión:

```
 a-b+d*5.0+u/2.0 // (((a-b)+(d*5.0))+(u/
2.0))
```

El orden de evaluación es el indicado por los paréntesis en la parte derecha de la línea (Las últimas operaciones en ejecutarse son las de los paréntesis más exteriores).

9.5.Sentencias.

Las **expresiones** de C son unidades o componentes elementales de unas entidades de rango superior que son las **sentencias**. Las sentencias son unidades completas, **ejecutables en sí mismas**. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

9.5.1.Sentencias simples.

Una sentencia simple es una expresión de algún tipo terminada con un **carácter (;)**. Un caso típico son las declaraciones o las sentencias aritméticas. Por ejemplo:

```
float real;
espacio = espacio_inicial + velocidad *
tiempo;
```

9.5.2.Sentencia vacía ó nula.

En algunas ocasiones es necesario introducir en el programa una sentencia que ocupe un lugar,

pero que **no realice ninguna tarea**. A esta sentencia se le denomina sentencia vacía y consta de un simple carácter (;). Por ejemplo:

```
for (a=0;a<5000;a++);
```

La sentencia anterior produce un retardo, no hace nada (;) hasta que a=5000.

9.5.3.Sentencias compuestas o bloques.

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de **sentencias compuestas**. Una sentencia compuesta es un conjunto de declaraciones y de sentencias **agrupadas dentro de llaves {...}**. También se conocen con el nombre de **bloques**. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas. Un ejemplo de sentencia compuesta es el siguiente:

```
{
int i = 1, j = 3, k;
double masa;
masa = 3.0;
k = y + j;
}
```

Las sentencias compuestas se utilizarán con mucha frecuencia en las estructuras de control de flujo.

9.6.Tablas-resumen.

OPERADORES ARITMÉTICOS		
Operador	**Operación**	**Ejemplo**
-	Resta o cambio de signo	cosa=3-valor;
+	Suma	printf(“%d”,4+6);
*	Producto	valor=2*PI*r;
/	División	test=5/9;(test es 0);
%	Módulo (Resto) de una división de enteros	resto=divdo%divsor;
OPERADORES DE ASIGNACIÓN E INCREMENTO		
Operador	**Operación**	**Ejemplo**
=	Asignación (¡No igual!)	Y=10;x=2*a;a=getchar()
--	Decrementar	Y=--x; Y=x--;
++	Incrementar	Y=++x; Y=x++;
+=	Suma y asignación	Y+=2; (Y=Y+2)
-=	Resta y asignación	Y-=b; (Y=Y-b)
*=	Producto y asignación	Y*=(7/c); (Y=Y*(7/c))
/=	División y asignación	Y/=4; (Y=Y/4)
En general:Variable operador=Expresión		oprerador=[+, -, *, /, %, <<, >>, &, | y

OPERADORES RELACIONALES		
Operador	**Operación**	**Ejemplo**
<	Es Menor	Resul=a<b;
>	Es Mayor	If(a>b) resul=b>c;
<=	Es Menor o igual	Resul=a<=b;
>=	Es Mayor o igual	If(c>=x) b=23;
==	Es Igual	While(b==c) {.....}
!=	Es Distinto	If(a! ▪=3){.....}

OPERADORES LÓGICOS		
Operador	**Operación**	**Ejemplo**
&&	AND	resul=dato1&&dato2;
‖	OR	resul=dato1‖dato2;
!	NOT	resul=!dato;

OPERADORES DE MANEJO DE BITS. (SÓLO PARA ENTEROS)		
Operador	**Operación**	**Ejemplo**
&	AND entre bits	C=a&b;
|	OR entre bits	B=c|a;
^	XOR entre bits	X=b^y;
~	Inversor bit a bit	B=~v
<<	Desplazamiento a la izquierda, por la derecha entran ceros	B<<4; Desplaza B cuatro bits
>>	Desplazamiento a la derecha, por la izquierda entran ceros o el bit de signo	A>>8; Desplaza A ocho bits

OTROS OPERADORES		
Operador	**Operación**	**Ejemplo**
Sizeof()	Tamaño en Byts	A=sizeof(float);
,	Coma	for(a=0,b=1;a<10;a++,b--)
&	Dirección	punt=&b;
*	Indirección	B=*punt;

9.7.Ejemplos.

Cada expresión tiene su valor:

Expresión	Valor
-4+6	2
C=3+8	11 ; C también vale 11
5>3	1 (verdadero)
10<1	0 (falso)
6+(c=3+8)	17
2>5+1	0 (la suma tiene preferencia)
a=8>5+1	1 (el igual tiene preferencia)
a=(8>5)+1	2
a=(8>5)+(b=1<3)	2

Para no tener problemas con las preferencias, es mejor colocar paréntesis.

```
/* cuadrados.c -- produce una tabla de
20 cuadrados */
#include <stdio.h>
int main(void)
{
    int num = 1;
    while (num < 21)
    {
           printf("%10d %10d\n", num,
                                  num*num);
           num++;
    }
    return 0;
}
```

Podemos hacer el bucle más C:

```
while(num<21) printf("%10d %10d\n",
                  num++, num*num);
```

O podemos hacerlo más obstruso:

```
while(printf("%10d %10d\n", num++,
                  num*num)&&num<21);
```

En el primer caso se utiliza el **postincremento** dentro de los argumentos de printf(), es decir, se incrementa num después de imprimirlo (num++).

En el segundo caso se realiza toda la función en una única línea, aprovechando que printf) devuelve el número de caracteres escritos en esa llamada. El bucle se realizará mientras se imprima correctamente y num sea menor que 21.

```
/* botellas.c -- cuenta atrás */
#include <stdio.h>
#define MAX 100
int main(void)
{
    int cont = MAX + 1;
    while (--cont > 0)
    {
           printf("¡%d botellas de vino en el estante, %d botellas!\n",

    cont, cont);
           printf("Alguien pasó por delante y ¿qué fue de ellas?\n");
           if (cont!=1)printf("¡Sólo quedan %d botellas!\n\n", cont - 1);
           else printf("El vino se terminó.");
    }
    return 0;
}
```

9.8.Ejercicios:

a) Suponiendo que todas las variables son de tipo int. Indicar el valor de cada una de las siguientes expresiones:

Expresión,Valor

(2+3)*6	
a=(22>5*5)	
(12+6)/2*3	
22==(a=(22<5*5)+21)	
(2+3)/4	
2==1+(a=3>1==(b=1))	
3+2*(x=7/2)	
4==(5*2+5>8)+3)	

b) Sospecho que hay varios errores en el siguiente programa. ¿Podrías localizarlos?

```
/*ejer9-b.c*/
#include <stdio.h>
int main(void)
{
    int i=1;
    float n;
    printf("¡Ojo, que va una ristra");
    printf(" de fracciones\n");
    while (i<30)
          n=1/i;
          printf(" %f",n);
    printf("Esto es todo, amigos");
    return 0;
}
```

c) Si los siguientes fragmentos formasen parte de un programa completo ¿Qué imprimirían?.

```
int x=0;
while(++x<3)
printf("%d",x);
```

```
int x=100;
while(x++<103)
printf("%d\n",x);
printf("%d\n",x);
```

```
char ch='s';
while(ch<'w')
{
printf("%c",ch);
ch++;
}
printf("%c\n",ch);
```

d) Diseña un programa que realice lo siguiente:

- Incremente en 10 la variable x.
- Incremente en 1 la variable x.
- Asigne a c el doble de la suma de a y b.
- Asigne a c el valor de a más el doble de b.

Hay que procurar utilizar los operadores **más adecuados** a las operaciones a realizar.

e) Usando un **bucle while**, convierte valores en minutos, introducidos por teclado, en otros expresados en horas y minutos.

f) Realizar un programa que solicite un número entero y a continuación imprima todos los enteros comprendidos entre este valor y otro superior en 10 unidades, ambos inclusive.

g) Diseñar un programa que sume el cuadrado de los números comprendidos entre 1 y un número introducido por teclado.

h) Modificar el programa anterior para que una vez finalizado, solicite una nueva cantidad para repetir el proceso. Se interrumpirá cuando se introduzca una letra. Preparar una salida honrosa cuando se introduzca un 0.

10.Arrays y cadenas.

10.1.Arrays.

Un **array** (también conocido como **vector** o **matriz**) es un modo de manejar una gran cantidad de **datos del mismo tipo** bajo un mismo nombre o identificador, y que se distinguen mediante uno o varios **índices** que se colocan entre **corchetes** " **[ ]** " a continuación del nombre o identificador.

Los arrays pueden ser **unidimensionales** o **multidimensionales**. En los arrays unidimensionales, cada elemento puede ser accedido de forma individual, mediante el nombre del array seguido de un índice, por ejemplo, la expresión dato[8]; hace referencia al elemento número 9 del array dato (el primer índice es 0).

En los arrays multidimensionales hacen falta tantos índices como dimensiones para hacer referencia a cada elemento, por ejemplo: dato[4][3]; hace referencia al elemento situado en la fila 4, columna 3, del array dato (existe la fila 0, columna 0).

Todos los elementos de un array se sitúan en posiciones de memoria **contiguas** (el primer elemento en la posición de memoria más baja), ocupando tantos bytes de memoria como sean necesarios para almacenar todos los elementos.

10.1.1.Declaración de arrays.

La declaración de arrays se rige por las mismas normas que las de cualquier otro tipo de datos. Es necesario declarar el array para que el compilador reserve la cantidad necesaria para almacenar todos los elementos del array de forma contigua.

El compilador, una vez asignada la memoria necesaria, **no realiza ninguna comprobación de los límites del array**, esto significa que el compilador no hará comprobaciones sobre los índices que se utilicen en el programa, pudiendo, por tanto, acceder a posiciones de memoria que no corresponden al array. Es **responsabilidad del programador** establecer los controles necesarios para manejar correctamente los índices.

10.1.1.1.Arrays unidimensionales.

La forma general de la declaración de un vector es la siguiente:

```
tipo nombre[numero_elementos];
```

Los elementos se numeran **desde 0** hasta **numero_elementos-1**. El tamaño de un vector puede definirse con cualquier expresión constante **entera**. Para definir tamaños son particularmente útiles las constantes simbólicas (**#define**). Por ejemplo, mediante la sentencia:

```
double a[10];
```

Se reserva espacio para **10 variables** de tipo **double**. Las 10 variables se llaman ***a*** y se accede a una u otra por medio de un índice, que es una expresión entera escrita a continuación del nombre entre corchetes **[...]**.

En C no se puede operar con todo un vector o toda una matriz como una única entidad, sino que hay que tratar sus elementos **uno a uno** por medio de bucles *for* o *while*.

Los elementos de un vector se utilizan en las expresiones de C como cualquier otra variable. Ejemplos de uso de vectores son los siguientes:

```
a[5] = 0.8;
a[9] = 30. * a[5];
a[0] = 3. * a[9] - a[5]/a[9];
a[3] = (a[0] + a[9])/a[3];
```

El **tamaño** de un array debe ser conocido por el compilador desde el momento de la declaración. El tamaño se puede indicar, o bien expresándolo en la **declaración** (int a[10]), como se ha explicado anteriormente, o bien **inicializando** los elementos en la propia declaración. Por ejemplo la sentencia:

```
float notas[]={1.5,2.3,4.5,6.7};
```

Declara e inicializa un array llamado notas que consta de 4 elementos.

La cantidad de memoria que ocupa un array unidimensional viene dada por la expresión:

```
bytes=tamaño*sizeof(tipo_dato_array)
```

Donde *tamaño* es el número de elementos del array y *sizeof* es un operador que devuelve el **número de bytes** que ocupa la expresión a la que se aplica, en este caso *tipo_dato_array*.

En el último ejemplo, la memoria ocupada por el array *notas[]* será:

```
bytes=4*sizeof(float);
```

10.1.1.2.Arrays multidimensionales.

Para declarar un array multidimensional hay que indicar expresamente el número de dimensiones mediante los correspondientes índices. La forma general de la declaración es:

```
tipo nombre[tam1][tam2]....[tamN];
```

Donde *tam1, tam2,....* y *tamN* son expresiones que indican el **tamaño de cada una de las dimensiones del array**. Se numeran también a partir de 0. La forma de acceder a los elementos de la matriz es utilizando su nombre, seguido de las expresiones enteras correspondientes a los dos subíndices, entre corchetes.

Los elementos de un array multidimensional se almacenan en memoria de forma consecutiva, siendo el índice de la derecha el que cambia más rápidamente.

La cantidad de memoria utilizada corresponde a la expresión:

```
bytes=tam1*tam2*...*tamN*sizeof(tipo)
```

No es habitual el uso de arrays de más de tres dimensiones, por lo complejo que resulta su manejo y la cantidad de memoria que necesitan. Son muy frecuentes, sin embargo, los arrays bidimensionales, también llamados **tablas**, organizados en filas y columnas. Su declaración sería:

```
tipo nombre[Nº_filas][Nº_columnas];
```

10.1.2.Inicialización de arrays.

La inicialización de un array se puede hacer de varias maneras:

- Declarando el array como tal e inicializándolo luego mediante lectura o asignación por medio de un bucle *for*:

```
double vect[N];
...
for(i = 0; i < N; i++)
scanf(" %lf", &vect[i]);
```

O bien en el caso de bidimensionales:

```
Int matriz[N][M], i , j;
...
for(i=0;i<N;i++)
    for(j=0;j<M;j++)
    {
    printf("Teclea el dato (%d, %d):",
                                i, j);
    scanf("%d",&matriz[i][j]);
    }
```

- Inicializándolo en la misma declaración, en la forma:

```
double v[6] = {1., 2., 3., 3., 2., 1.};
    /*Es necesario poner un punto
     decimal tras cada cifra,
     para que ésta sea reconocida
     como un valor de tipo //double.*/

int mat[3][2] = {{1, 2}, {3, 4}, {5,
6}};
```

Cuando se inicializa un array en su declaración, es posible **omitir** el tamaño de su primera dimensión (la más a la izquierda). El compilador calcula cada dimensión en función del número de datos que se incluyen, de modo que son válidas las siguientes líneas para declarar e inicializar arrays:

```
float d[] = {1.2, 3.4, 5.1};
    /* d[3] está implícito*/

int mat[][2] = {{1, 2}, {3, 4}, {5, 6}};
    / mat[3][2], el 3 está implícito*/
```

Es posible inicializar o actualizar elementos individuales de un array como variables ordinarias, mediante una sentencia de asignación cuyo destino sea el nombre del array con los índices correspondientes al elemento.

A continuación se muestran algunos ejemplos de declaración:

```
int f[100] = {0};
    /* todo se inicializa a 0*/
int h[10] = {1, 2, 3};
    /* restantes elementos a 0*/
float matriz [3][4]={0.0, 0.1, 0.2, 0.3,
                     1.0, 1.1, 1.2, 1.3,
                     2.0, 2.1, 2.2, 2.3};
```

```
char alfabet[][5]={'a','b','c','d','e',
                   'f','g','h','i','j',
                   'k','l','m','n','o',
                   'p','q','r','s','t',
                   'u','v','x','y','z'};
```

En la inicialización de arrays multidimensionales es importante recordar que **cambia más rápidamente el índice situado más a la derecha**. En el caso de bidimensionales las **columnas** son las que más rápido cambian.

En las inicializaciones anteriores, los arrays *matriz[]* y *alfabet[]*, dispondrán sus elementos en filas y columnas según sus dimensiones independientemente de la colocación de los elementos en el archivo. El orden que se ha elegido solo sirve para que el código fuente sea **más legible** y se conozcan más fácilmente los índices que corresponden a cada elemento. Por lo tanto también es válida la siguiente declaración (aunque menos legible):

```
char alfabet [][5]={    'a', 'b', 'c',
'd', 'e', 'f', 'g', 'h', 'i', 'j','k',
'l', 'm', 'n', 'o',
          'p', 'q', 'r', 's', 't', 'u',
'v', 'x', 'y', 'z'};
```

Recuérdese que, al igual que las variables escalares correspondientes, los arrays con modo de almacenamiento ***external*** y ***static*** se inicializan a **cero** automáticamente en el momento de la declaración.

Sin embargo, esto no está garantizado en los arrays ***auto***, y el que se haga o no depende del compilador.

10.2.Cadenas.

Una **cadena de caracteres** (**string** o simplemente **cadena**) es un array unidimensional en el que todos sus elementos son del tipo ***char***, y en el que, por convenio, el **último** elemento es el carácter nulo (***NULL***), que se representa por **'\0'**.

La declaración de una cadena es igual a la de un array inidimensional, con la salvedad de que el tipo de dato será *char*, es decir:

```
char nombre_cadena [longitud_cadena];
```

Donde *numbre_cadena* es el identificador de la cadena y *longitud_cadena* es una constante de tipo entero que determina el número de elementos de la cadena incluido el carácter nulo.

La inicialización de una cadena puede realizarse en el momento de la declaración de alguna de las siguientes formas:

```
char cadena[13]="portapapeles";
char cadenilla[5]={ 'H', 'o', 'l', 'a',
'\0'};
```

En los dos casos el tamaño se determina **contando también el carácter nulo del final** de la tira. Si el tamaño es menor que la constante asignada se produce un error de compilación, y si es mayor se rellenará con caracteres nulos.

El tamaño de la cadena se puede **omitir** en la inicialización, por lo tanto son válidas las siguientes inicializaciones:

```
char nombre[]="Pedro";
char apellido[]={ 'P', 'e', 'r', 'e',
'z', '\0'};
```

También es posible acceder a los elementos del array de forma independiente, mediante el nombre y la posición del elemento. Por ejemplo la expresión ***nombre[3]*** hace referencia al carácter '**d**' del ejemplo anterior.

Un array de caracteres ocupa tantas posiciones de memoria como hayan sido reservadas en al declaración, aunque la cadena almacenada puede tener una longitud inferior; **será el carácter nulo el que determinará el final de la cadena**.

10.2.1.Funciones que utilizan cadenas de caracteres.

Puesto que los arrays y las cadenas no son un tipo de datos en C, **no existen operadores** cuyos objetos puedan ser arrays y cadenas, sin embargo hay un considerable número de **funciones** que permiten realizar las operaciones más habituales con las cadenas. En este caso, el nombre del array o de la cadena sirve para hacer referencia a todos los elementos como un conjunto.

10.2.1.1.Función *strlen()*.

El prototipo de esta función es como sigue:

```
unsigned strlen(const char *s);
```

Su nombre proviene de **string length**, y su misión es contar el número de caracteres de una cadena, **sin incluir el '\0'** final. El paso del argumento se realiza por referencia, pues como argumento se emplea un puntero a la cadena (nombre de la cadena), y devuelve un entero sin signo que es el número de caracteres de la cadena.

La palabra **const** impide que dentro de la función la cadena de caracteres que se pasa como argumento sea modificada.

10.2.1.2.Función strcat().

El prototipo de esta función es como sigue:

```
char *strcat(char *s1, const char *s2);
```

Su nombre proviene de **string concatenation**, y se emplea para **unir dos cadenas** de caracteres poniendo *s2* a continuación de *s1*. El valor de retorno es un puntero a *s1*. Los argumentos son los punteros a las dos cadenas que se desea unir. La función almacena la cadena completa en la primera de las cadenas. **¡PRECAUCIÓN!** Esta función no prevé si tiene sitio suficiente para almacenar las dos cadenas juntas en el espacio reservado para la primera. Esto es responsabilidad del programador.

10.2.1.3.Funciones strcmp() y strcasecmp().

El prototipo de la función *strcmp()* es como sigue:

```
int strcmp(const char *s1, const char *s2)
```

Su nombre proviene de **string comparison**.

Sirve para **comparar** dos cadenas de caracteres. Como argumentos utiliza punteros a las cadenas que se van a comparar. La función devuelve **cero si las cadenas son iguales**, un valor **menor que cero si s1 es menor** (en orden alfabético) que s2, y un valor **mayor que cero si s1 es mayor** que s2.

La función ***strcasecmp()*** es completamente análoga, con la diferencia de que no hace distinción entre letras mayúsculas y minúsculas.

10.2.1.4.Función strcpy().

El prototipo de la función *strcpy()* es como sigue:

```
char *strcpy(char *s1, const char *s2)
```

Su nombre proviene de **string copy** y se utiliza para **copiar** cadenas. Utiliza como argumentos dos punteros a carácter: el primero es un puntero a la cadena copia, y el segundo es un puntero a la cadena original. El valor de retorno es un puntero a la cadena copia s1.

Es muy importante tener en cuenta que en C no **se pueden copiar cadenas de caracteres directamente**, por medio de una sentencia de asignación. Por ejemplo, sí se puede asignar un texto a una cadena en el momento de la **declaración**:

```
char s[] = "Esto es una cadena";
      /* correcto
```

Sin embargo, sería **ilícito** hacer lo siguiente:

```
char s1[20] = "Esto es una cadena";
char s2[20];
...
/* Si se desea que s2 contenga una copia
de s1*/
s2 = s1;          /* INCORRECTO: se hace
                     una copia de
                     punteros*/
strcpy(s2, s1);   /* CORRECTO: se copia
                     toda la cadena*/
```

10.2.1.5.Otras funciones útiles.

Se pueden consultar más funciones de cadenas con **man string**.

Y en la librería stdlib.h se encuentran también estas:

- ***atof(cad)*** devuelve el valor en doble precisión equivalente al representado por los dígitos que *cad* contiene.
- ***atoi(cad)*** devuelve el valor entero equivalente al representado por los dígitos que *cad* contiene.
- ***atol(cad)*** devuelve el valor entero largo equivalente al representado por los dígitos que *cad* contiene.
- ***fcvt()*** convierte un número real a una cadena de caracteres.

10.3.Arrays de cadenas de caracteres.

Un array de cadenas es un array bidimensional de caracteres en el que el índice izquierdo señala el **número de cadenas** y derecho la **máxima longitud** de cada cadena:

```
char numbre_array[num_cad][longi_cad];
```

Donde *num_cad* es el número de cadenas que contiene el array y *longi_cad* es la longitud máxima que podrá tener cualquiera de las cadenas (incluido el carácter nulo).

Se puede acceder a cada una de las cadenas mediante el nombre del array y **sólo el índice que indica el número de cadena**. Esto facilita, por ejemplo, la presentación de mensajes diversos según determinadas variables. Por ejemplo:

```
char frases[3][80]={“Error de lectura.”,
          “Error de escritura.”,
          “Error de acceso.”};
```

Reserva espacio para tres cadenas de 80 caracteres cada una (incluido el carácter nulo), lo que implica que **se reserva más espacio del necesario**, pero permite asegurar que todas las frases serán almacenadas correctamente. En el programa donde se incluya esta declaración, la sentencia ***puts(frase[1]);*** hará que aparezca en pantalla el mensaje **Error de escritura**.

10.4.Ejemplos.

```
/* diasmes3.c -- inicializa 10 de los 12 elementos */
#include <stdio.h>

int main(void)
{
    int indice;
    int dias[] = {31,28,31,30,31,30,31,31,30,31};
    for (indice = 0; indice < sizeof dias/sizeof(int); indice++)
    printf("El mes %d tiene %d días.\n", indice + 1,
    dias[indice]);
    return 0;
}
```

Los valores no inicializados tendrán el valor **0** po ser una variable de tipo ***auto***.

```
/* lluvia.c --   calcula totales anuales, promedio anual y promedio
                       mensual de datos pluviométricos en un periodo
                       determinado */
#include <stdio.h>
#define DOCE 12   /* número de meses del año */
#define ANS   5   /* número de años a tratar */
int main(void)
{
    /* inicialización de datos de lluvia en el periodo 1987 - 91 */
    static float lluvia[ANS][DOCE] = {
           {10.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.3, 6.1, 7.4},
           {9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 5.2},
           {6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 4.2},
           {4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
           {8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.2}
           };
    int anno, mes;
    float subtot, total;

    printf(" AÑO      LLUVIA (cm.)\n\n");

    for (anno = 0, total = 0; anno < ANS; anno++)
    {  /* en cada año suma la lluvia de todos los meses */
           for (mes = 0, subtot = 0; mes < DOCE; mes++)
                  subtot += lluvia[anno][mes];
           printf("%5d %12.1f\n", 1970 + anno, subtot);
           total += subtot;   /* calcula lluvia total del periodo */
    }
    printf("\nEl promedio anual ha sido %.1f cm.\n\n", total/ANS);

    printf("PROMEDIOS MENSUALES:\n\n");
    printf(" Ene  Feb  Mar  Abr  May  Jun  Jul  Ago  Sep  Oct ");
    printf(" Nov  Dic\n");
    for (mes = 0; mes < DOCE; mes++)
    {   /* suma la lluvia de todos los años en cada mes */
           for (anno = 0, subtot = 0; anno < ANS; anno++)
                  subtot += lluvia[anno][mes];
           printf("%4.1f ", subtot/ANS);
    }
    printf("\n");
    return 0;
}
```

```
/* test.c - recortador de tiras de caracteres */
#include <stdio.h>
#include <string.h> /* contiene declaraciones de funciones de tiras */

int main(void)
{
    char cuteria[] = "Agárreseme que vienen curvas.";
    int corte;
    puts(cuteria);
    puts("Por donde quieres cortar?");
    scanf("%d",&corte);
    if (corte > strlen(cuteria))
    {
          puts("No tengo tantos caracteres");
          return 1;
    }
    cuteria[corte]='\0';
    puts(cuteria);
    return 0;
}
```

```
/* str_cat.c -- uniendo dos tiras */
#include <stdio.h>
#include <string.h>/* declara la función strcat() */
#define TAM 80
int main(void)
{
    char flor[TAM];
    char apendice[] = " huelen a zapato usado.";

    puts("¿Cuáles son sus flores favoritas?");
    gets(flor);               /* gets es peligrosa, continuará guardando
                                    caracteres una vez alcanzado el final del flor*/
    strcat(flor, apendice);
    puts(flor);
    puts(apendice);           /* sigue conteniendo lo mismo*/
    return 0;
}
```

```
/* une_mira.c -- unión de dos tiras comprobando si caben */
#include <stdio.h>
#include <string.h>
#define MAX 80
int main(void)
{
    char flor[MAX];
    char apendice[] = " huelen a zapato usado.";
    puts("¿Cuáles son sus flores favoritas?");
    gets(flor);               /* gets es peligrosa, continuará guardando
                                    caracteres una vez alcanzado el final del flor*/
    if ((strlen(apendice) + strlen(flor) + 1) <= MAX)
          strcat(flor, apendice);
    else puts("No me cabe");
          puts(flor);
    return 0;
}
```

```
/* no_va.c -- ¿funciona esto? */
#include <stdio.h>
#define RESP "Blanco"
int main(void)
{
    char prueba[40];
    puts("¿De qué color es el caballo blanco de Santiago?");
    gets(prueba);
    while (prueba != RESP)
          {
                puts("No tienes ni idea. Prueba otra vez.");
                gets(prueba);
          }
    puts("¡Correcto!");
    return 0;
}
```

```
/* compara.c -- este sí funciona */
#include <stdio.h>
#include <string.h>/* declara strcmp() */
#define RESP "Blanco"
#define MAX 40
int main(void)
{
    char prueba[MAX];
    puts("¿De qué color es el caballo blanco de Santiago?");
    gets(prueba);
    while (strcmp(prueba,RESP) != 0)
    {
          puts("No tienes ni idea. Prueba otra vez.");
          gets(prueba);
    }
    puts("¡Correcto!");
    return 0;
}
```

```
/* comp_ret.c -- retorno de strcmp */
#include <stdio.h>
#include <string.h>
int main(void)
{
    printf("%d\t", strcmp("A", "A") );
    printf("%d\t", strcmp("A", "B") );
    printf("%d\t", strcmp("B", "A") );
    printf("%d\t", strcmp("C", "A") );
    printf("%d\n", strcmp("manzanas", "manzana") );
    return 0;
}
```

```
La salida del programa sería: 0      -1     1      1      1
```

```
/* copia1.c -- strcpy() en acción */
#include <stdio.h>
#include <string.h>/* declara strcpy() */
#define FRASE "Reconsidere su ultima entrada, por favor."
#define TAM 50
int main(void)
{
    char orig[] = FRASE;
    char copia[TAM] = "espacio reservado";
    puts(orig);
    puts(copia);
    strcpy(copia, orig);      //cuidado con el tamaño de copia
    puts(orig);
    puts(copia);
    return 0;
}
```

```
/* copia2.c -- strcpy() en acción segundo*/
#include <stdio.h>
#include <string.h>/* declara strcpy() */
#define TAM 40
int main(void)
{
    char orig[] = "burro";
    char copia[TAM] = "No sea blando con sus vicios";
    puts(orig);
    puts(copia);
    strcpy(copia + 7, orig);
    puts(copia);
    return 0;
}
```

```
/* comando.c -- genera una tira de órdenes */
// Utiliza sprintf() en stdio.h y system() en stdlib.h
#include <stdio.h>
#include <stdlib.h>    /* ANSI C: declara la función system() */
#define MAX 20
int main(void)
{
    char origen[MAX];
    char destino[MAX];
    char orden[2 * MAX + 5];

    puts("Introduzca nombre del fichero a copiar:");
    gets(origen);
    puts("Introduzca nombre elegido como destino:");
    gets(destino);
    sprintf(orden, "cp %s %s", origen, destino);
    printf("Ejecuto la siguiente orden: %s\n", orden);
    system(orden);
    return 0;
}
```

```
/* tiras_1.c -- hágase en solitario */
#include <stdio.h>
#define MSJ "Debe tener muchas cualidades. Dígame algunas."
/* constante simbólica de tira de caracteres   */
#define LIM  5
#define LONLIN 81         /* longitud máxima + 1 */
char m1[] = "Limítese a una sola línea.";
                                    /* inicialización de un array de char externo  */
char *m2 = "Si no se le ocurre nada, apaga y vámonos.";
                                /* inicialización de un puntero a char externo */

int main(void)
{
    char nombre[LONLIN];
    static char talentos[LONLIN];
    int i;
    int cont = 0;
    char *m3 = "\nYa basta sobre mí -- ¿cómo se llama?";
                                /* inicialización de un puntero */
    static char *mistal[LIM] = { "Sumo números con sutileza",
                        "Multiplico con precisión", "Almaceno datos",
                        "Sigo instrucciones al pie de la letra",
                        "Entiendo el lenguaje C"};
                        /* inicialización de un array de tiras */

    printf("¡Hola! Soy Juanito, su ordenador favorito.\n");
    printf("Tengo muchas cualidades. Le diré algunas.\n");
    puts("Las tengo en la punta del byte...¡Ah, sí! ahí van:");
    for (i = 0; i < LIM; i++)
           puts(mistal[i]);  /* imprime las cualidades del ordenador */
    puts(m3);
    gets(nombre);
    printf("Bien, %s. %s\n", nombre, MSJ);
    printf("%s\n%s\n", m1, m2);
    gets(talentos);
    puts("A ver si he cogido toda la lista:");
    puts(talentos);
    printf("Gracias por la información, %s.\n", nombre);
    return 0;
}
```

La salida del programa será;

```
¡Hola! Soy Juanito, su ordenador favorito.
Tengo muchas cualidades. Le diré algunas.
Las tengo en la punta del byte...¡Ah, sí! ahí van:
Sumo números con sutileza
Multiplico con precisión
Almaceno datos
Sigo instrucciones al pie de la letra
Entiendo el lenguaje C

Ya basta sobre mí -- ¿cómo se llama?
José Perez
Bien, José Perez. Debe tener muchas cualidades. Dígame algunas.
Limítese a una sola línea.
Si no se le ocurre nada, apaga y vámonos.
Programador, despellejador de cerebros y comedor de garbanzos.
A ver si he cogido toda la lista:
Programador, despellejador de cerebros y comedor de garbanzos.
Gracias por la información, José Perez.
```

10.5.Ejercicios.

a) Realiza un programa que lea una lista de N números enteros desde teclado y los muestre a continuación, calcule la media aritmética e indique cuanto ocupan en memoria.

b) Escribe un programa que lea desde teclado una matriz de 4X5 elementos de tipo real, muestre los datos leídos e indique la cantidad de memoria que ocupan.

c) Crea un programa que lea una cadena desde teclado y calcule su longitud utilizando un bucle ***while***.

d) Realiza un programa que lea una frase del teclado y la muestre al revés en pantalla.

e) Haz un programa que lea una cadena del teclado y sustituya los espacios por guiones.

f) Diseña un programa que lea dos cadenas y, utilizando las funciones de manejo de cadenas, calcule la longitud de ambas, las compare y las concatene.

g) Elabora un programa que lea una cadena de símbolos numéricos y obtenga su valor entero y su valor real.

h) Haz un programa que reserve memoria para un array de 25 cadenas de hasta 80 caracteres y, a continuación, lea texto del teclado hasta que se introduzca una línea vacía o hasta que se llegue al límite reservado, mostrando todo lo introducido antes de finalizar.

i) Escribe un programa que extraiga una parte de una cadena principal.

j) Haz un programa que localice la posición en que aparece un carácter determinado en una cadena.

k) Elabora un programa que lea una cadena del teclado y convierta todas las letras mayúsculas en minúsculas(Mirar toupper y tolower*)*.

l)Realiza un programa que lea un conjunto de cadenas de caracteres y muestre la de mayor longitud.

11.Ordenación y búsqueda en tablas.

11.1.Ordenación de series.

La ordenación de elementos de una tabla depende del tipo de elementos que la constituyen, de modo que si se trata de elementos literales, la ordenación podrá ser alfabética directa (de la A a la Z) o alfabética inversa (de la Z a la A). Si se trata de elementos numéricos, el orden puede ser creciente o decreciente.

Hay numerosos **algoritmos** de ordenación de series y aumentan en complejidad a medida que aumenta el número de elementos de la serie, la velocidad y su eficacia. Los algoritmos de ordenación más corrientes son:

- Ordenación por inserción directa.
- Ordenación por selección directa.
- Ordenación por intercambio directo.

En la explicación de los métodos de ordenación tomo como referencia una ordenación creciente, para que sea más comprensible.

11.1.1.Ordenación por inserción directa.

Es conocido también como **método de la baraja** y requiere un proceso como el siguiente:

- En la primera pasada se toma como referencia el segundo elemento y se compara con el anterior. Si es menor se intercambian.
- Se toma el siguiente elemento y se compara con los anteriores, comenzando por el primero, intercambiándolos en caso de que sea menor.
- El paso anterior se repite hasta terminar con todos los elementos.

Se deduce, por lo anterior que hay que recorrer la tabla tantas veces como elementos tiene menos uno. Ejemplo de ordenación:

Situación inicial:	4	3	5	1	2	
Primera pasada:	**4**	**3**	5	1	2	Desordenados.
Segunda pasada:	**3**	4	**5**	1	2	Ordenados.
	3	**4**	**5**	1	2	Ordenados.
Tercera pasada	**3**	4	5	**1**	2	Desordenados.
	1	**4**	5	**3**	2	Desordenados.
	1	3	**5**	**4**	2	Desordenados.
:Cuarto paso:	**1**	3	4	5	**2**	Ordenados
	1	**3**	4	5	**2**	Desordenados.
	1	2	**4**	5	**3**	Desordenados.
	1	2	3	**5**	**4**	Desordenados.
	1	2	3	4	5	Resultado final.

11.1.2.Ordenación por selección directa.

También conocido por el método de **buscar el menor**, cumple el siguiente proceso:

- Se toma el primer elemento no ordenado y se compara con todos los desordenados, buscando el menor, y una vez localizado se intercambian.

Se repite lo anterior hasta llegar al último elemento que ya estará ordenado.

Por lo anteriormente expuesto, se deduce que se recorre la tabla tantas veces como elementos tiene menos uno.

Situación inicial:	**4**	3	5	1	2	Menor inicial tabla[0]
Primera pasada:	4	**3**	5	1	2	Menor tabla[1]
	4	3	**5**	1	2	Menor tabla[1]
	4	3	5	**1**	2	Menor tabla[3]
	4	3	5	1	**2**	Menor tabla[3]
	1	3	5	**4**	2	tabla[0]<>tabla[3]
Segunda pasada	1	**3**	5	4	2	Menor inicial tabla[1]
	1	3	**5**	4	2	Menor tabla[1]
	1	3	5	**4**	2	Menor tabla[1]
	1	3	5	4	**2**	Menor tabla[4]
	1	**2**	5	4	**3**	tabla[1]<>tabla[4]
.....						
Cuarta pasada	1	2	3	4	**5**	menor inicial tabla[3]
						Y resultado final.

11.1.3.Ordenación por intercambio directo.

También conocido como método de **la burbuja**, cumple el siguiente proceso:

- Se recorre la tabla comparando elementos contiguos y se intercambian en caso de estar desordenados. En cada recorrido se queda ordenado el mayor.

Se repite lo anterior hasta que en un recorrido no se realice ningún cambio.

Con este método no sabemos las veces que es necesario recorrer la tabla, puesto que depende de cómo estuvieran colocados los elementos inicialmente.

Situación inicial:	4	3	5	1	2	
Primera pasada	**4**	**3**	5	1	2	Desordenados
	3	**4**	**5**	1	2	Ordenados
	3	4	**5**	**1**	2	Desordenados
	3	4	1	**5**	**2**	Desordenados

Segunda pasada	**3**	**4**	1	2	5	Ordenados
	3	**4**	**1**	2	5	Desordenados
	3	1	**4**	**2**	5	Desordenados
Tercera pasada	**3**	**1**	2	4	5	Desordenados
	1	**3**	**2**	4	5	Desordenados
Cuarta pasada	1	2	3	4	5	Ordenados

No hay modificaciones, por lo tanto FIN.

11.2.Búsqueda en tablas.

La búsqueda y localización de un elemento en una tabla tiene como objetivo obtener el valor del índice o índices de su posición.

Los algoritmos de búsqueda en tablas más utilizados son: búsqueda **lineal** y búsqueda **dicotómica**.

Si la tabla está desordenada, no hay más método de búsqueda que la secuencial.

11.2.1.Búsqueda secuencial.

Se trata de recorrer **todos** los elementos de la tabla hasta localizar el que estamos buscando.

En el caso de que la tabla esté ordenada hay un par de consideraciones a tener en cuenta:

- La búsqueda será más rápida si comenzamos por el extremo más cercano al elemento buscado.
- No es necesario recorrer toda la tabla, la búsqueda termina cuando encontramos el elemento buscado o el siguiente en orden, en este último caso, el elemento buscado no existe.

11.2.2.Búsqueda binaria o dicotómica.

Es un algoritmo de búsqueda en tablas ordenadas que requiere el siguiente proceso:

- Se compara el elemento buscado con el central de la tabla. Si no coinciden se determina en que mitad se debe encontrar el elemento buscado.
- Se compara el elemento buscado con el central de la mitad de la tabla determinada en el paso anterior. Si no coinciden se determina en que submitad se encuentra.

Se repite el paso anterior en las sucesivas submitades hasta encontrar el elemento buscado o hasta que el intervalo de búsqueda es nulo, en cuyo caso el elemento buscado no se encuentra en la tabla.

11.3.Ejemplos.

```
/*inserc.c----Ordenación por inserción directa*/
#include <stdio.h>
int main (void)
{
    register int i,j;
    int aux,tama;
    int tabla[]={1,7,2,6,3,5,4,8,9,15,13,14,12};
```

```
    tama=sizeof(tabla)/sizeof(int);
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);    //Imprime los elementos desordenados
    puts("\n");

    for(i=1;i<tama;i++) //Se toma un elemento, comenzando
    {                          //por el segundo.
          for(j=0;j<i;j++)    //Se toma otro elemento,comenzando
          {                                  //por el primero, hasta el anterior
                                             //al tomado en el bucle anterior.
                if(tabla[i]<tabla[j])
                {
                      aux=tabla[i];
                      tabla[i]=tabla[j];        //Si es menor
                      tabla[j]=aux;                    //se cambian.
                }
          }
    }
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);          //Imprime la tabla ordenada.
          puts("\n");
    return 0;
}
```

```
/* selec.c------Ordenación por selección directa*/
#include <stdio.h>
int main (void)
{
    register int i,j;
    int aux,tama,k,min;
    int tabla[]={1,7,2,6,3,5,4,8,9,15,13,14,12};

    tama=sizeof(tabla)/sizeof(int);        //Obtiene el tamaño del array
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);          //Imprime los elementos desordenados
    puts("\n");

    for(i=0;i<tama-1;i++)            //Tomamos el primer elemento desordenado
    {
          min=tabla[i];                    // Se asume que es el más pequeño
          k=i;
          for(j=i+1;j<tama;j++)     //Se compara con todos los siguientes
          {
                if(min>tabla[j])
                {
                      min=tabla[j];       //Si es mayor se cambia el menor
                      k=j;                // y se apunta el lugar que ocupa
                }
          }
          tabla[k]=tabla[i];//Una vez comparado con todos los desordenados
          tabla[i]=min;       // se coloca el menor encontrado en su posición.
    }
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);    //Imprime los elementos ordenados
    puts("\n");
    return 0;
}
```

```
/*burbuja.c----Ordenación por intercambio directo*/
#include <stdio.h>

#define NO 0
#define SI 1

int main (void)
{
    register int i;
    int aux,tama,cambio=SI;
    int tabla[]={1,7,2,6,3,5,4,8,9,15,13,14,12};

    tama=sizeof(tabla)/sizeof(int);        //Calcula el tamaño del array
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);   //Imprime los elementos desordenados
    puts("\n");

    while(cambio)                          //Mientras hay algún cambio
    {
          cambio=NO;                       //Inicio del flag no hay cambio
          for(i=0;i<tama-1;i++)            //Contador de parejas de elementos
          {
                if(tabla[i]>tabla[i+1]) //Si están desordenados
                {
                      cambio=SI;                //Ha habido un cambio
                      aux=tabla[i];             //se intercambian
                      tabla[i]=tabla[i+1];
                      tabla[i+1]=aux;
                }
          }
    }
    for(i=0;i<tama;i++)
          printf("%4d",tabla[i]);   //Imprime los elementos ordenados
    puts("\n");
    return 0;
}
```

11.4.Ejercicios.

a)Realizar un programa que busque, en una serie desordenada, un valor introducido por teclado.

b)Diseñar un programa que ordene una serie y localice un valor, introducido por teclado, por búsqueda dicotómica.

c)Hacer un programa que capture datos enteros del teclado, los ordene de mayor a menor, calcule la media y los muestre por pantalla.

12.Tipos de datos derivados.

12.1.Introducción.

En C se pueden definir y declarar asociaciones de datos que dan lugar a nuevos tipos de datos más complejos. También se les llaman **tipos de datos definibles**, puesto que su diseño queda a criterio exclusivo del programador.

Estos tipos de datos están compuestos por los tipos básicos en C y su combinación con los modificadores ya estudiados, y dan lugar a las **estructuras**, las **uniones** y los **campos de bits**.

12.2.Estructuras.

Una estructura es una forma de agrupar un conjunto de datos de **distinto tipo** bajo un mismo nombre o identificador. Por ejemplo, supóngase que se desea diseñar una estructura que guarde los datos correspondientes a un alumno. Esta estructura, a la que se llamará *alumno*, deberá guardar el **nombre**, la **dirección**, el **número de matrícula**, el **teléfono**, y las **notas** en las 10 asignaturas. Todos estos elementos se declaran a la vez que la estructura a la que pertenecen y pueden ser tratados como elementos independientes, aunque siempre estarán ligados, por el nombre a la estructura de que forman parte.

Una estructura se asemeja a una ficha en la que se incluyen distintos **campos** que, a su vez, son datos de diferente tipo.

12.2.1.Definición y declaración.

Para crear una estructura hay que **definir un nuevo tipo de datos** y, posteriormente, **declarar** las variables de este nuevo tipo que queremos utilizar. Al definir una estructura **no se reserva memoria para ella**, solamente se establece el tipo de datos y las variables que van a formar parte de ella.

Cada uno de estos datos se denomina **miembro** de la estructura. El modelo o patrón de esta estructura puede crearse del siguiente modo:

```
struct alumno {
          char nombre[31];
          char direccion[21];
          unsigned long no_matricula;
          unsigned long telefono;
          float notas[10];
};
```

En este caso se define nuevo tipo de dato llamado *alumno* que estará compuesto por cinco elementos (**campos** o **miembros**) llamados: *nombre*, *direccion*, *no_matricula*, *telefono* y *notas*, cada uno de un **tipo distinto**. Hasta este momento no se ha reservado memoria para ninguna variable del tipo alumno.

Para **declarar** variables de este nuevo tipo y, por lo tanto, reservar memoria para ellas, se utiliza la siguiente sintaxis:

```
struct alumno juan, pedro, maria, marta;
```

También se puede definir y declarar variables a un tiempo de la siguiente manera:

```
struct alumno {
    char nombre[31];
    char direccion[21];
    unsigned long no_matricula;
    unsigned long telefono;
    float notas[10];
}juan,pedro,maria,marta;
```

Pudiendo declarar posteriormente otras variables de este mismo tipo utilizando la siguiente declaración:

```
struct alumno pilar, pedro;
```

Es habitual definir la estructura de manera **global** para todo el programa (fuera de las funciones). De este modo es posible declarar variables de ese tipo en cualquier función en que se precise.

Los tipos de datos, dentro de una estructura, pueden ser de cualquier tipo, incluso otras estructuras, cuyo nivel de anidamiento permitido viene dado por las características del compilador, aunque más de dos niveles puede hacer demasiado complejo el manejo de los datos. Por ejemplo, una **estructura anidada** puede ser:

```
struct nombre{
    nom[12];
    ap1[12];
    ap2[12];
};

struct alumno {
    struct nombre nom;
    char direccion[21];
    unsigned long no_matricula;
    unsigned long telefono;
    float notas[10];
}juan,pedro,maria,marta,alumno1;
```

Los elementos de una estructura se almacenan en memoria de forma consecutiva y en el mismo orden en que se declaran.

La memoria ocupada por la estructura será la suma de los bytes que ocupa cada uno de sus campos y se podrá obtener con el operador ***sizeof()*** aplicado al nombre de la estructura o a cualquiera de las variables declaradas de ese tipo.

12.2.2.Inicialización.

Las reglas de inicialización a cero por defecto de los modos extern y static se mantienen. Por lo demás, una estructura puede inicializarse en el momento de la definición de modo análogo a como se inicializan los vectores y matrices, por medio de valores encerrados entre llaves **{}**. Por ejemplo, una forma de declarar e inicializar a la vez la estructura *alumno_nuevo* podría ser la siguiente:

```
struct alumno {
    char nombre[31];
    char direccion[21];
    unsigned long no_matricula;
    unsigned long telefono;
    float notas[10];
} alumno_nuevo = {"Paco Merpan",
        "San Martín 87, 2º A",
        62419,
        921427011};
```

Como no se proporciona valor para las notas, éstas se inicializan a **cero**.

También es correcto la inicialización en el momento de la declaración:

```
struct alumno alumno1={“Marta”,
          “C/ Jon 13, 1º-C”,
          87675,
          921427011};
```

12.2.3.Utilización.

Para acceder a los **miembros** de una estructura se utiliza el **operador punto (.)**, precedido por el nombre de la estructura y seguido del nombre del miembro. Por ejemplo, en la estructura anidada definida anteriormente, para dar el **valor** 903456 al *telefono* del *alumno alumno1*, se escribirá:

```
alumno1.telefono = 903456;
```

Para guardar la dirección de este mismo

alumno, se escribirá:

```
alumno1.direccion = "C/ Penny Lane 1,2-
A";
```

Para dar nombre y apellidos:

```
alumno1.nom.nom=”Carlos”;
alumno.nom.ap1=”Gomez”;
alumno.nom.ap2=”Pedrez”;
```

El tipo de estructura creado se puede utilizar para definir más variables o estructuras de tipo alumno, así como **arrays de estructuras** de este tipo. Por ejemplo:

```
struct alumno nuevo_alumno, clase[300];
```

En este caso, *nuevo_alumno* es una estructura de tipo alumno, y *clase[300]* es un array de estructuras con espacio para almacenar los datos de 300 alumnos. El número de matrícula del **alumno 264** podrá ser accedido como *clase[264].no_matricula*.

Además, las estructuras permiten ciertas operaciones globales que no se pueden realizar con arrays. Por ejemplo, la sentencia siguiente:

```
clase[298] = nuevo_alumno;
```

Hace que se copien **todos los miembros** de la estructura *nuevo_alumno* en los miembros correspondientes de la estructura *clase[298]*. Estas operaciones globales **no son posibles con arrays**.

12.3. Uniones.

Las uniones son a primera vista, entidades muy similares a las estructuras, están formadas por un número cualquiera de miembros, al igual que aquellas, pero en éste caso **no existen simultáneamente** todos los miembros, y sólo uno de ellos tendrá un valor válido en un momento determinado.

Cuando se declara una variable de tipo *union*, se le asigna una porción de memoria que es **compartida** por variables diferentes en distintos momentos.

La definición y declaración de uniones es igual que en el caso de estructuras, cambiando únicamente la palabra reservada ***union*** en lugar de ***struct***.

Supongamos que queremos guardar datos para un stock de materiales, pero los mismos pueden ser identificados, en un caso con el número de articulo (un entero) y en otro por su nombre (un string de 10 letras como máximo).

No tendría sentido definir dos variables, un int y un string, para cada artículo, ya que voy a usar una modalidad ú la otra, pero **no las dos simultáneamente**. Las uniones resuelven este caso, ya que si declaro una que contenga dos miembros, un entero y un string, sólo se **reservará lugar para el mayor de ellos**, en este caso, el string, de tal forma que si asigno un valor a éste se llenará ese lugar de la memoria con los caracteres correspondientes, pero si en cambio asigno un valor al miembro declarado como *int* éste se guardará en los dos primeros bytes del MISMO lugar de memoria. Por supuesto, en una unión, sólo uno de los miembros tendrá entonces un valor correcto.

```
union stock {
    int num;
    char nom[11];
    }material[100];
```

En este caso se declara un **array** de uniones de tipo *stock* que ocupará las posiciones de memoria necesarias para almacenar *nom[]*, utilizándose solo las primeras cuando utilicemos *num*.

Las uniones tienen más sentido cuando forman parte de una estructura que conforme una base de datos.

12.4.Ejemplo de estructuras y uniones.

Ejemplo de definición de una estructura simple para definir una base de datos de un almacén de prendas deportivas:

```
union tama{
    int numero;
    char letra;
    char siglas[4];
    }
```

```
enum col {negro, marron,rojo, naranja,
amarillo, verde, azul, violeta, gris,
blanco};

struct prenda{
    char articulo[15];
    int cantidad;
    union tama talla;
    enum col color;
    }prendas[200];
```

12.5.Campos de bits.

Los campos de bits son elementos especiales **dentro de una estructura**, en los que se puede definir su **tamaño en bits**.

La definición de un campo de bits tiene la siguiente forma:

```
Tipo_dato nombre_campo:longitud;
```

Siendo ***Tipo_dato*** un tipo **entero** válido(*int, signed int o unsigned int*), ***nombre_campo*** es el identificador del campo de bits y ***longitud*** es un número entero positivo que indica el **número de bits** que se asignan a este campo.

Los campos de bits se almacenan en memoria de manera continua, siempre que haya espacio en al celda de memoria; en caso contrario, el campo de bit comenzará en la siguiente celda.

La ordenación en memoria de los bits que forman parte de los campos de bits, depende del hardware, de la CPU y fundamentalmente del compilador, lo que **impide saber de antemano la disposición física de los campos en memoria** (turbo C asigna los campos de bits de derecha a izquierda de acuerdo con el orden en que hayan sido declarados.

Puesto que los campos de bits no son más que elementos de una estructura, es posible mezclarlos con otros elementos que no lo sean.

Los campos de bits tienen las siguientes restricciones:

- No se permiten arrays de campos de bits.
- No se puede obtener la dirección de un campo de bit.
- Un campo de bit no puede sobrepasar el tamaño físico de un entero en memoria.
- Ninguna función puede devolver un campo de bit.

Por ejemplo, con la declaración:

```
ctruct campo_bit{
    int entero;
    unsigned dos:2;
    unsigned seis:6;
    char letra;
    }cosas;
```

Se declara la variable *cosas* de tipo *estruc campo_bit* compuesta por un **entero**, un campo de **2 bits**, un campo de **6 bits** y un **carácter**, cuyos identificadores respectivos son: *entero, dos, seis y letra*.

La asignación de valores se realiza igual que para los elementos normales de la estructura:

```
cosas.entero=1000;
cosas.seis=16;
```

Los campos de bits facilitan las operaciones a nivel de bits y permiten el almacenamiento de las variables lógicas de tamaños distintos.

Sin embargo, el uso de estructuras con cam-

pos de bits supone la inclusión de mayor número de operaciones en la CPU (**mayor tiempo de ejecución**) y puede no conseguirse el pretendido ahorro de memoria debido al mecanismo de almacenamiento de los campos.

12.6.Tipos definibles.

La palabra reservada del lenguaje C ***typedef*** sirve para la creación de nuevos nombres de tipos de datos. Mediante esta declaración es posible que el usuario defina una serie de tipos de variables propios, no incorporados en el lenguaje y que **se forman a partir de tipos de datos ya existentes**. Por ejemplo, la declaración:

```
typedef int ENTERO;
```

Define un tipo de variable llamado *ENTERO* que corresponde a *int*.

Como ejemplo más completo, se pueden declarar mediante *typedef* las siguientes estructuras:

```
#define MAX_NOM 30
#define MAX_ALUMNOS 400

/* se define la estructura s_alumno*/
struct s_alumno {
char nombre[MAX_NOM];
short edad;
};

typedef struct s_alumno ALUMNO;
/* ALUMNO es un nuevo tipo de variable*/

struct clase {
ALUMNO alumnos[MAX_ALUMNOS];
char nom_profesor[MAX_NOM];
};

typedef struct clase CLASE;
```

Con esta definición se crean las dos palabras reservadas para tipos, denominadas ***ALUMNO*** y ***CLASE***.

Ahora podría declararse variables del siguiente modo:

```
ALUMNO alumno_nuevo;
CLASE 1ES;
```

Y trabajar normalmente con las variables:

```
alumno_nuevo.nombre=”Juan”;
1ES.alumnos[10].nombre=alumno_nuevo;
```

El comando ***typedef*** ayuda a parametrizar un programa contra problemas de portabilidad. Generalmente se utiliza *typedef* para los tipos de datos que pueden ser dependientes de la instalación. También puede ayudar a documentar el programa (es mucho más claro para el programador el tipo CLASE, que un tipo declarado como un una estructura complicada), haciéndolo más legible.

12.7.Ejemplos.

De estructuras:

```
/* libro.c -- inventario de un solo libro */
#include <stdio.h>
#define MAXTIT 41       /*longitud máxima de titulo + 1 */
#define MAXAUT 31    /* longitud máxima de autor + 1 */

/* nuestro primer patrón de estructura, la etiqueta es biblio */
struct biblio {
    char titulo[MAXTIT];  /*tira de caracteres para título*/
    char autor [MAXAUT];  /*tira de caracteres para autor */
    float precio;         /* variable para precio de libro */
    };
```

```
int main(void)
{
    struct biblio libro;    /* declara  libro  de tipo biblio */

8   printf("Introduzca título del libro.\n");
    gets(libro.titulo);       /* accede al campo título */

    printf("Introduzca ahora el autor.\n");
    gets(libro.autor);

    printf("Ahora ponga el precio.\n");
    scanf("%f", &libro.precio);

    printf("%s por %s: %.2f pts.\n", libro.titulo,
    libro.autor, libro.precio);

    printf("%s: \"%s\" (%.2f euros.)\n",
    libro.autor,libro.titulo, libro.precio);

    return 0;
}
```

Si tienes más de un libro, dos o tres (hay algunos que tienen mas) no te preocupes, a continuación solucionamos el problema.

```
/* mucholib.c -- inventario de varios libros */

#include <stdio.h>
#define MAXTIT 40
#define MAXAUT 40
#define MAXLIB 100   /* número máximo de libros     */
#define STOP ""      /* tira nula, finaliza entrada */

struct biblio {    /* prepara patrón estructura   */
    char titulo[MAXTIT];
    char autor[MAXAUT];
    float precio;
    };
int main(void)
{
    struct biblio libro[MAXLIB];/*array de estructuras biblio*/
    int cont = 0;
    int indice;

    printf("Introduzca título del libro.\n");
    printf("Pulse[INTRO]a comienzo de línea para parar.\n");
    while (cont < MAXLIB &&   strcmp(gets(libro[cont].titulo),STOP)!=0)
    {
           printf("Introduzca ahora el autor.\n");
           gets(libro[cont].autor);
           printf("Ahora ponga el precio.\n");
           scanf("%f", &libro[cont++].precio);
           while (getchar() != '\n') continue;    /* limpia Buffer hasta
                                                              el retorno de
carro*/
           if (cont < MAXLIB)
                 printf("Introduzca el siguiente título.\n");
           else
```

```
                printf ("Ya hemos llenado la estantería. No me caben más\n");
    }
    printf("Ahora va su lista de libros:\n");
    for (indice = 0; indice < cont; indice++)
          printf("%s por %s: %.2f euros.\n",
                        libro[indice].titulo,libro[indice].autor,
                        libro[indice].precio);
    return 0;
}
```

Para aclarar el funcionamiento del programa anterior hay que tomar en consideración lo siguiente:

- Array de estructuras de tipo libro:

libro[0]:	**libro[0].titulo**	**libro[0].autor**	**libro[0].precio**
libro[1]:	**libro[1].titulo**	**libro[1].autor**	**libro[1].precio**
libro[2]:	**libro[2].titulo**	**libro[2].autor**	**libro[2].precio**
	|	**|**	**|**
	Array de char[40]	**Array de char[40]**	**float**

- Correcto uso de los índices:

```
libro[0]={      "Viaje al centro de la tierra",
                "Julio Verne",
                25.90
        }
```

libro[0].autor corresponde a "Julio Verne".

libro.autor[0] corresponde a 'J'.

- Bucle principal del programa:

```
while (cont < MAXLIB &&  strcmp(gets(libro[cont].titulo),STOP)!=0)
```

  - ***gets(libro[cont].titulo)*** lee una tira de caracteres como título del libro.
  - ***strcmp()*** compara la tira leída con ***STOP*** que es una tira vacía, si se pulsa [**INTRO**] al comienzo de la línea se transmite la tira vacía y se finaliza el bucle.

- Recordemos que *scanf()* ignora espacios, y caracteres de nueva línea. Cuando se responde a la pregunta del precio y se responde, por ejemplo: 12.50[intro]:

  En el buffer habrá: 12.50\n

  La función *scanf()* recoge 1, 2, ., 5 y 0 pero **deja el \n**.

  Para limpiar el buffer se puede utilizar el código siguiente:

```
    while (getchar() != '\n') continue;
```

```
/* amigo.c -- ejemplo de estructura anidada */

#include <stdio.h>
#define LON 20
#define M1 "     Gracias por esa tarde maravillosa, "
#define M2 "Me has demostrado que realmente un "
#define M3 "no es una persona corriente. A ver si"
#define M4 " quedamos frente a un delicioso plato de "
#define M5 "y pasamos otra buena velada."

struct nombre {               /* primer patrón de estructura */
    char nom[LON];
    char apell[LON];
    };

struct tio{                           /* segundo patrón     */
    struct nombre es;   /* estructura anidada */
    char comifavo[LON];
    char trabajo[LON];
    float gana;
    };

int main(void)
{
    static struct tio colega = {           /* inicializa variable */
           {"Pepe", "Gafe"},
           "alcachofas",
           "sexador de pollos",
           3535000.00
           };

    printf("Querido %s, \n\n", colega.es.nom);
    printf("%s%s.\n", M1, colega.es.nom);
    printf("%s%s\n", M2, colega.trabajo);
    printf("%s\n", M3);
    printf("%s%s\n%s\n\n", M4, colega.comifavo, M5);
    printf("%40s%s\n", " ", "Hasta pronto,");
    printf("%40s%s\n", " ", "Juanita");
    return 0;
}
```

La salida del programa sería:

```
Querido Pepe,

     Gracias por esa tarde maravillosa, Pepe.
Me has demostrado que realmente un sexador de pollos
no es una persona corriente. A ver si
 quedamos frente a un delicioso plato de alcachofas
y pasamos otra buena velada.

                                        Hasta pronto,
                                        Juanita
```

12.8.Ejercicios.

a) Realizar un programa que permita introducir la edad y calificaciones de alumnos, para obtener el listado por orden de edad y la media de las calificaciones.

b) Haz una base de datos sencilla que permita almacenar componentes electrónicas.

c) Diseña un programa de agenda electrónica de teléfonos y direcciones.

13.Punteros

13.1. Introducción.

El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una **dirección** (que se suele expresar con un número hexadecimal). El ordenador mantiene una **tabla de direcciones** que relaciona el nombre de cada variable con su dirección en la memoria.

Gracias a los nombres de las variables (**identificadores**), de ordinario no hace falta que el programador se preocupe de la dirección de memoria donde están almacenados sus datos. Sin embargo, en ciertas ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables.

El lenguaje C dispone del **operador dirección** (**&**) que permite determinar la dirección de una variable, y de un tipo especial de variables destinadas a contener direcciones de variables. Estas variables se llaman **punteros** o **apuntadores** (en inglés pointers).

Así pues, un **puntero** es una variable que puede contener la **dirección** de otra variable.

Por supuesto, los punteros están almacenados en algún lugar de la memoria y tienen su propia dirección (más adelante se verá que existen punteros a punteros).

Se dice que un puntero apunta a una variable si su contenido es la dirección de esa variable. Un puntero ocupa de ordinario **4 bytes** de memoria.

13.2. Declaración.

Un puntero se debe declarar o definir de acuerdo con el tipo del dato al que apunta. Por ejemplo, un puntero a una variable de tipo **int** se declara del siguiente modo:

```
int *direc;
```

Lo cual quiere decir que a partir de este momento, la variable ***direc*** podrá contener la dirección de cualquier variable **entera**. La regla nemotécnica es que el valor al que apunta **direc** (es decir ***direc**, como luego se verá), es de tipo *int*. Los punteros a *long, char, float* y *double* se definen análogamente a los punteros a *int*.

13.3. Inicialización y operadores de punteros.

Como se ha dicho, el lenguaje C dispone del **operador dirección** (**&**) que permite hallar la dirección de la variable a la que se aplica.

Un puntero es una verdadera variable, y por tanto puede cambiar de valor, es decir, puede cambiar la variable a la que apunta. Para acceder al valor depositado en la zona de memoria a la que apunta un puntero se debe utilizar el **operador indirección** (*). Por ejemplo, supóngase las siguientes declaraciones y sentencias:

```
int i, j, *p;     /* p es un puntero a
                  entero */
p = &i;           /* p contiene la
                   dirección de i*/
*p = 10;          /* i toma el valor 10
p = &j;           /* p contiene ahora
                   la dirección de j*/
*p = -2;          /* j toma el valor –2*/
```

Hay una serie de limitaciones de uso de los punteros:

- Las **constantes** y las **expresiones no tienen** dirección, por lo que no se les puede aplicar el operador (&).

- Tampoco puede cambiarse la dirección de una variable.
- Los valores posibles para un puntero son las direcciones posibles de memoria.
- Un puntero puede tener valor 0 (equivalente a la constante simbólica predefinida **NULL**).
- No se puede asignar una dirección absoluta directamente (habría que hacer un **casting**).

Las siguientes sentencias son ilegales:

```
p = &34;   /* las constantes no tienen
              dirección*/
p = &(i+1);       /* las expresiones no
tienen
              dirección*/
&i = p;    /* las direcciones no se
              pueden cambiar*/
```

```
p = 17654; /* habría que escribir
           p = (int *)17654;*/
```

Para imprimir punteros con la función **printf()** se deben utilizar los formatos **%u** y **%p**.

No se permiten asignaciones directas (sin **casting**) entre punteros que apuntan a distintos tipos de variables. Sin embargo, existe un tipo indefinido de punteros (**void** *, o **punteros a void**), que puede asignarse y al que puede asignarse cualquier tipo de puntero. Por ejemplo:

```
int *p;
double *q;
void *r;
p = q;            // ilegal
p = (int *)q;     // legal
p = r = q;        // legal
```

13.4. Aritmética de punteros.

Como ya se ha visto, los punteros son unas variables un poco especiales, ya que guardan información (no sólo de la dirección a la que apuntan), sino también del tipo de variable almacenado en esa dirección.

Esto implica que no van a estar permitidas las operaciones que no tienen sentido con direcciones de variables, como **multiplicar** o **dividir**, pero **sí** otras como **sumar** o **restar**. Además estas operaciones se realizan de un modo correcto, pero que no es el ordinario. Así, la sentencia:

```
p = p+1;
```

Hace que **p** apunte a la dirección **siguiente** de la que apuntaba, teniendo en cuenta el tipo de dato. Por ejemplo, si el valor apuntado por **p** es *short int* y ocupa **2 bytes**, el sumar 1 a **p** implica añadir 2 bytes a la dirección que contiene, mientras que si **p** apunta a un *double*, sumarle 1 implica añadirle **8 bytes**.

También tiene sentido la diferencia de punteros al mismo tipo de variable. El resultado es la **distancia entre las direcciones** de las variables apuntadas por ellos, no en bytes sino en **datos de ese mismo tipo**. Las siguientes expresiones tienen pleno sentido en C:

```
p = p + 1;
p = p + i;
p += 1;
p++;
```

El siguiente ejemplo ilustra la aritmética de punteros:

```
/*aripunt.c -- ejemplo de aritmética de
punteros*/
#include <stdio.h>

int main(void)
{
int a, b, c;
int *p1, *p2;
void *p;
p1 = &a;          /*La dirección de a
                   es asignada a p1*/
*p1 = 1;          /*Equivale a a = 1;*/
p2 = &b;          /*La dirección de b
                   es asignada a p2*/
*p2 = 2;          /*Equivale a b = 2;*/
```

```
p1 = p2;          /*El valor del p1=p2*/
*p1 = 0;          /*b = 0*/
p2 = &c;          /*La dirección de c
                   es asignada a p2*/
*p2 = 3;          /*c = 3*/
printf("%d %d %d\n", a, b, c);
                  /*¿Qué se imprime?*/
p = &p1;          /*p contiene la
                   dirección de p1*/
p = p1;           /*p= p1;*/
p2=p;
*p2=7;
printf("%d %d %d\n", a, b, c);
                  /*¿Qué se imprime?*/
return 0;
}
```

Supóngase que en el momento de comenzar la ejecución, las direcciones de memoria de las distintas variables son las mostradas en la Tabla.

Variable	Dirección de memoria
a	00FA0000
b	00FA0002
c	00FA0004
p1	00FA0006
p2	00FA000A
p	00FA000E

La dirección de memoria está en hexadecimal; basta prestar atención al último de estos números.

La siguiente tabla muestra los valores de las variables en la ejecución del programa paso a paso. Se muestran en negrita los cambios entre paso y paso. Es importante analizar y entender los cambios de valor.

Paso	**a** 00FA0000	**b** 00FA0002	**c** 00FA0004	**p1** 00FA0006	**p2** 00FA000A	**P** 00FA000E
1				**00FA0000**		
2	**1**			00FA:0000		
3	1			00FA:0000	**000FA0002**	
4	1	**2**		00FA:0000	000FA0002	
5	1	2		**000FA0002**	000FA0002	
6	1	**0**		000FA0002	000FA0002	
7	1	0		000FA0002	**000FA0004**	
8	1	0	**3**	000FA0002	000FA0004	
9	1	0	3	000FA0002	000FA0004	
10	1	0	3	000FA0002	000FA0004	**000FA0006**
11	1	0	3	000FA0002	000FA0004	**000FA0002**
12	1	0	3	000FA0002	**000FA0002**	000FA0002
13	1	**7**	3	000FA0002	000FA0002	000FA0002

13.5. Tipos de punteros.

13.5.1. Punteros genéricos.

Es un puntero de tipo **void** que, posteriormente, puede apuntar a **cualquier tipo** de dato. Su declaración será:

```
void *nombre_puntero;
```

13.5.2. Puntero nulo.

Es una variable puntero que no apunta a ningún sitio y que se inicializa expresamente como tal:

```
Tipo_dato *nombre_puntero=NULL;
```

Donde NULL es una constante definida en stdio.h, que tiene valor cero.

Mientras un puntero apunta a un dato válido no tiene valor cero, utilizándose el puntero nulo para indicar **situaciones de error**.

13.5.3. Punteros constantes.

Un puntero constante es aquel que apunta siempre a la misma dirección de memoria y se declara expresamente como tal:

```
Tipo_dato *const nombre_puntero;
```

El puntero es constante, pero el dato apuntado puede cambiar a lo largo del programa.

Si lo que se quiere declarar como constante es el dato apuntado, la declaración es como sigue:

```
const tipo_dato *nombre_puntero;
```

Para declarar como constantes tanto el dato como el puntero, debe emplearse la siguiente sintaxis:

```
const tipo_dato *const nombre_puntero;
```

13.6. Punteros y arrays.

13.6.1. Relación entre punteros y arrays.

Existe una relación muy estrecha entre los arrays y los punteros. De hecho, el **nombre de un array es un puntero** (un puntero **constante**, en el sentido de que no puede apuntar a otra variable distinta de aquélla a la que apunta) a la dirección de memoria que contiene el primer elemento del array.

Supónganse las siguientes declaraciones y sentencias:

```
double vect[10];        /* vect es un
puntero
                  a vect[0] */
double *p;
...
p = &vect[0];     / p = vect;*/
...
```

El identificador ***vect***, es decir el nombre del array, es un puntero al **primer elemento** del array ***vect[ ]***. Esto es lo mismo que decir que el valor de ***vect*** es ***&vect[0]***. Existen más puntos de coincidencia entre los arrays y los punteros:

Puesto que el nombre de un array es un puntero, obedecerá las leyes de la aritmética de punteros. Por tanto, si ***vect*** apunta a ***vect[0]***, (***vect+1***) apuntará a ***vect[1]***, y ***(vect+i)*** apuntará a ***vect[i]***.

Recíprocamente (y esto resulta quizás más sorprendente), a los punteros se les pueden poner subíndices, igual que a los arrays. Así pues, si ***p*** apunta a ***vect[0]*** se puede escribir:

```
p[3]=p[2]*2.0;     /* equivalente a
                   vect[3]=vect[2]*2.0;*/
```

Si se supone que ***p==vect***, la relación entre punteros y arrays puede resumirse como se indica en las líneas siguientes:

***p equivale a vect[0], a *vect y a p[0]**

***(p+1) equivale a vect[1], a *(vect+1) y p[1]**

***(p+2) equivale a vect[2], a *(vect+2) y a p[2]**

Como ejemplo de la relación entre arrays y punteros, se van a ver varias formas posibles para sumar los N elementos de un array **a[ ]**. Supóngase la siguiente declaración y las siguientes sentencias:

```
int a[N], suma, i, *p;

/****      Forma 1****/
for(i=0, suma=0; i<N; ++i)
    suma += a[i];

/****      Forma 2 ****/
for(i=0, suma=0; i<N; ++i)
    suma += *(a+i);

/**** Forma 3 ****/
for(p=a, i=0, suma=0; i<N; ++i)
    suma += p[i];

/**** Forma 4 ****/
for(p=a, suma=0; p<&a[N]; ++p)
    suma += *p;
```

13.6.2. Relación entre punteros y matrices.

En el caso de las matrices la relación con los punteros es un poco más complicada. Supóngase una declaración como la siguiente:

```
int mat[5][3], **p, *q;
```

El nombre de la matriz (***mat***) es un puntero al primer elemento de un **array de punteros** ***mat[ ]*** (por tanto, existe un array de punteros que tiene también el mismo nombre que la matriz), cuyos elementos contienen las **direcciones** del primer elemento de cada fila de la matriz. El nombre ***mat*** es pues un puntero a puntero.

El array de punteros ***mat[ ]*** se crea automáticamente al crearse la matriz. Así pues, ***mat*** es igual a ***&mat[0]***; y ***mat[0]*** es ***&mat[0][0]***. Análogamente, ***mat[1]*** es *&**mat[1][0]***, ***mat[2]*** es ***&mat[2][0]***, etc. La dirección base sobre la que se direccionan todos los elementos de la matriz **no es** ***mat***, sino ***&mat[0][0]***.

Recuérdese también que, por la relación entre vectores y punteros, (***mat+i***) apunta a ***mat[i]***.

Recuérdese que la fórmula de direccionamiento de una matriz de N filas y M columnas establece que la dirección del elemento (i, j) viene dada por: **dirección (i, j) = dirección (0, 0) + i*M + j.**

Teniendo esto en cuenta y haciendo ****p = mat;** se tendrán las siguientes formas de acceder a los elementos de la matriz:

***p es el valor de mat[0]**

****p es mat[0][0]**

***(p+1) es el valor de mat[1]**

****(p+1) es mat[1][0]**

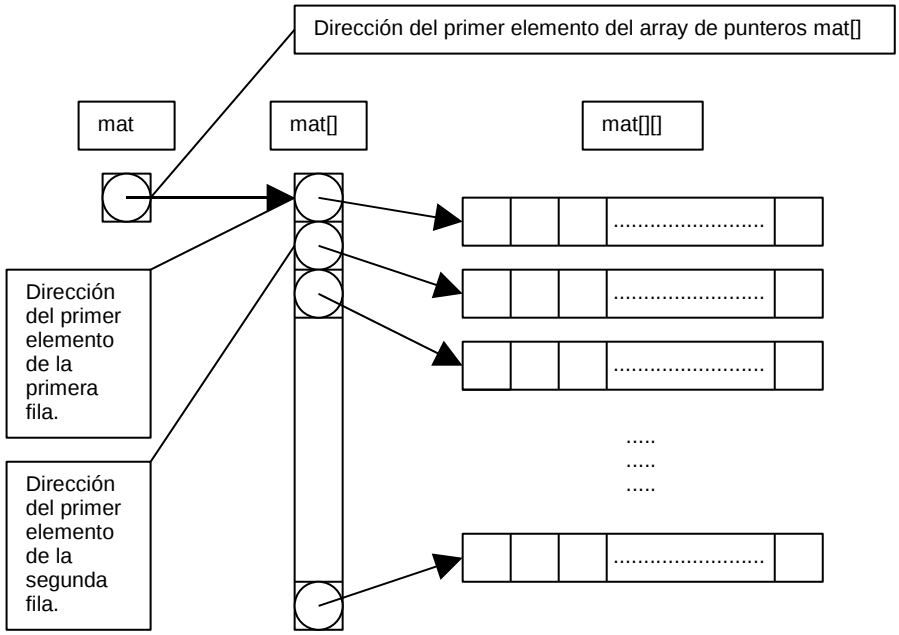
***(*(p+1)+1) es mat[1][1]**

Por otra parte, si la matriz tiene M columnas y si se hace ***q = &mat[0][0]*** (dirección base de la matriz. Recuérdese que esto es diferente del caso anterior ***p = mat***), el elemento ***mat[i][j]*** puede ser accedido de varias formas. Basta recordar que dicho elemento tiene por delante **i** filas completas, y **j** elementos de su fila:

```
*(q + M*i + j)   /* fórmula de
                    direccionamiento*/
*(mat[i] + j)    /* primer elemento
                    fila i desplazado
                    j elementos*/
(*(mat + i))[j]  /* [j] equivale a sumar
                    j a un puntero*/
*((*(mat + i)) + j)
```

Todas estas relaciones tienen una gran importancia, pues implican una correcta comprensión de los punteros y de las matrices. De todas formas, hay que indicar que las matrices no son del todo idénticas a los arrays de punteros: Si se define una matriz explícitamente por medio de arrays de punteros, las filas pueden tener diferente número de

elementos, y no queda garantizado que estén contiguas en la memoria (aunque se puede hacer que sí lo sean). No sería pues posible en este caso utilizar la fórmula de direccionamiento y el acceder por columnas a los elementos de la matriz. La Figura resume gráficamente la relación entre matrices y vectores de punteros.



13.7. Puntero a puntero.

Un puntero a puntero supone una indirección múltiple, de modo que el primer puntero contiene la dirección del segundo puntero, el cual apunta al dato deseado. Su declaración tiene la siguiente forma:

```
Tipo_dato **nombre_puntero;
```

Siendo ***nombre_puntero*** el que contiene la dirección de ***nombre_puntero** que, a su vez, contiene la dirección de ****nombre_puntero**, que es realmente el dato del tipo expresado.

Recordando lo expresado sobre arrays de punteros, un puntero a puntero puede entenderse como el equivalente a un puntero a un array de punteros, lo que nos permite direccionar un array bidimensional, siempre que cada puntero esté asociado a su correspondiente array.

13.8. Punteros a estructuras y uniones.

Se pueden definir también **punteros a estructuras**:

```
struct alumno *pt;
pt = &nuevo_alumno;
```

Ahora, el puntero ***pt*** apunta a la estructura ***nuevo_alumno*** y esto permite una nueva forma de acceder a sus miembros utilizando el **operador flecha** (->), constituido por los signos (-) y (>).

Al declarar un puntero de este tipo no se crea la estructura o la unión, sólo se crea la variable puntero. El tipo de dato (estructura o unión) **debe estar ya creado**.

Así, para acceder al teléfono del alumno

nuevo_alumno, se puede utilizar cualquiera de las siguientes sentencias:

```
pt->telefono;
(*pt).telefono;
```

Donde el paréntesis es necesario por la mayor prioridad del operador (.) respecto a (*).

A continuación se muestra una parte del código necesario para iniciar las fichas de un array de estructuras que permiten almacenar el nombre y la edad de un número de individuos determinado por la constante simbólica MAX:

```
struct ficha{
    char nombre[25];
    int edad;
    };
struct ficha lista[MAX], *punt;
for(i=0;i<MAX;i++)
    {
    punt=lista+I;
    puts("Nombre:?");
    gets(punt->nombre);
    puts("Edad:?");
    scanf("%d",&punt->edad);
    }
```

13.9. Ejemplos.

```
/* ptr_oper.c -- operaciones con punteros */
#include <stdio.h>
int main(void)
{
    static int urn[3] = {100,200,300};
    int *punt1, *punt2;

    punt1 = urn;    /* asigna una dirección al puntero */
    punt2 = &urn[2];     /* idem */

    printf("punt1 = %p, *punt1 =%d, &punt1 = %p\n",
                                    punt1, *punt1, &punt1);

    punt1++;                 /* incrementa el puntero */

    printf("punt1 = %p, *punt1 =%d, &punt1 = %p\n",
                                    punt1, *punt1, &punt1);

    printf("punt2 = %p, *punt2 =%d, &punt2 = %p\n",
                                    punt2, *punt2, &punt2);

    ++punt2;             /* sobrepasa el array */

    printf("punt2 = %p, *punt2 =%d, &punt2 = %p\n",
                                    punt2, *punt2, &punt2);
    /* substracci¢n de punteros */

    printf("punt2 - punt1 = %p\n", punt2 - punt1);

    return 0;
}
```

El resultado será:

```
punt1 = 0x8049828, *punt1 =100, &punt1 = 0xbffff0b4
punt1 = 0x804982c, *punt1 =200, &punt1 = 0xbffff0b4
punt2 = 0x8049830, *punt2 =300, &punt2 = 0xbffff0b0
punt2 = 0x8049834, *punt2 =0, &punt2 = 0xbffff0b0
punt2 - punt1 = 2
```

```
//palabras.c
//Obtención de palabras de una tira de caracteres

#include<stdio.h>
#define MAX 80
int main(void)
{
    char cad[MAX],token[MAX],*p,*q;

    puts("Introduce una frase");
    gets(cad);

    p=cad; //Hace que p apunte al comienzo de la cadena.

    while(*p)    //Mientras el carácter no sea \0.
    {
           q=token;     //q apunta al principio de la copia.

           while(*p!=' ' && *p)//Mientras no sea espacio ni /0.
           {
                  *q=*p;       //Copia el carácter en token[].

                  q++;p++;     //Siguiente.
           }

           if (*p) p++; //Si es espacio se lo salta
                  *q='\0';            //Y finaliza la tira de token[].
           puts(token);
    }
    return 0;
}
```

El resultado sería:

```
Introduce una frase
Serafina caracola vive en su casa ella sola
Serafina
caracola
vive
en
su
casa
ella
sola
```

Nos complicamos la vida:

```
/* puntarr.c Punteros y arrays */
#include <stdio.h>
int main(void)
{
    int datos[4][2]={   00,01,
                                    10,11,
                                    20,21,
                                    30,31};
    system ("clear");

    printf("datos=%p, datos[0]=%p,&datos[0]0]=%p\n",
                        datos, datos[0], &datos[0][0]);
```

```
    printf("**datos=%d,*datos[0]=%d\n",
                        **datos, *datos[0]);

    printf("**(datos+1)=%d,*(datos[0]+1)=%d\n",
                        **(datos+1), *(datos[0]+1));

    printf("**(datos+2)=%d,*(datos[0]+2)=%d\n",
                        **(datos+2), *(datos[0]+2));

    printf("**(datos+3)=%d,*(datos[0]+3)=%d\n",
                        **(datos+3), *(datos[0]+3));
    return 0;

}
```

La salida sería:

```
datos=0xbffff090, datos[0]=0xbffff090,&datos[0]0]=0xbffff090
**datos=0,*datos[0]=0
**(datos+1)=10,*(datos[0]+1)=1
**(datos+2)=20,*(datos[0]+2)=10
**(datos+3)=30,*(datos[0]+3)=11
```

```
//fortuna.c
//La buena fortuna
#include <stdio.h>
#include <time.h>             //para la función time()
#include <stdlib.h>           //para rand() y srand()

char *fortuna[]={       //Array de punteros a cadenas de char
    "Conseguiras algo de dinero",
    "Entrará el amor en tu vida",
    "Tendrás una vida larga y próspera",
    "Es un buen momento para invertir",
    "Aprobarás C facilmente"
                        };


int main(void)
{
    int eleccion;

    puts("Para ver tu fortuna pulsa una tecla");

    srand (time(NULL)); //se llama a la función srand()
                                    //para que no genere siempre el
                                    //mismo número.
                                    //La finción time() devuelve el tiempo
                                    //transcurrido, medido en segundos, desde
                                    //'la Época': las 0 horas 0 minutos 0 segundos,
                                    //tiempo universal coordinado, del 1 de enero
                                    //de 1970, por lo tanto la semilla siempre será
                                    //distinta.
    puts("");
    eleccion=rand();
    eleccion%=5;              //Número entre 0 y 4.
    puts(fortuna[eleccion]);
    return 0;
}
```

Ordenación de tiras sin strcpy(), tarda menos:

```
/* ordetira.c -- lectura y clasificación de tiras */
#include <stdio.h>
#include <string.h>

#define TAM 81     /*límite longitud tira contando \0 */
#define LIM 20     /* número máximo de tiras a leer */
#define PARA ""    /* tira nula para detener entrada */

int main(void)
{
    static char entra[LIM][TAM];  /* array para entrada */
    char *ptira[LIM];     /* array de variables puntero */
    int ct = 0;           /* contador de entrada */
    int k;                /* contador de salida */
    char *temp;           /* puntero auxiliar */
    int tope, busca;

    printf("Introduzca hasta %d líneas y las ordenaré.\n", LIM);
    printf("Para acabar, pulse[Intro]\n");

          /* Mientras no nos pasemos del límite de líneas*/
          /*La entrada sea válida*/
          /*Y no sea una línea en blanco*/

    while (ct < LIM && gets(entra[ct]) != NULL &&
                        strcmp(entra[ct], PARA) != 0)
    {
          ptira[ct]=entra[ct];/* asigna punteros a tiras */
          ct++;        /* contador de líneas*/
    }
                /* clasificador de tiras por punteros */
                /*Utilizamos ptira para ordenar los punteros,*/
                /*¡¡no se ordenan las tiras!!.*/
    for (tope = 0; tope <ct - 1; tope++)
          for (busca = tope + 1; busca < ct; busca++)
                if (strcmp(ptira[tope],ptira[busca]) > 0)
                {
                      temp = ptira[tope];
                      ptira[tope] = ptira[busca];
                      ptira[busca] = temp;
                }
    puts("Ahí va la lista ordenada:\n");
    for (k=0; k < ct; k++)
          puts(ptira[k]);              /* tiras ordenadas */
    puts("Y desordenada:\n");
    for (k=0; k < ct; k++)
          puts(entra[k]);              /* tiras desordenadas */
    return 0;
}
```

```
/* puntstr.c -- puntero a estructura */
#include <stdio.h>
#define LEN 20
struct datos {
             char nom[LEN];
             char apell[LEN];
             };

struct tio    {
           struct datos nombre;
```

```
            char comifavo[LEN];
            char trabajo[LEN];
            float gana;
                  };
int main(void)
{
    int i;
    static struct tio feten[2] = {
                                                        {{"Pepe", "Gafe"},
                                                        "alcachofas",
                                                        "sexador de pollos",
                                                        3535000.00},
                                                        {{"Santi", "Fever"},
                                                        "salmón ahumado",
                                                        "programador",
                                                        9999995.00}
                                                    };
    struct tio *este;          /* AQUI ESTA: puntero a estructura */
    printf("dir 1: %p; 2: %p\n", &feten[0], &feten[1] );
    este = &feten[0];  /* indica al puntero dónde apuntar */
    printf("puntero   1: %p; 2: %p\n", este, este + 1);
    for(i=0;i<2;este++,i++)
    {
           puts("");
           printf("A %s %s ",este->nombre.nom,este->nombre.apell);
           printf("le gusta comer %s\n",este->comifavo);
           printf("y gana %.2f euros en su trabajo de %s.\n",
           este->gana,este->trabajo);
    }
    return 0;
}
```

13.10. Ejercicios.

a)Realiza un programa que permita observar cuantas unidades se incrementan o decrementan las direcciones contenidas en punteros, para distintos tipos de datos.

b)Crea un programa que extraiga palabras de una tira y las muestre en pantalla en orden inverso de aparición en la tira (con punteros).

c)Escribe un programa que indique el número de veces que aparece un determinado carácter en una cadena utilizando punteros.

d)Elabora un programa que presente distintos mensajes en función de una variable entera, utilizando un array de punteros a cadenas de caracteres.

14. Funciones.

14.1. Introducción.

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener decenas y aún cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en sistemas poco manejables si no fuera por la **modularización**, que es el proceso consistente en dividir un programa muy grande en una serie de módulos mucho más pequeños y manejables.

A estos módulos se les suele denominar de distintas formas (subprogramas, subrutinas, procedimientos, funciones, etc.) según los distintos lenguajes.

El lenguaje C hace uso del concepto de función (function). Sea cual sea la nomenclatura, la idea es sin embargo siempre la misma: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal; éstas a su vez llaman a otras funciones más específicas y así sucesivamente.

La división de un programa en unidades más pequeñas o funciones presenta –entre

otras– las ventajas siguientes:

- **Modularización**. Cada función tiene una misión muy concreta, de modo que nunca tiene un número de líneas excesivo y siempre se mantiene dentro de un tamaño manejable.
- Además, una misma función (por ejemplo, un producto de matrices, una resolución de un sistema de ecuaciones lineales, ...) puede ser llamada muchas veces en un mismo programa, e incluso puede ser reutilizada por otros programas. Cada función puede ser desarrollada y comprobada por separado.
- **Ahorro de memoria y tiempo de desarrollo**. En la medida en que una misma función es utilizada muchas veces, el número total de líneas de código del programa disminuye, y también lo hace la probabilidad de introducir errores en el programa.
- **Independencia de datos y ocultación de información**. Una de las fuentes más comunes de errores en los programas de computador son los efectos colaterales o perturbaciones que se pueden producir entre distintas partes del programa. Es muy frecuente que al hacer una modificación para añadir una funcionalidad o corregir un error, se introduzcan nuevos errores en partes del programa que antes funcionaban correctamente. Una función es capaz de mantener una gran independencia con el resto del programa, manteniendo sus propios datos y definiendo muy claramente la interfaz o comunicación con la función que la ha llamado y con las funciones a las que llama, y no teniendo ninguna posibilidad de acceso a la información que no le compete.

Las funciones de C están implementadas con un particular cuidado y riqueza, constituyendo uno de los aspectos más potentes del lenguaje. Es muy importante entender bien su funcionamiento y sus posibilidades.

14.2.Nombre, valor de retorno y argumentos de una función.

Una función de C es una porción de código o programa que realiza una determinada tarea. Una función está asociada con un identificador o nombre, que se utiliza para referirse a ella desde el resto del programa. En toda función utilizada en C hay que distinguir entre su **definición**, su **declaración**

y su **llamada**. Para explicar estos conceptos hay que introducir los conceptos de **valor de retorno** y de **argumentos**.

Quizás lo mejor sea empezar por el concepto más próximo al usuario, que es el concepto de **llamada**. Las funciones en C se llaman incluyendo su nombre, seguido de los argumentos, en una sentencia del programa principal o de otra función de rango superior.

Los **argumentos** son datos que se envían a la función incluyéndolos entre paréntesis a continuación del nombre, separados por comas. Por ejemplo, supóngase una función llamada *power* que calcula x elevado a y. Una forma de llamar a esta función es escribir la siguiente sentencia:

```
power(x,y);
```

En este ejemplo *power* es el nombre de la función, **y x e y son los argumentos**, que en este caso constituyen los datos necesarios para calcular el resultado deseado.

¿Qué pasa con el resultado? ¿Dónde aparece? Pues en el ejemplo anterior el resultado es el **valor de retorno** de la función, que está disponible pero no se utiliza.

En efecto, el resultado de la llamada a *power* está disponible, pues **aparece sustituyendo al nombre de la función en el mismo lugar donde se ha hecho la llamada**; en el ejemplo anterior, el resultado aparece, pero no se hace nada con él. A este mecanismo de sustitución de la llamada por el resultado es a lo que se llama valor de retorno. Otra forma de llamar a esta función utilizando el resultado podría ser la siguiente:

```
distancia = power(x+3, y)*escala;
```

En este caso el primer argumento (*x+3*) es elevado al segundo argumento y, el resultado de la potencia (el valor de retorno) es multiplicado por escala, y este nuevo resultado se almacena en la posición de memoria asociada con el identificador distancia.

Este ejemplo resulta típico de lo que es una instrucción o sentencia que incluye una llamada a una función en el lenguaje C.

Para poder llamar a una función es necesario que en algún otro lado, en el mismo o en algún otro fichero fuente, aparezca l**a definición** de dicha función, que en el ejemplo anterior es la función *power*. **La definición de una función es ni más ni menos que el conjunto de sentencias o instrucciones necesarias para que la función pueda realizar su tarea cuando sea llamada**. En otras palabras, la definición es el código correspondiente a la función. Además del código, la definición de la función incluye la definición del tipo del valor de retorno y de cada uno de los argumentos. A continuación se presenta un ejemplo (incompleto) de cómo podría ser la definición de la función *power* utilizada en el ejemplo anterior.

```
double power(double base, double
exponente)
{
double resultado;
...
resultado = ... ;
return resultado;
}
```

La primera palabra *double* indica el tipo del valor de retorno. Esto quiere decir que el resultado de la función será un número de punto flotante de doble precisión.

Después viene el nombre de la función seguido de (entre paréntesis) la definición de los argumentos y de sus tipos respectivos. En este caso hay dos argumentos, base y exponente, que son ambos de tipo *double*.

A continuación se abren las llaves que contienen el código de la función.

La primera sentencia declara la variable resultado, que es también de tipo *double*.

Después vendrían las sentencias necesarias para calcular resultado como base elevado a exponente.

Finalmente, con la sentencia *return* se de-

vuelve resultado al programa o función que ha llamado a *power*.

Conviene notar que las variables *base* y *exponente* han sido declaradas en la **cabecera** (primera línea) de la definición, y por tanto ya no hace falta declararlas después, como se ha hecho con resultado.

Cuando la función es llamada, las variables *base* y *exponente* reciben sendas copias de los valores del primer y segundo argumento que siguen al nombre de la función en la llamada.

Además de la llamada y la definición, está también la **declaración** de la función. Ya se verá más adelante dónde se puede realizar esta declaración. La declaración de una función se puede realizar por medio de la primera línea de la definición, de la que pueden suprimirse los nombres de los argumentos, pero no sus tipos; **al final debe incluirse el punto y coma (;)**.

Por ejemplo, la función *power* se puede declarar en otra función que la va a llamar incluyendo la línea siguiente:

```
double power(double, double);
```

La declaración de una función permite que el compilador chequee el número y tipo de los argumentos, así como el tipo del valor de retorno. La declaración de la función se conoce también con el nombre de **prototipo de la función**.

14.3.Definición de una función.

La **definición de una función** consiste en la definición del código necesario para que ésta realice las tareas para las que ha sido prevista. La definición de una función se debe realizar en alguno de los ficheros que forman parte del programa. La forma general de la definición de una función es la siguiente:

```
tipo_valor_de_retorno
nombre_funcion(lista de argumentos con
tipos)
{
declaración de variables y/o de otras
funciones
codigo ejecutable
return (expresión); // optativo
}
```

La primera línea recibe el nombre de **encabezamiento (header)** y el resto de la definición (encerrado entre llaves) es el **cuerpo (body)** de la función.

Cada función puede disponer de sus propias variables, declaradas al comienzo de su código. Estas variables, por defecto, son de tipo auto, es decir, sólo **son visibles dentro del bloque en el que han sido definidas**, se crean cada vez que se ejecuta la función y permanecen ocultas para el resto del programa. Si estas variables de definen como *static*, conservan su valor entre distintas llamadas a la función.

También pueden hacerse visibles a la función variables globales definidas en otro fichero (o en el mismo fichero, si la definición está por debajo de donde se utilizan), declarándolas con la palabra clave *extern*.

El código ejecutable es el conjunto de instrucciones que deben ejecutarse cada vez que la función es llamada.

La **lista de argumentos** con tipos, también llamados **argumentos formales**, es una lista de declaraciones de variables, precedidas por su tipo correspondiente separadas por comas (,). Los argumentos formales son la forma más natural y directa para que la función reciba valores desde el programa que la llama, correspondiéndose en número y tipo con otra lista de argumentos en el programa que realiza la llamada a la función.

Los argumentos formales son declarados en el encabezamiento de la función, pero **no pueden ser inicializados en él**.

Cuando una función es ejecutada, puede devolver al programa que la ha llamado un valor (el

valor de retorno), cuyo tipo debe ser especificado en el encabezamiento de la función (si no se especifica, se supone por defecto el tipo *int*). Si no se desea que la función devuelva ningún valor, el tipo del valor de retorno deberá ser *void*.

La sentencia ***return*** permite devolver el control al programa que llama. Puede haber varias sentencias *return* en una misma función. Si no hay ningún *return*, el control se devuelve cuando se llega al final del cuerpo de la función. Le palabra clave *return* puede ir seguida de una expresión, en cuyo caso ésta es evaluada y el valor resultante devuelto al programa que llama como valor de retorno.

Los paréntesis que engloban a la expresión que sigue a *return* son optativos.

El valor de retorno es un valor único: **no puede ser un vector o una matriz**, aunque sí un puntero a un vector o a una matriz. Sin embargo, el valor de retorno **sí puede ser una estructura**, que a su vez puede contener vectores y matrices como elementos miembros.

Como ejemplo supóngase que se va a calcular a menudo el valor absoluto de variables de tipo *double*. Una solución es definir una función que reciba como argumento el valor de la variable y devuelva ese valor absoluto como valor de retorno. La definición de esta función podría ser como sigue:

```
double valor_abs(double x)
{
if (x < 0.0)
return -x;
else
return x;
}
```

14.4.Declaración a una función.

De la misma manera que en C es necesario declarar todas las variables, también toda función debe ser declarada antes de ser utilizada en la función o programa que realiza la llamada.

En C la declaración de una función se puede hacer de tres maneras:

- **Mediante una llamada a la función**. En efecto, cuando una función es llamada sin que previamente haya sido declarada o definida, esa llamada sirve como declaración suponiendo *int* como tipo del valor de retorno, y el tipo de los argumentos actuales como tipo de los argumentos formales. **Esta práctica es muy peligrosa** (es fuente de numerosos errores) y debe ser evitada.
- Mediante una **definición previa** de la función. Esta práctica es segura si la definición precede a la llamada, pero tiene el inconveniente de que si la definición se cambia de lugar, la propia llamada pasa a ser declaración como en el primer caso.
- Mediante una **declaración explícita, previa a la llamada**. Esta es la práctica más segura y la que hay que tratar de seguir siempre.

La declaración de la función se hace mediante el **prototipo de la función**, bien fuera de cualquier bloque, bien en la parte de declaraciones de un bloque.

C++ es un poco más restrictivo que C, y obliga a declarar explícitamente una función antes de llamarla.

La forma general del prototipo de una función es la siguiente:

```
tipo_valor_de_retorno
nombre_funcion(lista de tipos de
argumentos);
```

Esta forma general coincide sustancialmente con la primera línea de la definición (el encabezamiento), con dos pequeñas diferencias: en vez de la lista de argumentos formales o parámetros, en el prototipo basta incluir los tipos de dichos argumentos. Se pueden incluir también identificadores a

continuación de los tipos, pero son ignorados por el compilador. Además, una segunda diferencia es que **el prototipo termina con un carácter (;)**.

Cuando no hay argumentos formales, se pone entre los paréntesis la palabra *void*, y se pone también void precediendo al nombre de la función cuando no hay valor de retorno.

Los prototipos permiten que el compilador realice correctamente la conversión del tipo del valor de retorno, y de los argumentos actuales a los tipos de los argumentos formales.

La declaración de las funciones mediante los prototipos s**uele hacerse al comienzo del fichero**, después de los *#define* e *#include*. En muchos casos (particularmente en programas grandes, con muchos ficheros y muchas funciones), **se puede crear un fichero (con la extensión .h) con todos los prototipos de las funciones utilizadas en un programa**, e incluirlo con un *#include* en todos los ficheros en que se utilicen dichas funciones.

14.5.Llamada a una función.

La llamada a una función se hace **incluyendo su nombre en una expresión o sentencia** del programa principal o de otra función. Este nombre debe ir seguido de una lista de argumentos separados por comas y encerrados entre paréntesis.

A los argumentos incluidos en la llamada se les llama **argumentos actuales**, y pueden ser no sólo variables y/o constantes, sino también expresiones. Cuando el programa que llama encuentra el nombre de la función, evalúa los argumentos actuales contenidos en la llamada, los convierte si es necesario al tipo de los argumentos formales, y pasa **copias de dichos valores** a la función junto con el control de la ejecución.

El número de argumentos actuales en la llamada a una función debe coincidir con el número de argumentos formales en la definición y en la declaración.

Existe la posibilidad de definir funciones con un número variable o indeterminado de argumentos. Este número se concreta luego en el momento de llamarlas. Las funciones *printf()* y *scanf()*, son ejemplos de funciones con número variable de argumentos.

Cuando se llama a una función, después de realizar la conversión de los argumentos actuales, se ejecuta el código correspondiente a la función hasta que se llega a una sentencia *return* o al final del cuerpo de la función, y entonces se devuelve el control al programa que realizó la llamada, junto con el valor de retorno si es que existe. Recuérdese que el valor de retorno puede ser un valor numérico, una dirección (un puntero), o una estructura, pero **no una matriz o un array**.

La llamada a una función puede hacerse de muchas formas, dependiendo de qué clase de tarea realice la función. Si su papel fundamental es calcular un valor de retorno a partir de uno o más argumentos, lo más normal es que sea llamada incluyendo su nombre seguido de los argumentos actuales en una expresión aritmética o de otro tipo. En este caso, la llamada a la función hace el papel de un operando más de la expresión.

Obsérvese cómo se llama a la función seno en el ejemplo siguiente:

```
a = d * sin(alpha) / 2.0;
```

En otros casos, no existirá valor de retorno y la llamada a la función se hará incluyendo en el programa una sentencia que contenga solamente el nombre de la función, siempre seguido por los argumentos actuales entre paréntesis y terminando con un carácter (;). Por ejemplo, la siguiente sentencia llama a una función que multiplica dos matrices (nxn) A y B, y almacena el resultado en otra matriz C. Obsérvese que en este caso no hay valor de retorno (un poco más adelante se trata con detalle la forma de pasar vectores y matrices como argumentos de una función):

```
prod_mat(n, A, B, C);
```

Hay también casos intermedios entre los dos anteriores, como sucede por ejemplo con las funciones de entrada / salida. Dichas funciones tienen valor de retorno, relacionado de ordinario con el número de datos leídos o escritos sin errores, pero es muy frecuente que no se haga uso de dicho valor y que se llamen al modo de las funciones que no lo tienen.

La declaración y la llamada de la función *valor_abs()* antes definida, se podría realizar de la forma siguiente. Supóngase que se crea un fichero prueba.c con el siguiente contenido:

```
// fichero prueba.c
#include <stdio.h>
double valor_abs(double); // declaración
void main (void)
{
double z, y;
y = -30.8;
z = valor_abs(y) + y*y; // llamada en
una expresion
}
```

La función *valor_abs()* recibe un valor de tipo *double*. El valor de retorno de dicha función (el valor absoluto de y), es introducido en la expresión aritmética que calcula z.

La declaración (*double valor_abs(double)*) no es estrictamente necesaria cuando la definición de la función está en el mismo archivo *prueba.c* que *main()*, y dicha definición está antes de la llamada.

14.6.Relación entre funciones y variables.

Las variables que puede manejar una función pueden ser **variables locales y variables globales**.

Las **variables locales** a una función son las que se declaran en la propia función y **son desconocidas fuera de ella**, es decir, sólo existen en memoria durante la ejecución de la función. Se guardan en la pila (stack) de forma temporal y **no conservan su valor entre llamadas**, salvo si se declaran explícitamente como static, en cuyo caso son variables locales pero se guardan en memoria como las globales.

Las **variables globales** son aquellas que se declaran fuera de todas las funciones y se caracterizan por que se almacenan en una zona de memoria fijada por el compilador para los datos y la ocupan **mientras el programa está activo**. Pueden ser utilizadas por todas las funciones del programa. La declaración de las variables globales debe aparecer en el programa antes de ser utilizadas, aunque lo que se recomienda es **declararlas antes de la función *main()***.

14.7.Paso de argumentos a las funciones.

114.7.1.Paso por valor.

En la sección anterior se ha comentado que en la llamada a una función los argumentos actuales son evaluados y **se pasan copias de estos valores** a las variables que constituyen los argumentos formales de la función.

Aunque los argumentos actuales sean variables y no expresiones, y haya una correspondencia biunívoca entre ambos tipos de argumentos, los cambios que la función realiza en los argumentos formales **no se trasmiten** a las variables del programa que la ha llamado, precisamente porque lo que la función ha recibido son copias.

El modificar una copia no repercute en el original. A este mecanismo de paso de argumentos a

una función se le llama **paso por valor**. Considérese la siguiente función para permutar el valor de sus dos argumentos *x* e *y*:

```
void permutar(double x, double y) //
funcion incorrecta
{
double temp;
temp = x;
x = y;
y = temp;
}
```

La función anterior podría ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>
void main(void)
{
double a=1.0, b=2.0;
void permutar(double, double);
printf("a = %lf, b = %lf\n", a, b);
permutar(a, b);
printf("a = %lf, b = %lf\n", a, b);
}
```

Compilando y ejecutando este programa se ve que *a* y *b* siguen teniendo los mismos valores antes y después de la llamada a *permutar()*, a pesar de que en el interior de la función los valores sí se han permutado (es fácil de comprobar introduciendo en el código de la función los *printf()* correspondientes).

La razón está en que se han permutado los valores de las **copias de *a* y *b***, pero no los valores de las propias variables.

14.7.2.Paso por referencia.

Las variables podrían ser permutadas si se recibieran sus **direcciones** (en realidad, copias de dichas direcciones). Las direcciones deben recibirse en variables puntero, por lo que los **argumentos formales de la función deberán ser punteros**.

Una versión correcta de la función permutar() que pasa direcciones en vez de valores sería como sigue:

```
void permutar(double *x, double *y)
{
double temp;
temp = *x;
*x = *y;
*y = temp;
}
```

Que puede ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>
void main(void)
{
double a=1.0, b=2.0;
void permutar(double *, double *);
printf("a = %lf, b = %lf\n", a, b);
permutar(&a, &b);
printf("a = %lf, b = %lf\n", a, b);
}
```

Al mecanismo de paso de argumentos mediante direcciones en lugar de valores se le llama **paso por referencia**, y deberá utilizarse siempre que la función deba devolver argumentos modificados.

14.7.3.Función que recibe un array.

Un caso de particular interés es el paso de arrays (vectores, matrices y cadenas de caracteres). Este punto se tratará con más detalle un poco más adelante. Baste decir ahora que como los nombres de los arrays son punteros (es decir, direcciones), dichos datos se pasan **por referencia**, lo cual tiene la ventaja adicional de que no se gasta memoria y tiempo para pasar a las funciones copias de cantidades grandes de información.

14.7.4.Función que recibe una estructura.

Un caso distinto es el de las estructuras, y conviene tener cuidado. Por defecto las **estructuras se pasan por valor**, y pueden representar también grandes cantidades de datos (pueden contener arrays como miembros) de los que se realizan y transmiten copias, con la consiguiente pérdida de eficiencia. Por esta razón, las estructuras se suelen pasar de modo explícito por referencia, por medio de punteros a las mismas.

14.7.5.Función que recibe un puntero.

Cuando una función recibe un puntero, **está recibiendo una dirección por valor** (un puntero) y un dato por referencia (el dato apuntado por el puntero). La función puede modificar el dato apuntado, pero no podrá modificar su dirección.

14.8.Salida de una función.

Como ya se ha comentado la salida de una función se produce cuando se finalizan todas las operaciones que tiene programadas o bien cuando se ejecuta la sentencia ***return***. Esta última permite la salida de la función en cualquier punto y devolver un valor que debe corresponder con el tipo declarado en el prototipo de la función. La sintaxis de utilización de esta sentencia es la siguiente:

```
return expresión;
```

Cuando una función no devuelve ningún valor la salida puede hacerse mediante la sentencia *return* sin expresión de retorno.

La función ***exit()***, cuyo prototipo se encuentra en *stdlib.h*, fuerza la finalización del programa, independientemente del punto en que se encuentre su ejecución, incluso dentro de una función. Debe utilizarse con precaución, puesto que desestructura el programa haciendo muy difícil seguir el flujo de instrucciones. Se utiliza sobre todo para la **salida no traumática en caso de errores irrecuperables**.

14.8.1.Funciones que devuelven un puntero.

Las funciones solo devuelven un valor, sin embargo, si lo que devuelven es por ejemplo un puntero a un array de 100 elementos *float*, en realidad es como si estuviese devolviendo esos 100 valores.

La declaración de una función que devuelve un puntero tiene la siguiente forma:

```
Tipo_dato
*nombre_funcion(lista_parámetros);
```

Donde *tipo_dato* es el tipo de dato apuntado por el puntero.

Por ejemplo una función que devuelve un puntero a una cadena de caracteres y que recibe un entero y un array de caracteres, tendrá el prototipo siguiente:

```
char *Mifunción(int entero, char
array[]);
```

14.9.Función main().

Cuando se ejecuta un programa desde un terminal, tecleando su nombre, existe la posibilidad de **pasarle algunos datos**, tecleándolos a continuación en la misma línea. Por ejemplo, se le puede

pasar algún valor numérico o los nombres de algunos ficheros en los que tiene que leer o escribir información. Esto se consigue por medio de argumentos que se pasan a la función *main()*, como se hace con otras funciones.

Así pues, a la función *main()* se le pueden pasar argumentos y también puede tener valor de retorno.

El primero de los argumentos de main() se suele llamar ***argc***, y es una variable *int* que contiene el **número de palabras que se teclean a continuación del nombre del programa** cuando éste se ejecuta.

El segundo argumento se llama ***argv***, y es un array de punteros a cadenas que **contiene las direcciones de la primera letra o carácter de dichas palabras**. A continuación se presenta un ejemplo:

```
/*prueba.c – argumentos de main() */
int main(int argc, char *argv[])
{
int cont;
for (cont=0; cont<argc; cont++)
printf("El argumento %d es: %s\n", cont,
argv[cont]);
printf("\n");
return 0;
}
```

Si se teclea: ***prueba cadena 1 2 3 otromas*** y a continuación se pulsa la tecla intro, *argv[0]* apunta a la cadena "*prueba*", *argv[1]* apunta a "*cadena*", *argv[2]* apunta a "*1*", etc. Lógicamente, en al variable *argc* se recibe el valor 6. La salida del programa será:

```
El argumento 0 es prueba
El argumento 1 es cadena
El argumento 2 es 1
El argumento 3 es 2
El argumento 4 es 3
El argumento 5 es otromas
```

14.10.Funciones recursivas.

La recursividad es la **posibilidad de que una función se llame a sí misma**, bien directa o indirectamente. Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:

N! = N * (N-1)! = N * (N-1) (N-2)! = N * (N-1)*(N-2)*...*2*1

La función factorial, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long
numero)
{
if ( numero == 1 || numero == 0 )
return 1;
else
return numero*factorial(numero-1);
}
```

Supóngase la llamada a esta función para N=4, es decir factorial(4). Cuando se llame por primera vez a la función, la variable numero valdrá 4, y por tanto devolverá el valor de 4*factorial(3); pero factorial(3) devolverá 3*factorial(2); factorial(2) a su vez es 2*factorial(1) y dado que factorial(1) es igual a 1 (es importante considerar que sin éste u otro caso particular, llamado caso base, la función recursiva no terminaría nunca de llamarse a sí misma), el resultado final será 4*(3*(2*1)).

Por lo general la recursividad **no ahorra memoria**, pues ha de mantenerse una pila con los valores que están siendo procesados. **Tampoco es más rápida**, sino más bien todo lo contrario, pero **el código recursivo es más compacto** y a menudo más sencillo de escribir y comprender.

14.11.Punteros a funciones.

De modo similar a como el nombre de un *array* en C es un puntero, también **el nombre de**

una función es un puntero. Esto es interesante porque permite pasar como argumento a una función el nombre de otra función. Por ejemplo, si ***pfunc*** es un puntero a una función que devuelve un entero y tiene dos argumentos que son punteros, dicha función puede declararse del siguiente modo:

```
int (*pfunc)(void *, void *);
```

El primer paréntesis es necesario pues la declaración:

```
int *pfunc(void *, void *); //
incorrecto
```

Corresponde a una función llamada pfunc que devuelve un puntero a entero.

Considérese el siguiente ejemplo para llamar de un modo alternativo a las funciones *sin()* y *cos(x)*:

```
#include <stdio.h>
#include <math.h>
int main(void)
{
double (*pf)(double);
pf = sin;
printf("%lf\n", (*pf)(3.141592654/2));
pf = cos;
printf("%lf\n", (*pf)(3.141592654/2));
return 0;
}
```

Obsérvese cómo la función definida por medio del puntero tiene los mismos argumentos que las funciones seno y coseno. La ventaja está en que por medio del puntero *pf* las funciones seno y coseno podrían ser pasadas indistintamente como argumento a otra función.

14.12.Declaraciones complejas.

El operador puntero (*), los corchetes indicadores de array y los paréntesis dan lugar a declaraciones complejas y difíciles de descifrar. **Lo más aconsejable es evitarlo** y procurar, en al medida de lo posible, hacer declaraciones que no compliquen la escritura de los programas ni su lectura posterior.

En cualquier caso, para interpretar esas declaraciones complejas deben seguirse los siguientes pasos:

- Comenzar con el identificador teniendo en cuenta si a su derecha hay paréntesis (indicador de función) o corchetes (indicador de array) para considerarlo respectivamente como una función o un array.
- Mirar si hay un asterisco a la izquierda (indicador de puntero).
- Aplicar las reglas anteriores a cada nivel de paréntesis, de dentro hacia fuera.

A continuación se muestran algunos ejemplos de declaraciones complejas:

```
int (*lista)[20]
```

lista es un puntero a un array de 20 enteros.

```
char *datos[20]
```

datos es un array de 20 punteros a carácter.

```
void (*busc)()
```

busc es un puntero a una función que no devuelve nada.

```
char (*(*fun())[])()
```

fun es una función que devuelve un puntero a un array de punteros a funciones que devuelve cada una un carácter.

```
int (*(*tim[5])())[3]
```

tim es un array de 5 punteros a funciones que devuelven cada una un puntero a un array de 3 enteros.

14.13.Ejemplos.

```
/* cabecer1.c – usa una función para presentación*/
#include <stdio.h>
#define NOMBRE "ORDENATAS, S.A."
#define DIREC "Plaza del C Nº 12"
#define CIUDAD "Villafloat, E- 60006"
#define LIMITE 65

int main(void)
{
    void asteriscos(void);    /* prototipo de función */
    asteriscos();
    printf("%s\n", NOMBRE);
    printf("%s\n", DIREC);
    printf("%s\n", CIUDAD);
    asteriscos();             /*llamada a la función*/
    return 0;
}
/*****************************************/
    /* ahora viene la función asteriscos() */
/*****************************************/

void asteriscos(void)
{
    int cont;
    for (cont = 1; cont <= LIMITE; cont++)
    putchar('*');
    putchar('\n');
}
```

Más elegante con parámetros. Paso de argumentos por valor.

```
/* cabecer2.c */
#include <stdio.h>
#include <string.h>
#define NOMBRE "ORDENATAS, S.A."
#define DIREC "Plaza del C 12"
#define CIUDAD "Villafloat, E- 60006"
#define LIMITE 65
#define ESPACIO ' '

int main(void)
{
    int espacios;
    void n_car(char, int);    /*Declaración de la función*/
                                    /*recibe un char y un int*/
    n_car('*', LIMITE);       /* constante como argumento*/
    putchar('\n');

    n_car(ESPACIO, 25);       /* constante como argumento*/
    printf("%s\n", NOMBRE);
    espacios = (65 - strlen(DIREC)) / 2;/* dejamos al programa
                                                    calcular los espacios a saltar*/
    n_car(ESPACIO, espacios);
    printf("%s\n", DIREC);

    n_car(ESPACIO, (65 - strlen(CIUDAD)) / 2);    /* expresión como
                                                                argumento
```

```
*/
    printf("%s\n", CIUDAD);

    n_car('*', LIMITE);
    putchar('\n');
    return 0;
    }
/*********************************************/
/* aquí está n_car() */
/*********************************************/
void n_car(char car, int numero)
{
    int cont;
    for (cont = 1; cont <= numero; cont++ )
    putchar(car);
}
```

Funciones y variables.

```
/* buscadir.c -- averigua dónde se almacenan las variables */
#include <stdio.h>
void funcion(int);        /* declara función */

int main(void)
{
    int valora = 2, valorb = 5;
    printf("En main(), valora = %d y &valora = %p\n",valora, &valora);
    printf("En main(), valorb = %d y &valorb = %p\n",valorb, &valorb);
    funcion(valora);
    return 0;
}
/*************************************************/

void funcion(int valorb)    /* define función  */
{
    int valora = 10;
    printf("En mikado(), valora = %d y &valora = %p\n", valora, &valora);
    printf("En mikado(), valorb = %d y &valorb = %p\n", valorb, &valorb);
}
```

Salida del programa:

```
En main(), valora = 2 y &valora = 0xbff08340
En main(), valorb = 5 y &valorb = 0xbff0833c
En mikado(), valora = 10 y &valora = 0xbff08314
En mikado(), valorb = 2 y &valorb = 0xbff08320
```

Intento de modificación de variables.

```
/* cambio2.c –Tratamos de intercambiar dos valores pero no funciona */
#include <stdio.h>

void intercambia(int u, int v);
int main(void)
{
    int x = 5, y = 10;
```

```
    printf("En principio x = %d e y = %d.\n", x, y);
    intercambia(x,y);
    printf("Y despues x = %d e y = %d.\n", x, y);
    return 0;
}

void intercambia(int u, int v)
{
    int temp;
    printf("En principio u = %d y v = %d.\n", u, v);
    temp = u;
    u = v;
    v = temp;
    printf("Ahora u = %d y v = %d.\n", u, v);
}
```

Paso de argumentos por referencia. Este sí funciona.

```
/* cambio3.c -- intercambio realizado con punteros, este sí funciona */
#include <stdio.h>

void intercambia(int *u, int *v);

int main(void)
{
    int x = 5, y = 10;
    printf("En principio x = %d e y = %d.\n", x, y);
    intercambia(&x,&y);     /* envía las direcciones a la función */
    printf("Ahora x = %d e y = %d.\n", x, y);
    return 0;
}
void intercambia(int *u, int *v)
{
    int temp;
    temp = *u;/* temp toma el valor al que apunta u */
    *u = *v;
    *v = temp;
}
```

Array como argumento.

```
/*fuarray.c Paso de un array como argumento */
#include <stdio.h>
#include <stdlib.h>
/*Prototipos de las funciones*/
void Imprime_array(int *tabla,int max);
void PorDos(int *cosas,int tope);
int main(void)
{
    int tama,datos[]={1,3,5,7,9};
    system("clear");
    tama=sizeof(datos)/sizeof(int);/*Tamaño del array*/
    puts("Datos contiene:");
    Imprime_array(datos,tama);/*Función que imprime arrays*/
    PorDos(datos,tama); /*Función que multiplica por dos*/
    puts("Ahora datos contiene");
    Imprime_array(datos,tama);
```

```
    return 0;
}
/*********************************************/
/*        Definición de las funciones               */
/*********************************************/
void Imprime_array(int *tabla,int max)
{
    int i;
    for(i=0;i<max;i++) printf("%3d",tabla[i]);
    puts("");
}
/*********************************************/
void PorDos(int *cosas,int tope)
{
    int i;
    for(i=0;i<tope;i++)       /*Aritmética de punteros*/
    {
           *cosas*=2;
           cosas++;
    }
}
```

Paso de estructuras por valor.

```
/* fustruct.c -- pasa una estructura */
#include <stdio.h>
#include <stdlib.h>
struct fondos {
    char *banco;
    float ccorri;
    char *ahorro;
    float cahorro;
    }  garcia  = {      "Banco Pacífico",
                        10234.25,
                        "Banco de Poniente",
                        4239.21
 };

void Imprime(struct fondos);/* el argumento es una estructura */

int main(void)
{
    system ("clear");
    Imprime(garcia);
    return 0;
}
/*********************************************/
void Imprime(struct fondos pasta)
{
    printf("García tiene %.2f euros en el %s\n",pasta.ccorri,pasta.banco);
    printf("y además %.2f euros en el %s.\n",pasta.cahorro,pasta.ahorro);
    printf("Lo que hace un total de %.2f",pasta.ccorri+pasta.cahorro);
}
```

Paso de estructuras por referencia.

```
/* nombres1.c -- usa punteros a estructuras */
#include <stdio.h>
#include <string.h>
struct nombrect {
            char nombre[20];
            char apell[20];
            int letras;
                        };
void tomainfo(struct nombrect *);
void creainfo(struct nombrect *);
void sacainfo(struct nombrect *);

int main(void)
{
    struct nombrect persona;
    tomainfo(&persona);
    creainfo(&persona);
    sacainfo(&persona);
    return 0;
}

/*****************************************/

void tomainfo (struct nombrect *pst)
{
    printf("Introduzca su nombre.\n");
    gets(pst->nombre);
    printf("Introduzca su apellido.\n");
    gets(pst->apell);
}

void creainfo (struct nombrect *pst)
{
    pst->letras = strlen(pst->nombre) + strlen(pst->apell);
}

void sacainfo (struct nombrect *pst)
{
    printf("%s %s, su nombre contiene %d letras.\n",
    pst->nombre, pst->apell, pst->letras);
}
```

Devolución de un valor. Main() es un simple driver para comprobar el funcionamiento de la función imin().

```
/* elmenor.c -- decide el menor entre dos males */
#include <stdio.h>
int main(void)
{
    int mal1, mal2;
    int imin(int, int);

    while (puts ("Introduce dos números") && scanf("%d %d", &mal1, &mal2) == 2)
           printf("El menor entre %d y %d es %d.\n",
                        mal1, mal2,imin(mal1, mal2));
    return 0;
}
```

```
/*Esto es el cuerpo de la función*/
int imin(int n, int m)
{
    int min;
    if (n < m) min = n;
    else min = m;
    return min;
}
```

Argumentos de main().

```
/* eco.c -- main() con argumentos */
#include <stdio.h>

char *strupr ( char *);
int main(int argc, char *argv[])
{
    int cont;
    for (cont = 1; cont < argc; cont++)
    printf("%s ", strupr(argv[cont])); /* procesa los argumentos */
    printf("\n");
    return 0;
}

char *strupr ( char *palabra)
{
    char *letra;
    letra=palabra;
    while (*letra)
    {
           if (*letra>='a' && *letra<='z')
                  *letra-=32;
           letra++;
    }
    return palabra;
}
```

Si tecleáramos:

```
eco Me entero de lo que digo.
```

La salida sería:

```
ME ENTERO DE LO QUE DIGO.
```

```
/* mashola.c -- convierte argumento de línea de órdenes en número */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i, veces;

    if (argc < 2 || (veces = atoi(argv[1])) < 1)
           printf("Uso: %s número_positivo\n", argv[0]);
    else
           for (i = 0; i < veces; i++)
    puts("¡Hola, preciosa!");
    return 0;
}
```

Si tecleamos: hola 3. La salida sería:

```
¡Hola preciosa!
¡Hola preciosa!
¡Hola preciosa!
```

Recursividad. Recursión de cola.

```
/* factor.c -- cálculo de factoriales con recursividad */
#include <stdio.h>
int main(void)
{
    int num;
    long rfact(int n);
    while (puts("Introduce un número(una letra y sale):")&&scanf("%d", &num) == 1)
    {
           if (num < 0)
                  printf("Me horrorizan los números negativos.\n");
           else if (num > 15)
                  printf("Sólo puedo llegar hasta 15.\n");
                  else     printf("%d factorial = %ld\n", num, rfact(num));
    }
    return 0;
}
/*****************************************************************************/

long rfact(int n)     /* función recursiva */
{
    long ans;
    if (n > 0)
           ans = n * rfact(n-1);
    else
    ans = 1;
    return ans;
}
```

Tabla explicativa:

Variables	n	ans	n	ans	n	ans	n	ans
Llamada nivel 1	3	X	X	X	X	X	X	X
Llamada nivel 2	3	X	2	X	X	X	X	X
Llamada nivel 3	3	X	2	X	1	X	X	X
Llamada nivel 4	3	X	2	X	1	X	0	1
Retorno nivel 3	3	X	2	X	1	1	X	X
Retorno nivel 2	3	X	2	2	X	X	X	X
Retorno nivel 1	3	6	X	X	X	X	X	X
	L1		L2		L3		L4	

Retorno de la función: 6

Recursión e inversión.

```
/* binario.c -- imprime enteros en formato binario */
#include <stdio.h>
void a_binario(int n);
int main(void)
{
    int numero;
    while (puts("Introduce un número:")&&scanf("%d", &numero) == 1)
    {
           a_binario(numero);
           putchar('\n');
    }
    return 0;
}

void a_binario(int n) /* función recursiva */
{
    int r;
    r = n % 2;
    if (n >= 2)     a_binario(n / 2);
    putchar('0' + r);
    return;
}
```

Tabla explicativa:

Variables	N	r	imprime
Llamada nivel 1	10	0	
Llamada nivel 1	5	1	
Llamada nivel 1	2	0	
Llamada nivel 1	1	1	1
Retorno nivel 1	2	0	0
Retorno nivel 1	5	1	1
Retorno nivel 1	10	0	0
Imprime en total:			1 0 1 0

Imprimir una cadena hacia atrás.

```
/*rinver.c - imprime una cadena hacia atrás  de forma recursiva*/
#include <stdio.h>
void inversa(char *s);

int main(void)
{
    char cad[]="El misterio de las recursiones";
    inversa(cad);
    return 0;
}
```

```
void inversa(char *s)
{
    if (*s)
    inversa(s+1);
    putchar(*s);
}
```

Punteros a funciones.

```
/* punt_fun.c -- usa punteros a funciones */
#include <stdio.h>
#include <stdlib.h>
void palante(char *);
void patras(char *);
void saca(void (*fp)(char *ps), char *str);
/*Declaración de una función que recibe un puntero
    a una función, que recibe un puntero a char,
     y un puntero a char*/
int main(int argc, char *argv[])
{
    system("clear");
    if (argc < 2)
    {
           puts ("Uso: punt_fun palabra");
           exit(1);
    }
    saca(palante, argv[1]);/* usa palante()  */
    saca(patras, argv[1]);/* usa patras() */
    saca(palante,"Una");
    saca(patras,"arroz");
    saca(palante,"en");
    saca(patras,"amor");
    return 0;
}
/*************************************/

void palante(char *str)
{
    puts(str);          /*usa puts()*/
}

void patras(char *str)
{
    const char *princ = str;
    while (*str != '\0')
           str++;                /* va al final de la tira */
    while (str != princ)
           putchar(*--str);
    putchar('\n');
}

void saca(void (*fp)( char *ps), char *str)
              /* fp apunta a la función que se le pasa*/
{
    (*fp)(str);       /* pasa str a la función apuntada */
}
```

14.14.Ejercicios.

a)Diseña un programa para sumar dos números enteros utilizando una función para leer los datos y otra para realizar la suma.

b)Crea un programa que reciba como argumento dos números enteros y muestre los números primos comprendidos entre ambos.

c)Haz un programa que reciba como argumentos, en su llamada desde el sistema operativo, una letra y una cadena de caracteres y devuelva el número de veces que la letra aparece en la cadena.

d)Escribe un programa que permita elegir en un menú entre una serie de opciones utilizando funciones.

e)Elabora un programa que declare un puntero a una función y utiliza ese puntero para llamar a la función.

f)Realiza un programa que reciba como dato un entero y muestre su valor en binario.

g)Diseña un programa que lea una cadena y la muestre invirtiendo el orden de los caracteres.

15. Gestión dinámica de memoria.

15.1.Introducción.

Según lo visto hasta ahora, la reserva o asignación de memoria para vectores y matrices se hace de forma automática con la declaración de dichas variables, asignando suficiente memoria para resolver el problema de tamaño máximo, dejando el resto sin usar para problemas más pequeños. Así, si en una función encargada de realizar un producto de matrices, éstas se dimensionan para un tamaño máximo (100, 100), con dicha función se podrá calcular cualquier producto de un tamaño igual o inferior, pero aun en el caso de que el producto sea por ejemplo de tamaño (3, 3), la memoria reservada corresponderá al tamaño máximo (100, 100).

Este modo de asignación de memoria se denomina **asignación estática de memoria**.

Es muy útil el poder reservar más o menos memoria en **tiempo de ejecución**, es decir, tomando memoria de la que el sistema tiene libre en ese momento, según el tamaño del caso concreto que se vaya a resolver. A esto se llama **reserva o gestión dinámica de memoria**.

15.2.La memoria en los programas.

Cuando se ejecuta un programa, este se carga en la zona de memoria que el sistema operativo pone a su disposición. Esta memoria tiene distintos usos o aplicaciones y, para ello, se divide en distintas zonas:

- Zona de código, en una zona inalterable.
- Variables globales, en una zona llamada memoria de datos.
- Pila. Es en la pila donde se almacenan temporalmente variables y datos locales y las direcciones necesarias en las distintas llamadas a funciones.

Las variables **automáticas** son variables **locales a un bloque** de sentencias (subrutina, función o procedimiento). Se asignan automáticamente en la **pila de datos** cuando se entra en el bloque de código. Cuando se sale del bloque, las variables son automáticamente desasignadas. El programador no tiene control sobre la asignación de memoria para estas variables.

La **asignación dinámica de la memoria** es la que se realiza para el almacenamiento de variables en tiempo de ejecución de ese programa, sobre todo de *arrays* y estructuras. De esta forma al tamaño de memoria asignada puede **variar de tamaño** según las necesidades del programa. **En C es tarea del programador asignar y liberar este tipo de memoria**. Un objeto asignado dinámicamente permanece asignado hasta que es liberado explícitamente por el programador; esto es notablemente diferente de la asignación automática de memoria y de la asignación estática de memoria.

15.3.Asignación y liberación de memoria.

Existen en C dos funciones que reservan la cantidad de memoria deseada en tiempo de ejecución. Dichas funciones **devuelven (es decir, tienen como valor de retorno) un puntero a la primera posición de la zona de memoria reservada**.

Estas funciones se llaman ***malloc()*** y ***calloc()***, y sus declaraciones, que están en la librería ***stdlib.h***, son como sigue:

```
void *malloc(int n_bytes)
void *calloc(int n_datos,
                int tamaño_dato)
```

La función *malloc()* busca en la memoria el espacio requerido, lo reserva y devuelve un puntero genérico (void) al primer elemento de la zona reservada.

La función *calloc()* necesita dos argumentos:

El **nº de celdas** de memoria deseadas y el **tamaño** en bytes de cada celda.

Devuelve un puntero a la primera celda de memoria. La función calloc() tiene una propiedad adicional: **inicializa todos los bloques a cero**.

Estas dos funciones devuelven un puntero nulo (NULL) si no hay espacio libre.

Tener un puntero de tipo *void* apuntando a un bloque de memoria libre equivale a disponer de espacio en memoria para un array. El puntero genérico podrá convertirse al tipo de datos que interese en cada momento, de modo que se pueden almacenar **cualquier tipo válido** en C.

Por ejemplo para asignar memoria para un array de 5 elementos:

```
int *lista;
lista=(int*)malloc(5*sizeof(int));
if (lista==NULL)
    puts("Error en la asignación de
memoria");
```

Existe también una función llamada ***free()*** que deja libre la memoria reservada por *malloc()* o *calloc()* y que ya no se va a utilizar. Esta función usa como argumento el puntero devuelto por *calloc()* o *malloc()*.

La memoria no se libera por defecto, sino que **el programador tiene que liberarla explícitamente** con la función *free()*. El prototipo de esta función es el siguiente: void free(void *).

Para liberar la memoria asignada en el ejemplo anterior:

```
free(lista);
```

15.4.Arrays dinámicos.

Son aquellos cuyo tamaño no se establece mediante declaración expresa, sino que se determina en tiempo de ejecución, a través de asignación dinámica de memoria.

15.4.1.Arrays dinámicos unidimensionales.

Para crear un *array* dinámico unidimensional se requiere la declaración previa de un puntero al tipo de dato del *array*. El puntero declarado deberá apuntar al comienzo del bloque de memoria asignado mediante una función de asignación dinámica de memoria, normalmente *malloc()*.

El puntero permite acceder a todos los elementos del *array* **mediante indexación** (nombre del puntero, corchetes e índice del elemento) o **mediante la aritmética de punteros**, controlando siempre los límites de la memoria que puede utilizarse para no salir fuera de la zona asignada.

También se puede utilizar la función *calloc()*. Por ejemplo, las líneas siguientes reservan memoria para u *array* de 10 números de doble precisión y hace que la variable *punt* apunte el primer elemento del bloque:

```
double *punt;
punt=(double*)calloc(10,sizeof(double));
if(punt==NULL)
    puts("Error en la asignación de
memoria");
```

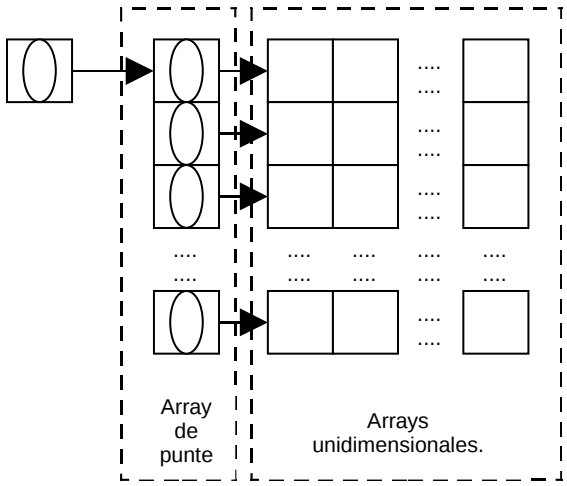
15.4.2.Arrays dinámicos bidimensionales.

La creación de un array dinámico bidimensional mediante un proceso de asignación dinámica

de memoria requiere los siguientes pasos:

- Asignar memoria a un **puntero a puntero**, con lo que podremos disponer de un **array de punteros**. El número de elementos del *array* de punteros determinará el **número de filas** del *array* bidimensional.
- Asignar memoria a cada uno de los punteros del *array* de punteros, lo que determinará el **número de columnas** del *array* aunque, cada fila podrá tener una longitud diferente.

Representación gráfica del proceso:



Por ejemplo, si se desea reservar memoria para un *array* bidimensional de números enteros de **NUMFIL** filas y **NUMCOL** columnas, se escribirá:

```
int **punt;       /* puntero a puntero a
                          entero*/
punt=(int **)
          malloc(NUMFIL*sizeof(int *));
if(punt==NULL)
    puts("Error en asignación de\\
          memoria");
for (i=0;i<NUMFIL;i++)
{
    punt[i]=(int *)
          malloc(NUMCOL*sizeof(int));
    if (punt==NULL)
          puts("Error en asignación\\
                de memoria");
}
```

De modo que cada puntero punt[i] apuntará al primer elemento de un array de NUMCOL elementos de tipo int. Los datos podrán referenciarse mediante la expresión **punt[i][j]**.

15.5.Reasignación de bloques de memoria.

Es posible **cambiar el tamaño** de un bloque de memoria previamente asignado de forma dinámica. Para este fin se dispone de la función ***realloc()***, cuyo prototipo se encuentra en ***stdlib.h*** y es:

```
Void * realloc(void *pbloc,unsigned
nbytes);
```

Donde ***pbloc*** es un puntero al comienzo del bloque de memoria actual, y ***nbytes*** indica el **nuevo tamaño en bytes** que deberá tener el bloque de memoria.

La función *realloc()* devuelve un puntero genérico al bloque de memoria asignado, **cuya dirección de comienzo puede ser diferente de la dirección de comienzo del bloque original**. No se pierden los datos que estuviesen almacenados en el bloque asignado inicialmente.

15.6.Ejemplos.

```
/*mallo1.c - reserva dinámica*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 81
```

```
int main(void)
{
    char *entra, *sale;     /*Punteros a char*/
    register int i;
                /*Reserva 81 caracteres de forma dinámica*/
    if (!(entra=(char*) malloc (sizeof(char)*MAX)))
    {
           puts("No hay memoria");/*error si no hay memoria libre*/
           exit (1);                        /*con este error no se puede continuar*/
    }
    sale=entra;         /*iguala los punteros*/
    puts ("¿ Qué quieres ? (máximo 80 caracteres).\n");
                        /*recoge los datos hasta el retorno de carro
                          y sin pasarse del maximo*/
    for(i=0;((*entra=getchar())!='\n') && i<MAX;i++,entra++);
    entra=sale;     /*actualiza entra al inicio de la cadena*/
    puts("\nPues entonces");             /*Imprime todo
                                                        menos las vocales*/
    for (i=0;(i<(sizeof(char)*MAX))&&(*sale!='\r');i++,sale++)
    {
           if (*sale=='a'||*sale=='e'||*sale=='i'||*sale=='o'||*sale=='u')
                 continue;
           putchar(*sale);
    }
    putchar('\n');
    free(entra);    /* Libera el bloque de memoria asignada*/
    return 0;
}
```

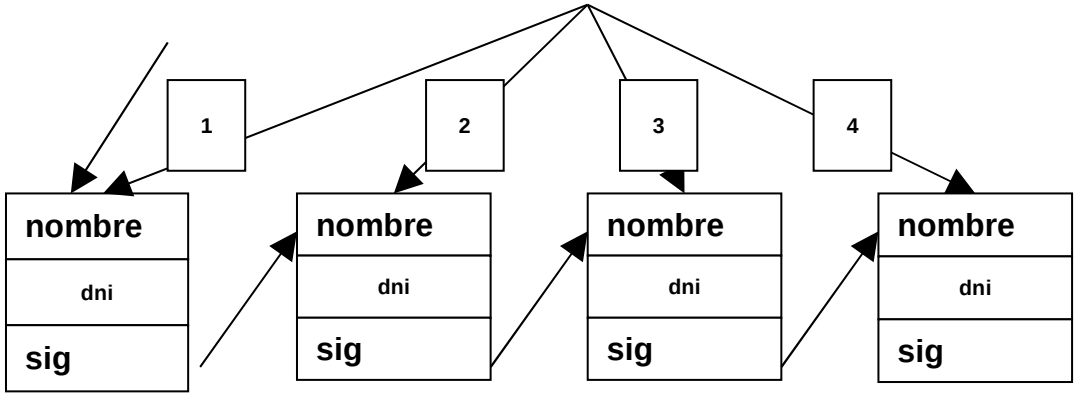
Estructuras y asignación dinámica.

```
/*malloc2.c --- asignación dinámica de estructuras */
#include <stdio.h>
#include <stdlib.h>
#define MAX 80

int main(void)
{
    struct datos{
           char nombre [MAX];  // struct datos *perso;
           long dni;
           } *perso;   /*Puntero a estructura*/

                 /*Reserva espacio para una estructura datos*/
    if(!(perso = (struct datos *) malloc (sizeof(struct datos))))
    {
           puts("\nNo cabe");  /*Si no cabe error*/
           exit (1);                 /*con este error no se puede continuar*/
    }
    puts("\nNombre: ");
    gets(perso->nombre);
    puts("\nD.N.I.: ");
    scanf("%ld", &perso->dni);
    printf("\n El nº de D.N.I. de %s es %ld", perso->nombre, perso->dni);
    free (perso);   /*Libera bloque de memoria asignado*/
    return 0;
}
```

Base de datos con reserva dinámica de memoria.

Estructura de la base de datos:

Punteros: prime perso




```
/*malloc3.c --Base de datos dinámica*/
#include <stdio.h>
#include <stdlib.h>
#define MAX 80

int main(void)
{
    struct datos{
           char nombre [MAX];
           long dni;
           struct datos *sig;  /*Puntero al dato siguiente*/
           } *perso, *prime;       /*Punteros a estruct datos*/

    if(!(perso = (struct datos *) malloc (sizeof(struct datos))))
    {
           puts("\nNo cabe");  /*Asigna espacio a una estruc datos*/
           exit (1);           /*error si no cabe*/
    }
    prime = perso;      /*Iguala punteros*/
    for(;;)                     /*Bucle infinito*/
    {
           puts("\nNombre (# para terminar):");
           gets(perso->nombre);
           if(perso->nombre[0] == '#')
           {
                  perso->sig=NULL;
                  break;
           }
           puts("\nD.N.I.: ");
           scanf("%ld", &perso->dni);
           while(getchar()!='\n');
           if(!(perso->sig = (struct datos *) malloc (sizeof(struct datos))))
           {
                  puts("\nNo cabe");  /*Asigna espacio a una estruc datos*/
                  exit (1);           /*error si no cabe*/
           }
           perso = perso->sig;     /*Hahora perso apunta al siguiente dato*/
    }
    perso = prime;              /*Perso apunta al primer registro*/
    while(perso->sig)          /*Mientras no sea NULL*/
    {
           printf("\n El nº de D.N.I. de %s es %ld", perso->nombre, perso->dni);
```

```
            perso=perso->sig;       /*Siguiente registro*/
    }
    perso = prime;
    while(perso->sig==NULL)
    {
            free (perso);       /*libera la memoria de todos los registros*/
            perso=perso->sig;   /*menos el último*/
    }
    free(perso);            /*libera memoria del último registro*/
    return 0;
}
```

Array dinámico bidimensional.

```
/*malloc4.c -- arrays y asignación dinámica*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 80
int main(void)
{
    float dia, cae,**datos;     /*datos es un puntero a punteros float*/
    int numdat=0,i,total=0,totdi=0;
    system ("clear");
    if(!(datos=malloc(sizeof(float *))))
    {
            puts("No hay memoria"); /*asigna memoria para un puntero*/
            exit (1);               /*a punteros float*/
    }
    puts("Introduce el Nº del día (mayor de 31 para salir):");
    scanf("%f",&dia);
    while (getchar()!='\n');
    while(dia<32)
    {
            puts("Introduce precipitación en litros");
            scanf("%f",&cae);
            if(!(datos=realloc(datos,(numdat+1)*sizeof(float *))))
            {
                    puts("No hay memoria"); /*aumenta la memoria */
                    exit (1);               /*reservada a punteros*/
            }
            if(!(datos[numdat]=malloc(sizeof(float)*2)))
            {
                    puts("No hay memoria"); /*reserva memoria para dos float*/
                    exit(1);
            }
            datos[numdat][0]=dia;
            datos[numdat][1]=cae;
            numdat++;
            puts("Introduce el Nº del día (mayor de 31 para salir):");
            scanf("%f",&dia);
            while (getchar()!='\n');
    }
    puts("Los datos introducidos son:");
    puts("\tMES\tLITROS");
    for(i=0;i<numdat;i++)
    {
            printf("\t%.0f\t%.0f\n", datos[i][0],datos[i][1]);
            total+=datos[i][1];
            totdi++;
```

```
    }
    printf("En %d dias han caido %d litros de agua.",totdi,total);
    for(i=0;i<numdat;i++)
          free(datos[i]);     /*libera la memoria utilizada*/
    free(datos);
    return 0;
}
```

15.7.Ejercicios.

a)Crear un programa para leer una cadena de caracteres desde el teclado y mostrarla en pantalla, reservando la memoria de manera dinámica.

b)Realiza un programa que reserve memoria para un array de caracteres utilizando la función calloc().

c)Diseña un programa que reserve espacia para un mensaje de 10 caracteres y luego amplíe el espacio para 20 caracteres.

d)Elabora un programa que permita capturar cadenas de longitud variable.

e)Haz un programa que asigne dinámicamente memoria para un array de mensajes.

f)Elabora un programa que permita crear una matriz de N*M elementos y los inicialice.

16. Archivos y secuencias en C.

16.1.Introducción.

En C se distingue entre archivos y secuencias. Un **archivo** es considerado como un **dispositivo físico** y real sobre el que se realizan las operaciones de entrada y salida. Una **secuencia** (stream) es un **ente abstracto** sobre el que realizan operaciones de entrada salida genérica; es decir, el conjunto de datos independientemente del dispositivo físico que los soporta.

El lenguaje C **contempla los archivos como secuencias continuas de bytes**, cada uno de los cuales puede ser accedido de forma individual.

Todas la funciones de entrada / salida sirven para todas las secuencias, solo hay que redirigirlas al archivo deseado en cada ocasión.

Algunos sistema operativos permiten abrir un archivo de dos modos distintos: modo texto y modo binario. Según el modo, los archivos serán considerados como secuencias de texto o secuencias binarias respectivamente.

Cuando un archivo se abre en modo texto el final de línea se representa en C como '\n' (CR-Retorno de carro), mientras que en Windows se requiere códigos '\r' y '\n' (CR-LF-Retorno de carro y avance de línea). Esto significa que en un archivo abierto en modo texto, si las funciones de lectura de C encuentran la secuencia CR-LF la convierten en el carácter CR y si encuentran el código correspondiente a ctrl-z lo interpretan como EOF (end of file- fin de fichero). En ciertas ocasiones estas conversiones pueden causar problemas.

Por ejemplo un fichero Windows que incluya estas líneas:

```
Serafina Caracola
Vive en su casa, ella sola.
```

Cuando se lea desde un programa en C abierto en modo binario se obtendrá:

```
Serafina Caracola\r\nVive en su casa,
ella sola.^z
```

Mientras que si se abre en modo texto se obtiene:

```
Serafina Caracola\nVive en su casa, ella
sola.^z
```

En linux no tenemos este problema puesto que un fin de línea se codifica como \n y los archivos se abren siempre en modo binario.

De todas formas, si queremos que nuestros programas puedan ser transportado a entornos no Unix, puede ser buena idea utilizar las funciones como si hubiera diferencia entre archivos de texto y binarios.

16.2.Apertura y cierre de un archivo.

La apertura y cierre de archivos desde C se realiza mediante funciones definidas por el estándar ANSI C y requiere un **puntero a un archivo**.

Un puntero a un archivo es un **puntero de tipo *FILE*** que apunta a toda la información necesaria para operar con el archivo (nombre, estado, posición actual, etc.) y se emplea para referirse al archivo en todas las operaciones de E/S. La declaración de este tipo de punteros es:

```
FILE *nombre_del_puntero;
```

Donde *FILE* es un tipo de estructura definido mediante *typedef* en el archivo *stdio.h*. Los miembros de la estructura *FILE* son transparentes al usuario, es decir, **son gestionados por el sistema operativo**.

16.2.1.Apertura de un fichero.

La función ***fopen()*** abre un archivo y devuelve un puntero de tipo *FILE* al archivo abierto o un puntero *NULL* en caso de error. Está declarada en ***stdio.h*** y su prototipo es:

```
FILE *fopen(char *nombre_archivo, char
*modo);
```

Donde *nombre_archivo* es una cadena constante (o el nombre de la misma) con el **nombre del archivo**, que puede incluir su ruta de acceso completa y modo representa otra cadena que determina el **modo de apertura del archivo**, de acuerdo con la siguiente tabla:

	Modo texto.	Modo binario.	
Abrir para lectura.	r	rb	Si no existe produce error.
Crear para escritura.	w	wb	Si existe se pierde el contenido.
Abrir/crear para añadir.	a	ab	Si no existe se crea.
Abrir para leer/escribir.	r+	rb+ o r+b	Debe existir.
Crear para leer/escribir.	w+	wb+ o w+b	Si existe se pierde el contenido.
Abrir/crear para añadir/leer	a+	ab+ o a+b	Si no existe se crea.

Todo esto es estrictamente por compatibilidad con ANSI X3.159-1989 (‘‘ANSI C’’) y no tiene efecto. **La ‘‘b’’ se ignora en todos los sistemas conformes con POSIX**, incluido Linux.

Otros sistemas pueden tratar los ficheros de texto y los ficheros binarios de forma diferente, y añadir la ‘‘b’’ puede ser una buena idea si realiza E/S de un fichero binario y espera que su programa pueda ser transportado a entornos no Unix.

De acuerdo con la tabla para crear un archivo en modo binario se escribirá:

```
FILE *pfich;      /*Puntero a fichero*/
If((pfich=fopen(“fichas”,”wb+”))==NULL)
{
    puts(“No se pudo crear el archivo”);
    exit(1);
}
    /*queda abierto el fichero fichas*/
```

Cuando se ha abierto un archivo, la estructura de tipo ***FILE*** a la que apunta el puntero asociado, contiene toda la información necesaria para su manejo; en esta estructura se **incluye un puntero de lectura / escritura que se desplaza por el archivo según las operaciones que se realicen**.

16.2.2.Cierre de un fichero.

Para que un archivo quede correctamente construido al finalizar las operaciones de E/S sobre él, debe ser cerrado. Esto se debe a que **las operaciones de E/S se realizan sobre un *buffer***, creado en **memoria** para este fin, que posteriormente es guardado en su lugar definitivo de almacenamiento.

El cierre de los archivos se produce automáticamente si el programa termina normalmente, no obstante, **es conveniente cerrar explícitamente** los archivos para liberar y reutilizar los punteros usados. Además si un archivo no se cierra queda inutilizado y **la información se pierde**.

La función ***fclose()*** cierra el archivo apuntado por el puntero que se le pasa como argumento. Su prototipo es:

```
int fclose(FILE *nombre_puntero);
```

La función devuelve un entero, que es cero si el cierre es correcto o EOF si se ha producido algún error.

Para cerrar el archivo abierto en el ejemplo anterior:

```
fclose(pfich);
```

16.2.3.Fin de fichero.

El carácter de fin de archivo está representado por la constante simbólica ***EOF*, definida en *stdio.h* como –1**, y es el byte que ocupa la última posición en el fichero.

El sistema operativo , utiliza para determinar el final de un fichero binario su tamaño, de esta forma puede almacenar todo tipo de caracteres, incluido *EOF*.

Algunas funciones pueden detectar *EOF* sin que se trate del final real del fichero. Para evitar problemas, sobre todo cuando se trata de secuencias binarias, se utiliza la función ***feof()***. Esta función, definida en ***stdio.h***, **devuelve un valor entero distinto de cero si se está leyendo el fin del fichero**. La sintaxis habitual de esta función es la siguiente:

```
while(¡feof(punt_fichero)
{
    /* Acciones sobre el archivo
abierto*/
}
```

16.3.Entrada/salida de texto.

16.3.1.E/S de caracteres.

Las dos funciones ***fgetc()*** y ***fputc()*** trabajan de una forma semejante a *getchar()* y *putchar()*. La diferencia es que se debe especificar el archivo que usan.

Así pues, la función para leer un carácter desde un fichero, previamente abierto, se utiliza de la siguiente manera:

```
cract=fgetc(punt_fichero);
```

Esta función devuelve el carácter leído o *EOF* en caso de fin de fichero o error.

Como ejemplo:

```
FILE *pfich;
char letra;
pf=fopen(“carta.txt”, “r”);
letra=fgetc(pfich);
```

Este ejemplo lee el primer carácter del fichero *carta.txt*.

La función **fputc()** escribe un carácter en el archivo apuntado por el puntero que recibe como argumento y devuelve el carácter escrito o *EOF* en caso de error.

El siguiente ejemplo muestra como se escribe un carácter en el archivo *carta.txt*:

```
FILE *pfich;
char letra=’a’;
pfich=fopen(“carta.txt”, “w”);
fputc(letra,pfich);
```

16.3.2.E/S de cadenas.

Para la lectura y escritura de cadenas se utilizan las funciones ***fgets()*** y ***fputs()*** declaradas en ***st-***

dio.h.

La función ***fgets()*** permite **leer una cadena** de un fichero abierto previamente.

```
char *fgets(char *s, int tam,
                  FILE *flujo);
```

Lee como mucho **uno menos de tam** caracteres del flujo y los guarda en el *búfer* al que apunte ***s***. La lectura se para tras un ***EOF* o una nueva-línea**. El carácter de nueva línea, también se guarda en el *búfer*. Tras el último carácter en el búfer se guarda un **'\0'**.Por ejemplo:

```
FILE *pfich;
char cadena[10];
pfich=fopen("carta.txt", "r");
fgets(cadena, 5, pfich);
```

En este caso se leen **4 caracteres** del fichero apuntado por ***pfich*** y se almacenan en la dirección apuntada por ***cadena***, **añadiendo el carácter nulo**.

Se observa que hay que pasarle a la función la **longitud de la cadena que queremos leer mas uno, para el carácter de fin de cadena**.

La función ***fputs()*** escribe una cadena de caracteres en un fichero **sin su terminador '\0'**,devuelve un **número no negativo** si todo funciona correctamente o *EOF* en caso de error.

A modo de ejemplo, la línea ***fputs("Hola\n",stdout)***, muestra en pantalla el mensaje "Hola" y retorno de carro. Para hacer lo mismo en un archivo:

```
FILE *pfich;
char cadena[]="Hola\n";
pfich=fopen("carta.txt","w");
fputs(cad,pfich);
```

16.4.Lectura y escritura de datos binarios.

La forma de almacenar datos numéricos es utilizar el mismo patrón que usa el programa. Así pues, **un dato de tamaño double debería almacenarse en un cajón de tamaño double**. Cuando el dato se almacena en un archivo que emplea la misma representación que el programa, se dice que estamos almacenando el dato en forma binaria, las funciones ***fread()*** y ***fwrite()*** son las encargadas de dar este servicio binario. Su prototipo es:

```
unsigned fread(void *ptr,
    unsigned tama,unsigned n,FILE *pf);
unsigned fwrite(void *ptr,
    unsigned tama,unsigned n,FILE *pf);
```

Donde ***ptr*** es un puntero a los datos que se van a leer o escribir. Para leer o escribir una variable, **la función debe recibir su dirección** (la variable puede ser de cualquier tipo de datos válido en C, incluidos los tipos definidos).

El valor ***tama*** representa el **tamaño en bytes de cada uno de los datos** o elementos que se van a escribir y que se obtiene aplicando el operador ***sizeof*** al tipo de dato que se está manejando.

El término ***n*** indica el número de datos o elementos que se van a leer o escribir.

Estas funciones **devuelven el número de elementos leídos o escritos**, que no tiene que coincidir necesariamente con el número de Bytes.

Ambas funciones tratan los datos con el mismo tamaño y formato con el que son almacenados en memoria, lo cual es importante cuando se trabaja con **estructuras y uniones**.

Suponiendo que se ha abierto un fichero cuyo puntero es ***pfich***, para guardar un dato de tipo ***float*** almacenado en la variable ***dato***, habrá que escribir la sentencia:

```
fwrite(&dato,sizeof(dato),1,pfich);
```

Si la operación fuese de lectura, la línea sería:

```
fread((&dato,sizeof(dato),1,pfich);
```

Si la variable ***dato*** fuese una estructura con múltiples miembros, no habría ninguna diferencia en el uso de las funciones anteriores.

16.5.Entrada/salida con formato.

Todas las funciones de entrada / salida estándar que hemos utilizado anteriormente tienen su equivalente en entrada / salida de archivos. La diferencia fundamental es que necesitamos un puntero a *FILE* para indicar a las nuevas funciones el archivo en el que tienen que trabajar.

Las funciones equivalentes de *printf()* y *scanf()* para lectura escritura de archivos son ***fscanf()*** y ***fprintf()***, definidas en ***stdio.h*** y sus prototipos son los siguientes:

```
Int fprintf(FILE *pf,
    ”cadena de control”,
    lista_de_argumentos);

Int fscanf(FILE *pf,
    ”cadena de control”,
    lista_de_argumentos);
```

En los dos casos **pf** representa el puntero al fichero sobre el que queremos operar, y los demás argumentos son los mismos que para *printf()* y *scanf()*.

La función *fprintf()* devuelve el número de bytes escritos, sin incluir el '\0', si ha funcionado correctamente, o *EOF* en caso de error.

La función *fscanf()* devuelve el número de campos correctamente procesados o *EOF* en caso de error o de que se haya alcanzado el final del fichero.

A continuación se muestra un ejemplo para almacenar tres tipos de datos en un fichero:

```
FILE *pf;
char car='H';
int ente=23;
float real=3.1416;
pf=fopen(“datos.dat”,”W+”);
fprintf(pf,”%c %d %f”,car,ente,real);
fclose(pf);
```

Las líneas para leer los datos del ejemplo anterior serán:

```
FILE *pf;
char car;
int ente;
float real;
pf=fopen(“datos.dat”,”r+”);
fscanf(pf,”%c %d %f”,&car,&ente,&real);
fclose(pf);
```

16.6.Entrada/salida de acceso aleatorio.

16.6.1.Función fseek():

En las funciones anteriores, cada vez que se lee o escribe un dato, **el puntero de lectura/escritura de la estructura *FILE* se incrementa automáticamente**, apuntando al siguiente dato sobre el que opera. Este sistema se denomina modo de **acceso secuencial**.

Si queremos acceder a los datos en cualquier posición del archivo se utiliza el denominado modo de **acceso aleatorio**, que nos permite tratar el fichero como si fuese un ***array***, moviéndose directamente a un byte determinado del archivo abierto con *fopen()*.

Para realizar estas operaciones se deben incluir los mecanismos necesarios para controlar los movimientos a lo largo del archivo para no exceder sus límites en ningún caso.

La función ***fseek()*** está diseñada para facilitar el acceso directo a los archivos, ya que permite

cambiar la posición direccionada por el puntero de lectura / escritura. Está declarada en ***stdio.h*** y su prototipo es el siguiente:

```
int fseek(FILE *pf,long nbytes,
    int origen);
```

Donde ***pf*** es el puntero al fichero sobre el que se opera, ***nbytes*** representa el número de bytes que queremos desplazarnos con relación al punto de partida, y ***origen*** es el número entero que representa el punto de partida a la hora de considerar el desplazamiento.

Como valor de *origen* se puede incluir cualquiera de las siguientes constantes simbólicas (definidas en *stdio.h*):

- **SEEK_SET**. Que corresponde al principio del archivo.
- **SEEK_CUR**. Para la posición actual.
- **SEEK_END**. Que indica el final del fichero.

La función *fseek()* devuelve un valor nulo si el movimiento del puntero se realizó con éxito y un valor distinto de cero en caso contrario.

Ejemplo de la utilización de la función *fseek()*:

```
int datos[20,10];
int mat[20][10];
FILE *pf;
pf=fopen("datos.dat","a+");
……
fwrite(datos,sizeof(int),20*10,pf);
fseek(pf,sizeof(int)10*n,SEEK_SET);
fread(mat[n],sizeof(int),10,pf);
fclose(pf);
```

La llamada a *fwrite()* sirve para guardar la matriz ***datos*** en el archivo ***datos.dat***. La llamada a *fseek()* mueve el puntero de lectura / escritura del archivo al principio de la **fila *n***, ya que *10*sizeof(int)* sirve para calcular el **tamaño en bytes de la fila de la matriz** del ejemplo, de esta forma la llamada a *fread()* que hay a continuación lee todos los datos de la **fila *n***.

16.6.2.Función ftell().

Esta función devuelve en un entero de tipo *long* la **posición relativa del puntero de lectura/escritura de un fichero** (en bytes contados a partir del principio del fichero). Está declarada en *stdio.h* y su prototipo es:

```
long ftell(FILE *pf);
```

16.6.3.Función rewind().

La función ***rewind()*** inicializa el indicador de posición del archivo, **haciendo que apunte al principio**. Está definida en *stdio.h*, no devuelve ningún valor y su prototipo es el siguiente:

```
void rewind(FILE *pf);
```

16.7.Otras operaciones con archivos.

16.7.1.Eliminación de archivos.

La función ***remove()*** borra el archivo cuyo nombre se especifica en la cadena que recibe como argumento. Está declarada en *stdio.h* y su prototipo es:

```
Int remove(char *nombre_archivo);
```

Recibe como argumento una cadena con el

nombre del archivo o un puntero a esa cadena.

Devuelve cero si la operación tiene éxito y –1 si se produce algún error, en cuyo caso la variable global ***errno*****,** definida en ***errno.h*** indicará el tipo de error.

16.7.2.Volcado de una secuencia.

La función ***fflush()*** vacía el contenido de una secuencia, borrando toda la información almacenada en el *buffer* de memoria asociado. Está declarada en ***stdio.h*** y su prototipo es el siguiente:

```
Int fflush(FILE *pf);
```

Devuelve cero si la operación tiene éxito y *EOF* en caso de error. El puntero que recibe como argumento representa un puntero a cadena de caracteres que contiene el nombre de la secuencia.

16.8.Ejemplos.

Abrir un fichero y contar el número de caracteres que tiene.

```
/* cuentas.c -- uso de E/S estándar */
#include <stdio.h>
#include <stdlib.h>    /* prototipo ANSI C de exit() */
int main(int argc, char *argv[])
{
    int ch;          /* aquí se almacenan los caracteres que se leen */
    FILE *fp;        /* "puntero a fichero" */
    long cont = 0;

    if (argc != 2)
    {
           printf("Uso: %s nombre de fichero\n", argv[0]);
           exit(1);
    }
    if ((fp = fopen(argv[1], "rb")) == NULL)
    {
           printf("No puedo abrir %s\n", argv[1]);
           exit(1);
    }
    while ((ch =getc(fp)) != EOF)
    {
           putc(ch, stdout);
           cont++;
    }
    fclose(fp);
    printf("El fichero %s tiene %ld caracteres\n", argv[1], cont);
    return 0;
}
```

Lectura y escritura de ficheros.

```
/* cripto.c -- encripta chapuceramente un fichero de texto */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
```

```
{
    FILE *in, *out;           /* declara dos punteros FILE   */
    int ch;
    char nombre[40];          /* para guardar el nombre delfichero salida */
    char exten[4];            /* para guardar la extensión del fichero */
    char *nom;

    if (argc < 3)    /* comprueba que hay fichero de entrada */
    {
          fprintf(stderr, "Uso: %s nombre del fichero letra\n", argv[0]);
          exit(1);
    }
    if ((in = fopen(argv[1], "rb")) == NULL) /*abre el fichero para lectura*/
    {
          fprintf(stderr, "No puedo abrir el fichero \"%s\"\n", argv[1]);
          exit(2);
    }
    strcpy(nombre, argv[1]);  /* copia nombre del fichero en un array */
    nom=nombre;
    for(;*nom!='.';nom++);
          *nom='\0';
    nom++;
    if(!strcmp(nom,"txt")) strcpy(exten,".cod");
    else if(!strcmp(nom,"cod")) strcpy(exten,".txt");
          else
          {
                fprintf(stderr,"No has usado la extensión correcta.");
                exit(3);
          }
    strcat(nombre,exten);               /* añade la extensión al nombre */
    if ((out = fopen(nombre, "wb")) == NULL)   /* abre otro fichero para escritura*/
    {
          fprintf(stderr, "No puedo crear fichero de salida.\n");
          exit(3);
    }
    while ((ch = fgetc(in)) != EOF)
          fputc(ch^argv[2][0], out);     /* envía un carácter codificado*/
    if (fclose(in) != 0 || fclose(out) != 0)
          fprintf(stderr, "Error al cerrar ficheros\n");
    return 0;
}
```

Ejemplo de uso de fprintf(), fscanf() y rewind().

```
/* mete_pal.c -- usa fprintf(), fscanf() y rewind() */
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
int main(void)
{
    FILE *fi;
    char palabras[MAX];

    if ((fi = fopen("palabras", "a+")) == NULL)
    {
          fprintf(stderr, "No puedo abrir el fichero \"palabras\".\n");
          exit(1);
    }
    puts("Introduzca texto a añadir al fichero; pulse la tecla");
    puts("Intro al comienzo de una línea para terminar.");
```

```
    while (gets(palabras) != NULL && palabras[0] != '\0')
          fprintf(fi, "%s ", palabras);
    puts("Contenido del fichero:");
    rewind(fi);                  /* retrocede al comienzo del fichero */
    while (fscanf(fi, "%s", palabras) != EOF)     /*recordemos que scanf para
                                                  cuando encuentra un espacio
                                                   en blanco*/
          puts(palabras);
    if ((fclose(fi)) != 0)
          fprintf(stderr, "Error al cerrar fichero\n");
    return 0;
}
```

Uso de fgets() y fputs().

```
/* eco.c -- usa fgets() y fputs() */
#include <stdio.h>
#define LINEAMAX 20

char *struper (char *);

int main(void)
{
    char linea[LINEAMAX];
    while (fgets(linea, LINEAMAX, stdin) != NULL && linea[0] != '\n')
          fputs( struper (linea), stdout);
    return 0;
}

char *struper (char *palabra)
{
    char *letra;
    letra=palabra;
    while (*letra)
    {
          if (*letra>='a' && *letra<='z')
                *letra-=32;
          letra++;
    }
    return palabra;
}
```

La salida sería:

```
yo@panoramix:~$ ./ecoeco
Viento en popa
VIENTO EN POPA
a toda vela
A TODA VELA
```

Utilización de fseek() y ftell().

```
/* al_reves.c -- presenta un fichero en orden inverso */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

```
    char ch;
    FILE *fp;
    long cont, ultimo;

    if (argc != 2)
    {
          printf("Uso: al_reves fichero\n");
          exit(1);
    }
    if ((fp = fopen(argv[1], "rb")) == NULL)
    {                                  /* modo sólo lectura binario */
          printf("al_reves no puede abrir %s\n", argv[1]);
          exit(1);
    }
    fseek(fp, 0L, SEEK_END);  /* va al final del fichero */
    ultimo = ftell (fp);
    for (cont = 1L; cont <= ultimo; cont++)
    {
          fseek(fp, -cont, SEEK_END);  /* retrocede en el fichero */
          ch = getc(fp);
          putchar(ch);
    }
    fclose(fp);
    return 0;
}
```

Ficheros y estructuras.

```
/* guardalb.c -- guarda estructura en un fichero */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXTIT 40
#define MAXAUT 40
#define MAXLIB 100   /* número máximo de libros      */
#define STOP ""      /* tira nula, finaliza entrada  */
struct biblio {      /* prepara patrón de estructura */
          char titulo[MAXTIT];
          char autor[MAXAUT];
          double precio;
          };
int main(void)
{
    struct biblio libro[MAXLIB];  /* array de estructuras biblio */
    int cont = 0;
    int indice, fichcont;
    FILE *plibros;
    int ocupa = sizeof (struct biblio);
    if ((plibros = fopen("libro.dat", "a+b")) == NULL)
    {
          fputs("No puedo abrir el fichero libro.dat\n", stderr);
          exit(1);
    }
    rewind(plibros);                     /* va a comienzo del fichero */
    while (cont < MAXLIB && fread(&libro[cont], ocupa, 1, plibros) == 1)
    {
          if (cont == 0)
                puts("Contenido actual de libro.dat:");
          printf("%s por %s: %.2f euros.\n", libro[cont].titulo,libro[cont].autor,
                                                          libro[cont].precio);
```

```
            cont++;
      }
      fichcont = cont;
      if (cont == MAXLIB)
      {
            fputs("El fichero libro.dat está lleno.", stderr);
            exit(2);
      }
      puts("Añada nuevos títulos de libros.");
      puts("Pulse [Intro] a comienzo de línea para parar.");
      while (cont < MAXLIB &&strcmp(gets(libro[cont].titulo),STOP) != 0)
      {
            puts("Introduzca ahora el autor.");
            gets(libro[cont].autor);
            puts("Ahora ponga el precio.");
            scanf("%lf", &libro[cont++].precio);
            while (getchar() != '\n');    /* limpia línea entrada */
            if (cont < MAXLIB)
                  puts ("Introduzca el siguiente título.");
      }
      puts("Ahí va su lista de libros:");
      for (indice = 0; indice < cont; indice++)
            printf("%s por %s: %.2f euros.\n", libro[indice].titulo,
                                                      libro[indice].autor,
  libro[indice].precio);
      fseek(plibros, 0L, SEEK_END);  /* va al final del fichero */
      fwrite(&libro[fichcont], ocupa, cont - fichcont, plibros);
      fclose(plibros);
      return 0;
  }
```

Ejercicios.

a) Haz un programa que copie un archivo de texto en otro.

b) Diseña un programa que almacene en un archivo las letras introducidas por el teclado, finalizando con el carácter '#'.

c) Crea un programa que lea los datos de una estructura del teclado y los almacene en un fichero, como único registro, utilizando fread() y fwrite() para comprobar el funcionamiento correcto.

d) Elabora un programa que tome datos numéricos de un fichero, los sume y guarde los resultados en otro archivo, haciendo uso de las funciones fscanf() y fprintf().