



# **Introducción a la programación en C**

Diego Rodríguez-Losada González

Javier Muñoz Cano

Cecilia García Cena



Introducción a la programación en C

© 2008. Todos los derechos reservados. Prohibida la reproducción.

Diego Rodríguez-Losada González, Javier Muñoz Cano, Cecilia García Cena  
Universidad Politécnica de Madrid

ISBN:

Dirección, coordinación y montaje: Diego Rodríguez-Losada

Revisiones: Diego Rodríguez-Losada y Javier Muñoz

Impresión: EUITI-UPM. Ronda de Valencia, 3. 28012. Madrid, España



# Contenidos

<b>CONTENIDOS .....</b>	<b>1</b>
<b>PRÓLOGO.....</b>	<b>5</b>
<b>1. INTRODUCCIÓN A LA PROGRAMACIÓN.....</b>	<b>7</b>
1.1 LENGUAJES DE PROGRAMACIÓN.....	7
1.2 EL LENGUAJE DE PROGRAMACIÓN C .....	8
1.3 PROCESO DE CREACIÓN DE UN PROGRAMA EJECUTABLE .....	9
1.4 TIPOS DE ERRORES.....	10
1.5 INGENIERÍA DEL SOFTWARE .....	11
<b>2. INTRODUCCIÓN AL LENGUAJE C.....</b>	<b>13</b>
2.1 INTRODUCCIÓN .....	13
2.2 PRIMER PROGRAMA EN C: HOLA MUNDO.....	13
2.3 ELEMENTOS DEL LENGUAJE C .....	15
2.3.1 Comentarios .....	15
2.3.2 Separadores .....	15
2.3.3 Palabras clave.....	16
2.3.4 Identificadores.....	16
2.3.5 Constantes.....	18
2.3.6 Operadores.....	18
2.4 VARIABLES Y CONSTANTES.....	19
2.4.1 Tipos de variables, declaración e inicialización.....	19
2.4.2 Modo de almacenamiento y ámbito de una variable.....	22
2.5 OPERADORES .....	23
2.5.1 Operadores aritméticos .....	23
2.5.2 Operadores relacionales.....	24
2.5.3 Operadores incrementales .....	24
2.5.4 Operadores lógicos .....	25
2.5.5 Operadores de bits .....	25
2.5.6 Operadores de asignación .....	26
2.5.7 Conversiones implícitas de tipo .....	27
2.5.8 Otros operadores.....	27
2.5.9 Precedencia y asociatividad .....	28
2.5.10 Funciones matemáticas .....	30
2.6 ENTRADA Y SALIDA POR CONSOLA .....	30
2.7 ESTILO DEL CÓDIGO.....	32
2.8 EJERCICIOS RESUELTOS .....	33
2.9 EJERCICIOS PROPUESTOS .....	35
<b>3. SENTENCIAS DE CONTROL .....</b>	<b>37</b>
3.1 INTRODUCCIÓN .....	37
3.2 CONDICIONALES.....	37
3.2.1 Instrucción if-else.....	37
3.2.2 Instrucción switch-case .....	43
3.3 BUCLES .....	44
3.3.1 Generalidades .....	44
3.3.2 Bucle for.....	44
3.3.3 Bucle while.....	47
3.3.4 Bucle do - while.....	48
3.3.5 ¿Que bucle elegir? .....	49
3.3.6 Bucles anidados.....	50

3.3.7	Algorítmica vs. Matemáticas.....	51
3.3.8	Un error típico con los bucles.....	52
3.4	ALTERACIONES DEL FLUJO DE EJECUCIÓN .....	52
3.4.1	Sentencia <i>break</i> .....	53
3.4.2	Sentencia <i>continue</i> .....	53
3.4.3	Instrucción <i>goto</i> .....	55
3.5	EJERCICIOS RESUELTOS .....	56
3.6	EJERCICIOS PROPUESTOS .....	60
<b>4.</b>	<b>TIPOS AVANZADOS DE DATOS .....</b>	<b>61</b>
4.1	VECTORES.....	61
4.1.1	Declaración de vectores .....	61
4.1.2	Inicialización de un vector.....	62
4.1.3	Acceso a datos de un vector.....	63
4.1.4	Operaciones con vectores .....	66
4.1.5	Vectores de dimensión variable .....	68
4.1.6	Ordenamiento de un vector.....	70
4.2	MATRICES.....	70
4.2.1	Declaración de matrices .....	70
4.2.2	Inicialización de matrices .....	71
4.2.3	Acceso a datos de una matriz.....	71
4.3	CADENAS DE CARACTERES .....	73
4.3.1	Declaración de cadenas.....	73
4.3.2	Inicialización de cadenas, entrada y salida por consola .....	73
4.3.3	Operaciones con cadenas .....	74
4.3.4	Librería <i>string.h</i> .....	76
4.4	ESTRUCTURAS.....	78
4.4.1	Creación de una estructura.....	78
4.4.2	Declaración de una variable de tipo estructura .....	78
4.4.3	Inicialización de una estructura.....	79
4.4.4	Acceso a miembros de una estructura .....	79
4.4.5	Copia de estructuras.....	80
4.4.6	Vector de estructuras.....	80
4.4.7	Estructuras que contienen otras estructuras y/o vectores .....	83
4.5	UNIONES .....	84
4.6	ENUMERACIONES .....	85
4.7	TIPOS DEFINIDOS POR EL USUARIO .....	86
4.8	EJERCICIOS RESUELTOS .....	86
<b>5.</b>	<b>PUNTEROS.....</b>	<b>91</b>
5.1	ORGANIZACIÓN DE LA MEMORIA.....	91
5.2	¿QUÉ ES UN PUNTERO? .....	93
5.3	DECLARACIÓN DE UN PUNTERO.....	93
5.4	OPERADORES .....	94
5.5	INICIACIÓN DE UN PUNTERO.....	95
5.6	OPERACIONES CON PUNTEROS .....	96
5.6.1	Asignación.....	96
5.6.2	Aritmética y comparación.....	96
5.7	PUNTEROS A VECTORES .....	98
5.8	PUNTEROS A CADENAS DE CARACTERES .....	100
5.9	PUNTEROS A ESTRUCTURAS .....	101
5.10	PUNTEROS A PUNTEROS.....	102
5.11	ASIGNACIÓN DINÁMICA DE MEMORIA.....	103
<b>6.</b>	<b>ESTRUCTURA DE UN PROGRAMA .....</b>	<b>107</b>
6.1	DIRECTIVAS DEL PREPROCESADOR .....	107
6.1.1	<i>#include</i> .....	107
6.1.2	<i>#define</i> y <i>#undef</i> .....	108
6.1.3	Macros .....	108
6.2	FUNCIONES. ESTRUCTURA DE UN PROGRAMA.....	109

6.2.1	Flujo de ejecución .....	110
6.2.2	Declaración de una función: prototipo .....	110
6.2.3	Definición de una función.....	112
6.2.4	Llamada a una función .....	113
6.2.5	¿Donde se declara el prototipo de la función?.....	113
6.3	ÁMBITO DE UNA VARIABLE.....	114
6.3.1	Variables Globales .....	114
6.3.2	Variables Locales .....	114
6.4	PARÁMETROS DE UNA FUNCIÓN .....	115
6.4.1	Paso por valor.....	117
6.4.2	Paso por referencia .....	117
6.5	RETORNO DE UNA FUNCIÓN .....	120
6.5.1	Funciones que no devuelven resultados (tipo void) .....	120
6.5.2	Funciones que devuelven resultados .....	121
6.5.3	Funciones con retorno booleano.....	124
6.6	FUNCIONES QUE LLAMAN A OTRAS FUNCIONES.....	125
6.7	FUNCIONES Y VECTORES .....	127
6.8	FUNCIONES Y MATRICES .....	130
6.9	FUNCIONES Y CADENAS .....	131
6.10	FUNCIONES Y ESTRUCTURAS .....	133
6.10.1	Estructuras como parámetros de una función.....	133
6.10.2	Estructura como retorno de una función.....	136
6.11	ESTRUCTURACIÓN DE FUNCIONES EN FICHEROS .....	137
6.12	RECURSIVIDAD .....	138
6.13	PARÁMETROS DE LA FUNCIÓN MAIN().....	140
6.14	EJERCICIOS RESUELTOS .....	141
<b>7.</b>	<b>ENTRADA Y SALIDA.....</b>	<b>147</b>
7.1	INTRODUCCIÓN .....	147
7.2	ENTRADA Y SALIDA ESTÁNDAR.....	147
7.2.1	Salida estándar.....	148
7.2.2	Entrada estándar.....	155
7.3	GESTIÓN DE FICHEROS .....	160
7.3.1	Abrir un fichero: la función fopen().....	161
7.3.2	Cerrar un fichero: la función fclose() .....	162
7.3.3	Gestión de errores .....	163
7.3.4	Operaciones de lectura/escritura.....	165
7.4	EJERCICIOS RESUELTOS .....	174
<b>8.</b>	<b>PROBLEMAS RESUELTOS .....</b>	<b>179</b>
8.1	MÁXIMO COMÚN DIVISOR (MCD).....	179
8.2	ADIVINAR UN NÚMERO .....	180
8.3	LOTERÍA.....	181
8.4	INTEGRAL.....	183
8.5	NÚMEROS PRIMOS .....	183
8.6	FIBONACCI .....	184
8.7	MEDIA Y RANGO DE UN VECTOR.....	185
<b>9.</b>	<b>APLICACIÓN PRÁCTICA: AGENDA.....</b>	<b>187</b>
9.1	CARACTERÍSTICAS DE LA AGENDA .....	187
9.2	PROTOTIPOS DE FUNCIONES Y FUNCIÓN MAIN() .....	188
9.3	FUNCIONES DE GESTIÓN DE UN CONTACTO .....	189
9.4	FUNCIONES QUE MANEJAN 2 CONTACTOS .....	190
9.5	FUNCIONES DE GESTIÓN DE LA AGENDA .....	191
9.6	FUNCIONES AUXILIARES DE GESTIÓN DE E/S .....	193
9.7	RESULTADO .....	195
<b>BIBLIOGRAFÍA .....</b>		<b>197</b>





# Prólogo

A la hora de lanzarse a escribir un libro, especialmente de un tema del que existe una numerosa y excelente bibliografía, como es la programación en C, merece la pena preguntarse si es necesario y si aporta algo a lo ya existente. Creemos que este libro presenta un nuevo enfoque al aprendizaje inicial de la programación en C, gracias a la experiencia de los autores como programadores y docentes. Este libro se inspira a partes iguales en los apuntes “Aprenda C como si estuviera en primero”, bibliografía reconocida como los libros de “Deitel & Deitel”, y en nuestro día a día en la enseñanza de la materia. El resultado es un material que puede ser usado tanto para el autoaprendizaje, como de soporte y apoyo para clase teórica y práctica.

Enseñar y aprender a programar (en cualquier lenguaje, incluido el C) no es una tarea sencilla. En este libro hemos seguido una metodología incremental, en la que siempre se comienzan por los ejemplos más básicos posibles y se va avanzando gradualmente en la materia a través de explicaciones teóricas breves acompañadas de programas, aplicaciones y ejemplos. Nos centramos en lo esencial de la programación como es la algorítmica, el manejo de datos, y especialmente en la programación estructurada o uso de funciones.

Es un libro didáctico, no pretende ser un libro de referencia. Así, el programador experto encontrara algunos vacíos u omisiones que realmente no son tales, sino que han sido obviadas intencionadamente para librar al lector no experto de “distracciones” que le aparten de las ideas fundamentales. No obstante se ha intentado ser equilibrado entre lo que es la programación real y la simplicidad necesaria para el novel. Por ejemplo se evita el uso de variables globales que es un recurso frecuente de los programadores inexpertos, ya que la Ingeniería del Software recomienda evitarlas, y así se hace para evitar aprendizajes “viciados”.

Conviene recordar al lector, que no es posible aprender un lenguaje de programación mediante la lectura de ningún libro, manual o tutorial. Es necesario practicar delante del computador para adquirir el conocimiento. Por tanto se recomienda la lectura de este libro delante del ordenador, probando los ejemplos y resolviendo los ejercicios propuestos.



# 1. Introducción a la programación

## 1.1 Lenguajes de programación

Un lenguaje de programación es un lenguaje artificial que permite escribir un programa (conjunto de instrucciones que interpreta y ejecuta un ordenador). Por ejemplo, un sistema operativo, un navegador de Internet o un procesador de textos son programas. Para que un ordenador pueda “entender” y hacer lo que dice un programa, sus instrucciones tienen que estar codificadas en lenguaje binario, o lenguaje máquina, compuesto únicamente por dos símbolos: 0 y 1.

Sin embargo, las personas están acostumbrados a expresarse en un lenguaje natural y escribir los programas directamente en binario resultaría una tarea terriblemente ardua y propensa a errores. Por ejemplo, el código para escribir simplemente Hola es 01001000 01101111 01101100 01100001. Para facilitar esta tarea, los lenguajes de programación se parecen más al lenguaje humano y hacen uso de un traductor que convierte lo escrito en código binario.

Cuando un lenguaje de programación utiliza signos convencionales cercanos a los de un lenguaje natural se dice que es de alto nivel. En cambio, si el lenguaje de programación es similar al lenguaje máquina se dice que es de bajo nivel.

El lenguaje de más bajo nivel es el lenguaje ensamblador, que es muy parecido al lenguaje máquina pero con pequeñas modificaciones mnemotécnicas que facilitan su uso. Por ejemplo, el fragmento siguiente es de un programa escrito en ensamblador (procesadores INTEL 80XX/80X86e) que imprime en la pantalla del ordenador el mensaje Hola Mundo:

```
DATA      SEGMENT
SALUDO    DB          "Hola Mundo",13,10,"$"
DATA      ENDS
CODE      SEGMENT
          ASSUME CS:CODE, DS:DATA, SS:STACK
INICIO:
          MOV  AX,DATA
          MOV  DS,AX
          MOV  DX,OFFSET SALUDO
          MOV  AH,09H
          INT  21H
          MOV  AH,4CH
          INT  21H
CODE      ENDS
          END  INICIO
```

En cambio, los lenguajes de alto nivel son más parecidos al lenguaje natural y por tanto más fáciles de leer y escribir. Algunos de los más conocidos son ADA, Modula-2 o Pascal. Entre los lenguajes de alto nivel y el ensamblador existen otros lenguajes, denominados de nivel medio, que combinan las características de los primeros (control del flujo de ejecución de los programas, programación estructurada, funciones, etc.) con operaciones propias del lenguaje ensamblador, como la manipulación de bits, direcciones de memoria, etc. Entre éstos podemos encontrar a C, C++ o Java.

Por ejemplo, el siguiente programa está escrito en lenguaje C y también imprime en la pantalla el mensaje Hola Mundo:

```
#include <stdio.h>

void main()
{
    printf ("Hola Mundo\n");
}
```

Comparando este programa con el anterior puede verse que es más corto, sencillo e inteligible. Incluso sin saber nada del lenguaje C, parece que la línea que empieza por `printf()` tiene algo que ver con la impresión del mensaje “Hola Mundo”.

Entonces, ¿por qué se programa en ensamblador si es más fácil hacerlo con lenguajes de alto nivel? La respuesta es la velocidad, ya que los programas se ejecutan más rápido. Además, se aprovechan mejor los recursos hardware del ordenador porque los programas ocupan menos espacio en la memoria. Sin embargo, no todo son ventajas, los programas son más largos y hay que aprender un lenguaje ensamblador específico para cada procesador.

En cambio, los lenguajes de alto nivel permiten escribir programas sin necesidad de conocer el procesador que utiliza el ordenador. Además, el mismo programa se puede ejecutar en otro ordenador con distinto hardware si se dispone del traductor apropiado que convierta el código fuente (lo que escribe el programador) en el código binario requerido.

## **1.2 El lenguaje de programación C**

El lenguaje C fue inventado por Dennis Ritchie en los Laboratorios Bell en 1972 para ser usado con el sistema operativo UNIX. C es hijo del lenguaje B, escrito en 1970 por Ken Thompson para el primer sistema operativo UNIX, y nieto del lenguaje BCPL, creado por Martin Richards en los años 60. El nombre de C es porque viene después del B, aunque el que sigue a C no se llama D, sino C++ (que es básicamente C al que se han añadido clases para formar un lenguaje de programación orientado a objetos).

Hasta los años 80, aunque existían compiladores para diversos sistemas operativos y arquitecturas de ordenadores, C siguió estando ligado casi exclusivamente con UNIX. En 1989, tras seis años de trabajo, C fue convertido en el estándar ANSI C. Los trabajos de estandarización no se detuvieron aquí y en 1999 se publicó un nuevo estándar de C conocido como C99. Por tanto, aunque su origen es relativamente antiguo comparado con la rápida evolución de los ordenadores, C se ha mantenido en la vanguardia de los lenguajes de programación no orientados a objetos y en la actualidad sigue siendo muy popular, no sólo entre los programadores profesionales, sino también entre científicos e ingenieros que necesitan una herramienta de cálculo numérico.

C es un lenguaje de propósito general, compacto (sólo tiene 32 palabras) y fácil de aprender, que combina la programación estructurada con la eficiencia y velocidad del ensamblador. Su principal ventaja es que permite programar en diferentes niveles, desde los drivers de un dispositivo hardware del ordenador hasta un sistema operativo. Por ejemplo, UNIX o DOS están escritos en C. Además, aprender C proporciona una buena base para abordar después la programación orientada a objetos con C++.

Este libro realiza una introducción a C que aborda los aspectos esenciales de este lenguaje desde un enfoque eminentemente práctico haciendo especial hincapié en el manejo de las sentencias de control y en el uso de funciones, que son la base de la programación estructurada.

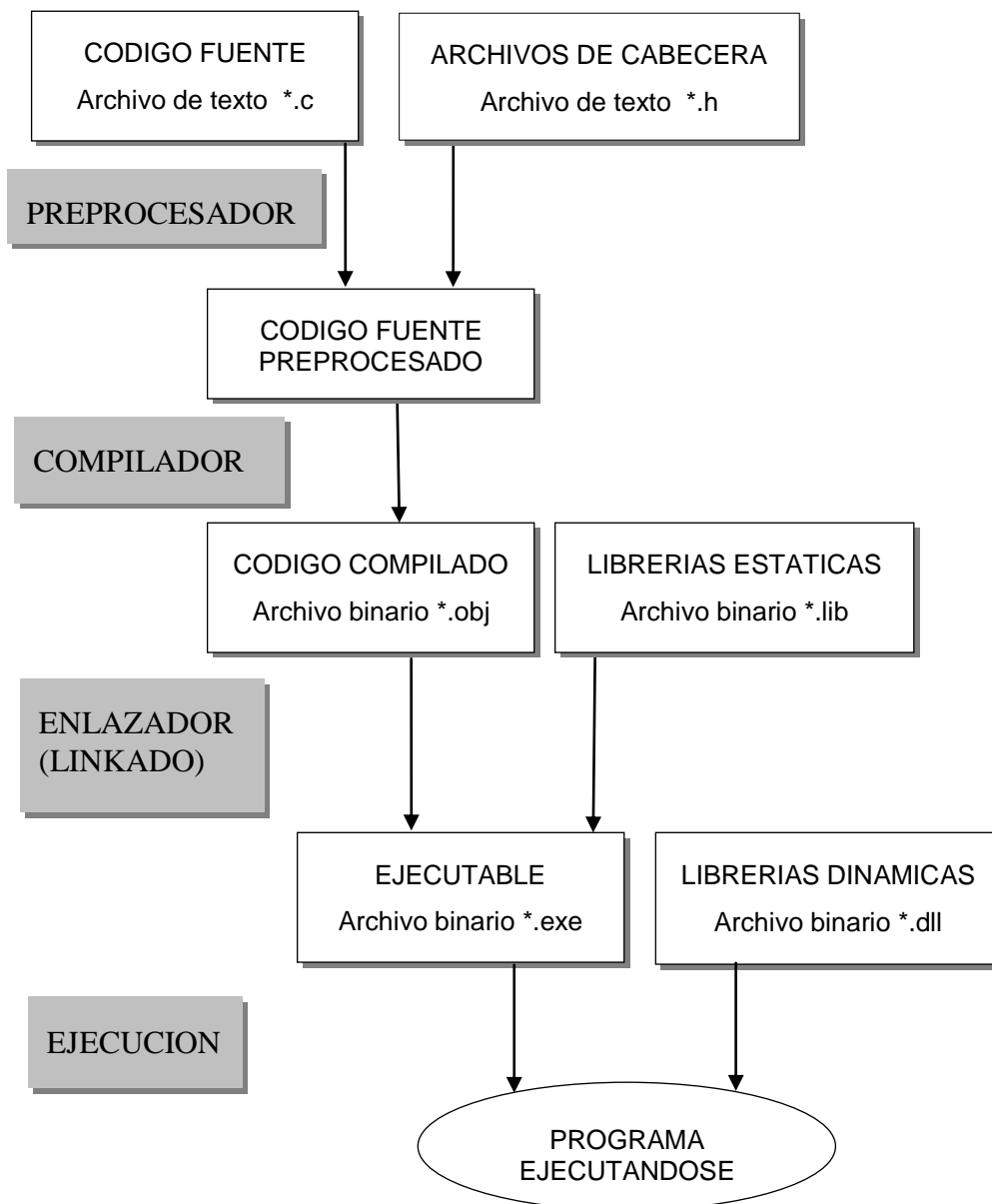
Resolver un problema con el ordenador, mecanizar una solución, automatizar un proceso, etc. son conceptos semejantes que entrañan escribir una serie de órdenes que el ordenador debe entender y ejecutar correctamente. Lo mismo que el lenguaje y los gestos permiten la comunicación entre los seres humanos, los lenguajes de programación permiten la comunicación del hombre con la máquina.

### **1.3 Proceso de creación de un programa ejecutable**

Como se ha descrito anteriormente, el computador solo entiende código máquina, compuesto por dígitos binarios. El proceso de transformar un archivo de código fuente de un lenguaje de programación (como el C) a código máquina se llama construcción (build) y consta generalmente de las siguientes etapas:

- **Preprocesado:** El fichero de código que teclea el programador es un fichero de texto, como una carta o un libro. No es entendible por el computador para su ejecución. El primer paso es el preprocesado, que procesa el archivo fuente, generando una versión del mismo (no necesariamente en el disco duro, sino en memoria) en el que se han realizado algunas tareas básicas como sustituir algunas partes del código por otras equivalentes, o analizar dependencias entre ficheros. El resultado final sigue siendo texto, no entendible por el computador.
- **Compilado:** El compilado transforma el archivo preprocesado en un archivo binario de código máquina que generalmente es visible en el disco duro (con la extensión .obj en Windows y .o en Linux). Estos archivos ya son parte del programa binario que puede ser ejecutado por el procesador. Pero son solo una parte, no es el programa completo.
- **Enlazado estático (linkado):** Se necesitan algunos otros fragmentos o funciones de código, que ya están compilados en código binario en librerías estáticas (con la extensión .lib en Windows y .a en Linux). El enlazador o linker realiza la composición del archivo ejecutable (extensión .exe en Windows, sin extensión en Linux). Los fragmentos de la librería necesarios, quedan embebidos dentro del ejecutable, por lo que no es necesario adjuntar dichas librerías para su ejecución.
- **Enlazado dinámico:** Cuando se comienza a ejecutar un programa binario es posible que se necesiten algunos otros fragmentos de código que no fueron enlazados en la etapa anterior y que se encuentran en las llamadas librerías dinámicas o compartidas (con extensión .dll en Windows y .so en Linux). Cuando dichas librerías dinámicas se encuentran en el Sistema Operativo del computador (bastante a

menudo) no es necesario distribuirlas con el ejecutable. Cuando se produce el enlazado dinámico (llevado a cabo por el sistema operativo), el programa se ha convertido ya en un proceso completo en ejecución.



**Figura 1-1 Proceso de creación de un programa**

Entre los compiladores de C existentes destacan el de Microsoft (Visual C), el de Borland (Builder) y el GCC, el más generalizado en sistemas Unix y Linux. Para escribir un programa C puede utilizarse cualquier editor, como Notepad.exe (o bloc de notas), etc. ó algunos otros específicamente creados para escribir y manejar programas fuentes en C.

### **1.4 Tipos de errores.**

Los errores de un programa se pueden dividir básicamente en dos tipos, errores en tiempo de desarrollo y errores en tiempo de ejecución:

- Errores en tiempo de desarrollo. Cuando se encuentran estos errores, el archivo ejecutable no puede llegar a ser. Estos errores pueden ser a su vez de dos tipos:
  - o Errores de sintaxis: Cuando se comete un error denominado de sintaxis en la escritura de un programa, el compilador lo detecta e informa al usuario. Es como una falta de ortografía en el programa. Estos errores son los más fáciles de detectar y subsanar.
  - o Errores de linkado: El código es sintácticamente correcto, pero sin embargo falla la creación de un ejecutable porque falta algún trozo de código (por ejemplo una librería), o se ha omitido algún archivo o función. Estos errores también son relativamente sencillos de detectar.

En tiempo de desarrollo, el compilador también puede emitir avisos o “warnings” que informan de anomalías que no son fatales para la compilación, pero pueden ser origen de fallos en tiempo de ejecución. Los avisos no impiden la creación del ejecutable.

- Errores en tiempo de ejecución. El ejecutable se construye correctamente, pero sin embargo falla cuando se ejecuta. La detección de estos fallos es mucho más complicada. Es muy común el uso de herramientas de depuración (debugger) que permiten ejecutar un programa paso a paso, visualizando los datos internos al mismo, para analizar y descubrir los errores. Este fallo puede ser debido a diversas causas, con distintos efectos. Por ejemplo, podemos equivocarnos en una formula, con lo que el resultado será incorrecto. Pero también podemos cometer un error fatal, por ejemplo intentando acceder a datos ajenos al propio programa o intentando dividir por cero, en cuyo caso el programa suele abortar o finalizar súbitamente de forma anormal.

## 1.5 Ingeniería del software

Realmente, la programación es solo una pequeña parte de lo que se conoce como Ingeniería del Software. La creación de software no es un proceso artístico realizado por un programador, y la programación no es una tarea creativa en la que se improvisan soluciones para el problema abordado. La ingeniería del software establece metodologías para el desarrollo efectivo de código, abarcando muchas áreas diferentes como análisis de requisitos, análisis del dominio del problema, diseño, implementación o programación, pruebas, validación, documentación, control de versiones, etc.

El lector debe de ser consciente que cuando programa no esta sino poniendo los ladrillos de un edificio, como haría un albañil. Pero un edificio necesita un buen proyecto con un buen diseño, realizado por un arquitecto. El albañil no improvisa la colocación de ladrillos, sino que se ciñe al proyecto y lo ejecuta, siguiendo lo especificado en el mismo y con un resultado que será revisado por el jefe de obra. Igualmente el programador tiene que seguir un proyecto y su trabajo no quedara para el solo, sino que tiene que ser revisado o incluso continuado o mejorado por otros programadores.

Como no se pretende realizar un libro sobre ingeniería del software, se da al lector una directriz básica: **Cuando se programa, hay que hacerlo pensando en que otra persona va a leer tu código, e incluso trabajar sobre tu código, ampliándolo o**

**mejorándolo.** Por lo tanto, el código tiene que ser muy legible. Para conseguirlo, hay que:

- Comentar los programas adecuadamente, explicando los detalles, pero sin entrar en obviedades.
- Respetar el estilo de código adoptado, con las tabulaciones, indexados, saltos de línea, etc.
- Utilizar una nomenclatura adecuada para variables y funciones. Por ejemplo si realizas un programa que calcula el área de un círculo, y tienes una variable, mejor llamarla “radio”, que “r” o “x”.
- Generalizar los programas cuando esto sea posible. En muchos casos cuesta el mismo esfuerzo hacer un programa más versátil y general que uno particular. Por ejemplo, cuesta lo mismo hacer un programa que maneje un vector de datos de 10 elementos, que uno que maneja un vector de cualquier número de elementos.
- Tener en cuenta que el hecho de que un programa funcione, en el sentido que muestra por pantalla una solución que parece correcta, no garantiza que el programa este bien realizado y que la solución sea buena.



# 2. Introducción al lenguaje C

## 2.1 Introducción

En este capítulo se aborda el desarrollo de los primeros programas en lenguaje C. Su contenido, al igual que el resto de este libro está centrado en el lenguaje, y no se incluye información relativa a un entorno de desarrollo concreto, sino que los programas aquí presentados deberían de poder ser escritos, compilados y ejecutados en cualquier computador con cualquier sistema operativo, que incluya herramientas de desarrollo de C.

Este capítulo presenta a menudo programas completos y funcionales. Inicialmente, no se pretende que el lector entienda el porqué de todas las partes de cada programa, sino que se vaya familiarizando con los aspectos generales del lenguaje. Por ejemplo, se describe brevemente la forma de mostrar información por pantalla y capturar información tecleada por el usuario. Para ello hay que utilizar funciones, que realmente son descritas en un capítulo muy posterior. La única forma de poder realizar los primeros programas es aprender la forma de realizar dicha entrada y salida a modo de receta, sin profundizar en su uso.

Sin embargo si que se tratan a fondo aspectos como la sintaxis general del lenguaje, el formato y estilo del código, las constantes y las variables, y los operadores que permiten manipular datos.

Al finalizar este capítulo, el lector debe de ser capaz de realizar programas sencillos que soliciten algunos datos al usuario, realicen unos cálculos matemáticos simples con ellos y muestren el resultado por pantalla.

## 2.2 Primer programa en C: Hola mundo.

Como es común en el aprendizaje de lenguajes de programación, se comienza presentando en esta sección el código de un programa muy simple, que lo único que hace es mostrar por pantalla el mensaje “Hola mundo”.

```
#include <stdio.h>

void main()
{
    printf("Hola mundo\n");
}
```

Teclee el siguiente programa en su editor, compílelo y ejecútelo para ver su resultado. Si lo ha hecho correctamente, la salida será la siguiente.

**Hola mundo**

Vamos a analizar línea por línea este programa:

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

Esta línea es necesaria en nuestro programa para poder utilizar funciones de entrada y salida estándar. La entrada y salida estándar es la consola, en la que se pueden imprimir mensajes así como leer lo que teclea el usuario. El significado intuitivo de esta línea de código es decir que “incluya” (`include`) el fichero `stdio.h` (std=estandar, io=input, output, entrada y salida; stdio: entrada y salida estándar).

```
void main()
```

Esta línea define el punto de entrada al programa, esto es, la función `main()` o principal. Cualquier programa en C necesita la existencia de una función denominada `main()` y solo una función puede ser denominada así.

```
{
```

La llave de apertura define el comienzo de un nuevo bloque de sentencias. En este caso esta llave define el comienzo del bloque de sentencias correspondiente a la función `main()`.

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

Esta línea realiza una llamada a la función `printf()`, que sirve para imprimir mensajes de texto en la consola. Esta función se encuentra declarada en el fichero `stdio.h`, y por eso ha sido necesaria la primera línea del programa con `#include <stdio.h>`. Nótese que la línea acaba con un punto y coma indicando el final de una **sentencia**.

```
}
```

La llave de cierre indica el final de un bloque de sentencias, en este caso el bloque de sentencias de la función `main()`. Cuando el programa alcanza el final de la función `main()`, entonces termina su ejecución.

Todos los programas se ejecutan en secuencia, una sentencia detrás de otra, de arriba hacia abajo. Esto no quiere decir que el usuario sea capaz de percibirlo, ya que el computador ejecuta las ordenes generalmente muy rápidamente, pero el programador tiene que tenerlo en cuenta. Nótese que en el siguiente ejemplo se ha omitido la palabra `void` antes de `main()`, lo que es aceptado igualmente por el compilador.

```
#include <stdio.h>
main()
{
    printf("Hola mundo\n");
    printf("Que tal estas\n");
    printf("Adios mundo\n");
}
```

```
Hola mundo
Que tal estas
Adios mundo
```

Cada sentencia es delimitada por un punto y coma y generalmente se suele poner una por cada línea, aunque esto no es imprescindible, es solo cuestión de estilo. Un conjunto de sentencias se pueden agrupar en un bloque de sentencias, delimitado por una llave que abre al principio del bloque y otra llave que cierra al final.

El lenguaje C siempre distingue entre mayúsculas y minúsculas. De tal forma, son diferentes: METRO, Metro y metro.

## 2.3 Elementos del lenguaje C

### 2.3.1 Comentarios

Todo programa, escrito en cualquier lenguaje de programación, debe ser sencillo y claro. Para ello se emplean comentarios descriptivos, los cuales no son más que texto que es totalmente despreciado por el compilador y por lo tanto no forman parte del ejecutable. Solo son útiles para las personas que leen el programa fuente. En C los comentarios de una línea van precedidos de una doble barra `//`, mientras que los comentarios que abarquen varias líneas en el programa fuente van precedidos por `/*` y terminan con `*/`. Puede haber tantos como se crea necesario e incluso pueden escribirse a la derecha de una sentencia.

```
//comentario de una linea
#include <stdio.h>

/*Este es un comentario de
varias lineas, explicando lo
que sea necesario*/
void main() //aquí puede ir un comentario
{
    //aquí otro comentario
    printf("Hola mundo\n"); //aquí también puede ir otro comentario
}
```

### 2.3.2 Separadores

Un separador es un carácter o conjunto de caracteres que separan elementos del lenguaje. Los separadores puede ser espacios, retornos de carro (nueva línea), tabulaciones. Los comentarios también podrían considerarse como separadores. Un separador delimita dos elementos diferentes, pero **no** tiene ningún efecto sobre la sintaxis. Los separadores se usan típicamente para hacer más legible el código. Como ejemplo, el programa anterior podía haber sido escrito de las siguientes formas, añadiendo o quitando espacios y retornos de carro, siendo el ejecutable binario generado al compilar exactamente el mismo.

<pre>#include &lt;stdio.h&gt; void main() {     //aquí comentario     printf("Hola mundo\n"); }</pre>	<pre>#include &lt;stdio.h&gt; void main() {     //aquí comentario     printf("Hola mundo\n");}</pre>
---	--

```
#include <stdio.h>
void main()
{
printf ("Hola mundo\n");
}
```

```
#include <stdio.h>

void main(){printf ("Hola mundo\n");}
```

### 2.3.3 Palabras clave

Se dispone de 32 palabras clave (definidas en el estándar ANSI) en el lenguaje C. Estas palabras suelen venir resumidas en un cuadro en orden alfabético. No obstante, presentamos aquí un cuadro con las palabras claves clasificadas por su función:

**Tabla 2-1. Palabras clave de C**

Modificadores de tipo	Tipos primitivos	Tipos datos avanzados	Sentencias de control
auto	char	enum	break
extern	double	struct	case
register	float	typedef	continue
static	int	union	do
const	long		else
volatile	short		if
signed	void		for
unsigned	sizeof		while
			return
			goto
			default
			switch

Como se aprecia, las tres primeras columnas están relacionadas con los tipos de datos, mientras que la última tiene que ver con el flujo de ejecución del programa. Cuando programamos en C tenemos que decir al compilador los tipos de datos que utilizamos. así, si vamos a utilizar una variable que representa la edad de una persona, esa variable será de tipo entero (18, 24, 33 años), y lo haremos con la palabra clave ‘`int`’ abreviatura de “integer”, entero en inglés. Sin embargo, si queremos utilizar una variable que representa la altura de una persona en metros (1.85, 1.68 m), tendremos que utilizar decimales y lo haremos con la palabra clave ‘`float`’ o ‘`double`’ en función de la precisión decimal que queramos. Las tres primeras columnas sirven para este estilo de cosas. La última está relacionada con la forma de ejecutar el programa. Si queremos que el programa tome una decisión condicional utilizaremos ‘`if`’, mientras que si queremos que repita varias veces la misma cosa podremos utilizar ‘`for`’, ‘`while`’, etc.

### 2.3.4 Identificadores

Un identificador es un nombre simbólico que se refiere a un dato o a una parte de un programa. Los identificadores son pues nombres asignados a datos (variables o constantes) y funciones.

Los identificadores en C siguen unas reglas:

1. Los identificadores solo pueden contener caracteres alfanuméricos, mayúsculas o minúsculas, y el carácter “underscore” o ‘`_`’. El alfabeto es el inglés, así que la letra ‘ñ’ está excluida.

A-Z, a-z, 0-9, \_

2. No pueden contener el caracter espacio ni caracteres especiales ni de puntuación.
3. El primer caracter no puede ser un dígito, esto es, debe comenzar por una letra o el caracter ‘\_’.
4. Se distinguen mayúsculas de minúsculas. Así el identificador ‘masa’ es distinto del identificador ‘MASA’ y de ‘Masa’ y de ‘MaSa’, etc.
5. El estándar ANSI C admite un máximo de 31 caracteres de longitud, aunque en la practica a veces se permite utilizar identificadores de más caracteres.
6. No se pueden utilizar palabras clave reservadas como ‘long’, ‘auto’, ‘const’, etc.

A continuación presentamos algunos ejemplos de identificadores válidos y no válidos:

**Tabla 2-2. Ejemplos de identificadores válidos y no válidos**

No válidos (motivo)		Sí válidos
1valor	Empieza por dígito	UnValor
2_valor	Empieza por dígito	TiempoTotal tiempototal
tiempo-total	Caracter – (menos) no permitido	tiempo_total
dolares\$	Caracter no permitido	dolares
continue	Palabra clave	continuar
%final	Caracter no permitido	_continue

Aunque no son reglas, existen unas buenas costumbres seguidas por los programadores:

1. Se recomienda utilizar identificadores o nombres significativos. así para representar una masa podríamos utilizar el identificador ‘masa’ mejor que el identificador ‘m’. Ambos serian válidos, pero el primero haría el programa más legible.
2. Cuando se utilizan variables auxiliares o con un sentido matemático, se recomienda utilizar nombres de identificadores similares a sus homólogos matemáticos. así, si en matemáticas denotamos la componente de un vector ‘v’ con el subíndice ‘i’, ‘vi’, utilizaremos el identificador ‘i’ para contadores, repeticiones e indexar vectores. Asimismo, utilizaremos identificadores ‘i’, ‘j’, ‘k’ para contadores, bucles anidados o indexar matrices. Utilizaremos los identificadores ‘x’ e ‘y’ cuando nos refiramos a coordenadas espaciales o valores de variable independiente e independiente de una función matemática.

### 2.3.5 Constantes

Las constantes son datos del programa que nunca cambian y que no pueden ser modificados en tiempo de ejecución. Estas constantes pueden ser numéricas (enteras, decimales, hexadecimales, octales), caracter y cadenas de caracteres.

En las constantes enteras el tipo `int` se utiliza si el número es lo suficientemente pequeño y `long` es usado automáticamente si su valor es demasiado grande. Para explicitar el formato `long` se le añade una `L` al final del número.

34            constante tipo `int`

34L          constante tipo `long`

También se pueden utilizar constantes en formato octal, precediendo el número (que solo puede contener los dígitos 0-7) por un cero. De la misma forma, se pueden utilizar constantes hexadecimales, precediendo al número (compuesto por dígitos 0-9 y letras A-F, que pueden ser minúsculas o mayúsculas) por `0x` ó `0X`

07            constante octal

0xff         constante hexadecimal

0XA1         constante hexadecimal

Las constantes reales, son por defecto de tipo `double`, a no ser que se diga explícitamente lo contrario con la letra `f` ó `F`. También es permisible la notación científica mediante la letra `e` o `E`, seguida del exponente (base 10)

2.28         constante `double`

1.23f         constante `float`

0.234         constante `double`

.345         constante `double`

.67F         constante `float`

123e3         constante `double` (123000.0)

.23E-2f        constante `float`(0.0023f)

Las constantes de caracteres se representan entre comillas simples:

'a'            constante caracter (letra a)

'+'            constante caracter (símbolo +)

Las constantes de cadenas de caracteres se limitan por comillas dobles:

"Hola mundo"

### 2.3.6 Operadores

Los operadores son símbolos especiales(+, \*, etc.) que permiten hacer tareas sobre variables como sumas, restas, multiplicaciones, etc. Estos operadores son explicados en detalle más adelante en este mismo capítulo.

## 2.4 Variables y constantes

Las variables son los manejadores básicos de datos en un programa. Variable es un campo de memoria con un nombre identificativo, que puede almacenar valores diferentes a lo largo de la ejecución del programa. Al nombre dado a una variable se le llama identificador y tiene que cumplir las normas establecidas para los identificadores en el apartado 2.3.

Las constantes son valores fijos (números) con los que se puede: hacer asignación a variables, operar con variables y constantes, etc. y en general utilizarse a lo largo de la ejecución de un programa.

En C existen además las constantes simbólicas, establecidas mediante la palabra clave `const` y cuyo valor no puede ser modificado en el código.

```
void main()
{
    const int c=4; //declara una constante simbolica de tipo entero
                  //denominada 'c' y que vale 4
    int a; //declara una variable de tipo entero, denominada 'a'
    a=3;   //el numero 3 es una constante, 'a' toma el valor 3
    a=8;   //el numero 8 es una constante, 'a' toma ahora el valor 8
          // 'a' es variable
    a=c;   //esto es correcto, ya que 'a' variable toma el valor 4
          //pero la constante 'c' no es modificada
    c=5;   //esto es un error de sintaxis, ya que intenta modificar
          //la constante 'c'
}
```

### 2.4.1 Tipos de variables, declaración e inicialización

Toda variable que vaya a ser utilizada en un programa debe ser previamente declarada. Declarar una variable es definir su tipo, asignarle un nombre y opcionalmente asignarle un valor inicial. El tipo indica los valores que puede tomar (enteros, decimales, caracteres, etc.) y las operaciones que están permitidas, con el objeto de ahorrar espacio en memoria.

Los tipos de datos enteros son: `char`, `short`, `int`, `long`

Los tipos de datos reales son: `float`, `double`

Se entiende por enteras aquellas cantidades que no tienen decimales ni tan siquiera punto decimal. Real (decimal o de coma flotante) son las que tienen parte entera y fraccionaria y aunque ésta no exista (sea cero), al menos tienen punto decimal. Un caracter no es más que un número entero correspondiente a un símbolo de la tabla ASCII. El manejo de caracteres y cadena de caracteres se deja para un capítulo posterior, centrándonos ahora en el uso de los tipos numéricos.

La forma de declarar una variable es la siguiente, donde los corchetes indican que es una parte opcional:

```
<tipo> <identificador> [=valor inicial];
```

Ejemplos de declaración:

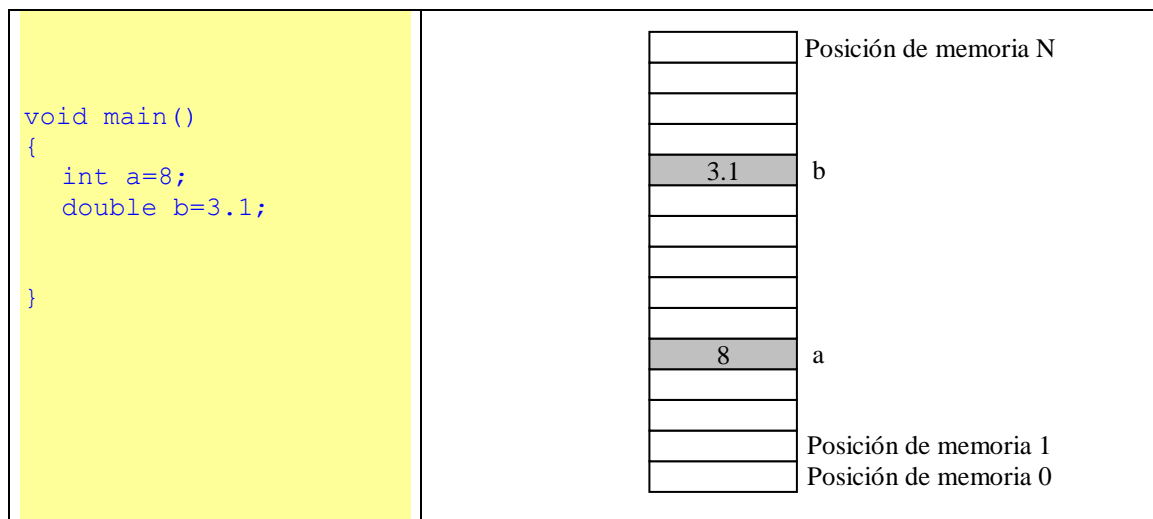
```
void main()
{
    int a;           //declara una variable de tipo entero denominada 'a'
    double b=3.1;    //declara una variable real de precision doble,
                    //cuyo valor inicial es 3.1
}
```

```

a=4;           //correcto
b=14.567;      //correcto
c=23;          //error de sintaxis, variable 'c' no declarada
int d;         //error de sintaxis,
}

```

Cuando se declara una variable, realmente se le dice al computador que reserve una posición de memoria, a la que se denominara con el nombre de la variable, de tal forma que cuando se lea o se escriba en esa variable, realmente se lea o se escriba la información en esa posición de memoria. Entre todas las posiciones de memoria disponibles, es el computador (realmente el sistema operativo) el que asigna una posición libre, sin que el programador pueda decidir en que posición. Haciendo un símil: la memoria es un armario con muchos cajones, cuando se declara una variable se reserva automáticamente un cajón, al que se le pone una etiqueta con el nombre de la variable. El contenido del cajón serán los datos (números, letras) que se guardan en la variable. En función del tipo de dato, el cajón tendrá un tamaño, por ejemplo, si la variable es de tipo `float`, el cajón ocupara realmente 4 bytes de memoria, mientras que si es de tipo `double`, ocupara 8 bytes en memoria.



**Figura 2-1. Reserva de memoria al declarar variables**

En C, cuando las declaraciones de variables se realizan en un bloque de sentencias, delimitado por llaves, dichas declaraciones tienen que ir siempre ubicadas al principio, siendo un error de sintaxis intentar declararlas en otro lugar. En cada bloque de sentencias la zona de declaraciones termina con la primera línea que no es una declaración.

```

void main()
{
    //principio de bloque de sentencias
    int a;      //declaracion
    a=4;        //esta ya no es una declaracion

    {
        //esto es un nuevo bloque de sentencias
        int c=4; //se permiten nuevas declaraciones
        c=8;     //esto ya no es una declaracion

        int d; //error de sintaxis
    }
}

```



La diferencia entre los distintos tipos de datos enteros y reales, es el tamaño que ocupan en memoria, o lo que es lo mismo, el número de bytes que utilizan para codificar el valor correspondiente. En función de dicho tamaño, el rango de valores numéricos admisibles varia. En general se puede afirmar que el tamaño `char` ≤ `short` ≤ `int` ≤ `long` y que el tamaño `float` ≤ `double`.

En los valores enteros se pueden añadir los especificadores de tipo `signed` o `unsigned`, indicando si las variables pueden tener signo (enteros) o no (naturales).

El tamaño y rango de los diferentes tipos de datos depende del computador, el SO y el compilador concretos que se utilicen. La siguiente tabla resume, suponiendo un ordenador de 32 bits con WinXP y Visual C 6.0, los tamaños asignados a cada tipo de variables y los valores máximos posibles de almacenar en ellas son los indicados en la Tabla 2-3.

**Tabla 2-3. Tipos de datos básicos**

Signo	Tipo	Tamaño	Rango (min)	Rango (max)
	char	1 byte	-128	127
unsigned	char	1 byte	0	255
	short	2 bytes	-32768	32767
unsigned	short	2 bytes	0	65535
	int	4 bytes	-2147483648	2147483647
unsigned	int	4 bytes	0	4294967295
	long	4 bytes	-2147483648	2147483647
unsigned	long	4 bytes	0	4294967295
	float	4 bytes	-3.4E 38	3.4E 38
	double	8 bytes	-1.7E 308	1.7E 308

Además del rango, las variables de tipos reales `float` y `double` tienen distinta precisión, debido a la codificación interna que tienen. Según el formato IEEE 754, un número `float` de precisión simple ocupa 4 bytes (32bits) y utiliza 1 bit para el signo, 8 para el exponente y 23 para la mantisa. En la practica esto se traduce a una precisión de aproximadamente 7 dígitos significativos. El número `double` ocupa 8 bytes (64 bits), 1 para el signo, 11 para el exponente y 52 para la mantisa, lo que en la practica significa una precisión de 16 dígitos significativos.

Varias variables del mismo tipo se pueden declarar e inicializar en las misma línea separando con comas, con la siguiente sintaxis:

```
<tipo> <identificador_1> [=valor1], <identificador_2> [=valor2], ... ;
```

Ejemplos de declaración:

```
//Ejemplo de declaracion en la misma línea
void main()
{
    int a=3, b, c=8, d;      //declara a,b,c y d, inicializa a y c
    double e, f=1.2;
    int g=123, char h=123; //Esto es un error de sintaxis
}
```

El rango de valores no es comprobado por el compilador ni por el computador durante la ejecución, con lo que se pueden producir desbordamientos por arriba o por abajo.

```
void main()
{
    unsigned char c=255; // c es un numero entero
    c=c+1;                // c pasa a valer 0
    c=0;
    c=c-1;                // c pasa a valer 255
}
```

Para almacenar letras o caracteres se utiliza el tipo de datos `char` (o `unsigned char`). Dado que este tipo de datos es de números enteros, lo que realmente se guarda en la variable declarada es el número entero correspondiente al caracter, tal y como se define en la Tabla ASCII.

```
void main()
{
    char letra='a';      //letra=97
    char simbolo='+';    //simbolo=43
}
```

## 2.4.2 Modo de almacenamiento y ámbito de una variable

Por su modo de almacenamiento las variables se dividen en globales y locales (o automáticas). Una variable global existe desde el punto en el que se declara hasta el final del fichero en el que se declara, mientras que una variable local existe desde el punto en el que se declara hasta el final del bloque de sentencias en el que se ha declarado.

En general en este libro se tratará de reducir al máximo el uso de variables globales, ya que su uso no es muy recomendable. Se volverá sobre este tema al hablar de las funciones.

```
int x=3; //la variable x es global

void main()
{
    int y=5; //variable y es local a la funcion main
    //sentencias

    x=145; //correcto, ya que estamos dentro del ambito de la variable x
    { //bloque de sentencias
        int z=3; //variable z es local a este bloque de sentencias
        ...
        ...
        y=17; //notese que tenemos acceso a la variable y
    }

    z=14; //esto es un error de sintaxis, ya que el ambito de z
        //se limita al bloque anterior
}

Ambito de z
Ambito de y
Ambito de x
```

## 2.5 Operadores

Los operadores son símbolos (+, \*, etc.) que permiten hacer tareas sobre variables y constantes, produciendo un resultado como fruto de la operación sobre uno, dos o tres **operandos**. Los operadores pueden ser **unarios**, es decir aplicar a un único operando, **binarios** (2 operandos) y **ternarios** (3 operandos).

Estos operandos pueden ser **numéricos** (`int`, `float`, `double`, `char`, etc.) o **booleanos** (1-verdadero, 0-falso). El resultado de la operación puede ser igualmente numérico o booleano.

### 2.5.1 Operadores aritméticos

Los operadores aritméticos existentes en C son los siguientes:

- + **suma**
- **resta**
- \* **multiplicación**
- / **división**
- % **módulo o resto de la división entera**

Son operadores binarios que aceptan dos operandos de tipo numérico, devolviendo un valor numérico del mismo tipo de los operandos. El operador modulo % solo puede ser aplicado a datos de tipo entero. Aplicarlo a constantes o variables de tipo real (`float` o `double`) es un error de sintaxis.

Estos operadores intervienen típicamente en el lado derecho de una asignación. Si uno de los operadores también esta en el lado izquierdo, se debe tener en cuenta que primero se evalúa el lado derecho y luego se realiza la asignación.

También hay que tener en cuenta que los números no tienen precisión infinita, así, se producirán redondeos por ejemplo en las operaciones de números de tipo real, redondeando el dígito menos significativo, según las reglas de redondeo matemático.

El orden de evaluación (prioridad) es similar al matemático. Las multiplicaciones y divisiones tienen más preferencia que las sumas y restas. Nótese bien que no existe operador de C para realizar la potencia de un número.

```
//Ejemplos operadores aritmeticos

int a=3, b=5, c;
double d=3.0 ,e=5.0, f;
float r;

c=a+b; //c valdra 8, a y b no modifican su valor
c=a*6; //c valdra 18, a no modifica su valor
c=b/a; //division entera, c valdra 1
c=a/b; //division entera c vale 0
c=b%a; //resto, c valdra 2
c=a*b+2; //c vale 17, primero a*b=15 y luego se suma 2

f=d/e; //f valdra 0.6
f=d%e; //Error de sintaxis

a=a+b; //a pasaria a valer 3+5, es decir 8

r=2.0f/3.0f; //r vale 0.666667
```

## 2.5.2 Operadores relacionales

Los operadores relacionales efectúan operaciones de comparación entre datos numéricos (variables y constantes). El resultado de un operador relacional es booleano, es decir, devuelve 0 si es la relación falsa y un 1 (o distinto de 0) si es verdadera. El resultado de estas operaciones se puede asignar por tanto a variables de tipo entero (`int`, `char`). Los operadores relacionales son:

- <**    menor que
- >**    mayor que
- <=**   menor o igual que
- >=**   mayor o igual que
- ==**   igual a
- !=**   distinto a

```
//Ejemplos operadores relacionales

int x=10, y=3, r;

r = (x==y); //primero se compara si x es igual a y, lo que es falso (0)
           //el resultado de la operación se asigna a r, que vale 0
r = (x>y);  //r vale 1
r = (x!=y); //r vale 1
r = (x<=y); //r vale 0
r = (x>=2); //r vale 1
```

Nótese bien que el operador de comparación de dos variables o constantes es un doble signo igual, que es diferente de la asignación con un solo signo igual.

## 2.5.3 Operadores incrementales

Los operadores incrementales, incrementan o decrementan un único operando (operador unario) de tipo numérico (generalmente tipo entero) en una unidad. Los operadores incrementales son:

- ++**   incrementa una unidad
- decrementa una unidad

En función de la posición relativa del operador respecto del operando se definen las siguientes operaciones:

```
a++; //post incremento      a--; //post decremento
++a; //pre incremento      --a; //pre decremento
```

El efecto final de los operadores sobre el operando es el mismo, incrementan o decrementan una unidad.

```
int a=0;
++a; //incrementa 1, ahora 'a' valdra 1
++a; //incrementa 1, ahora 'a' valdra 2
--a; //decrementa 1, ahora 'a' valdra 1
a--; //decrementa 1, ahora 'a' valdra 0
```

La diferencia es el valor devuelto por el operador en el caso en el que la expresión deba de ser evaluada en una sentencia con otras operaciones. En el caso de preincremento y predecremento el valor obtenido es igual al valor final del operando, ya que primero (pre) se hace la operación y luego se transfiere el resultado

```
int a=2,r;
r=++a; //primero se incrementa 'a', que pasa a valer 3
      //el resultado es el valor ya incrementado
      //con lo que 'r' vale tambien 3
```

Se podría decir que esta operación es equivalente a:

```
++a; // (pre) primero se incrementa
r=a; // luego se asigna
```

Cuando es un postincremento o postdecremento, primero se transfiere el resultado (valor de la variable) y luego se incrementa dicha variable.

```
int a=2,r;
r=a++; //primero se realiza la asignacion
      //luego 'r' vale 2
      //y luego (post) se incrementa 'a' que valdra 3
```

Se podría decir que esta operación es equivalente a:

```
r=a; // primero se asigna
a++; // (post) luego se incrementa
```

## 2.5.4 Operadores lógicos

Los operadores lógicos implementan las operaciones booleanas. Por lo tanto actúan sobre operandos booleanos y dan un resultado igualmente booleano. Las operaciones AND y OR son binarias (2 operandos) mientras que la negación NOT es unaria.

**&&    AND (y lógico)**

**||      OR (o lógico)**

**!       NOT (negación lógica)**

Los operadores lógicos se usan generalmente para combinar operadores relacionales formando expresiones lógicas complejas

```
float a=3.0f,b=2.0f;
int c=4,d=5,r;
r= (a>b) && (c<d);        //como 3.0>2.0 y 4<5, 'r' valdra 1
r= (a<11.3f) || (d!=5); //3.0<11.3, 'r' valdra 1, aunque 'd' valga 5
r= (a!=b) && (2*d < 8); // 'r' vale 0
```

También es posible que uno o ambos operandos sea una variable o constante numérica. En ese caso se evaluará como falso (0) si su contenido es 0, y como verdadero (1) si su contenido es cualquier otro.

```
int c=4,d=5,r;
r= !c;                    // 'c' es verdadero, luego 'r' sera falso (0)
r= (c) && (!d);        // 'c' es verdadero y 'd' tb. 'r' sera falso (0)
r= (c<d) || (!d); // 'r' valdra 1, ya que 'c' es menor que 'd'
```

## 2.5.5 Operadores de bits

Los operadores de bits son operadores binarios, a excepción de la negación (NOT) de bits que es unario. Todos ellos aplican a variables de tipo entero, dando como resultado otro número entero aplicando las operaciones lógicas correspondientes a los bits de los operandos.

- | **OR de bits**
- & **AND de bits**
- >> **desplazamiento de bits a la derecha**
- << **desplazamiento de bits a la izquierda**
- ^ **XOR (O exclusivo) de bits**
- ~ **NOT de bits**

```
unsigned char a=8, b=4, c; //a=0001000, b=00000100
c = a & b;                //c=00000000, luego c=0
c = a | b;                //c=00011000, c=12
c = a << 2;               //c=01000000, c=32
c = ~a;                   //c=11110111, c=247
```

Cuando se desplazan bits a la izquierda, se rellena con ceros por la derecha. Cuando se desplazan bits a la derecha, se rellena con ceros por la izquierda si el dígito más significativo es un cero, y se rellena con unos si es un uno y además el tipo de variable es con signo. Los dígitos que se desplazan fuera del tamaño de la variable se pierden.

## 2.5.6 Operadores de asignación

El operador básico de asignación es el signo '=', pero también se pueden realizar otras muchas operaciones ya vistas de forma combinada cuando aparece el mismo operando en el lado izquierdo y derecho de la asignación.

- = **asignación**
- += **incremento y asignación**
- = **decremento y asignación**
- \*= **multiplicación y asignación**
- /= **división y asignación**
- %= **módulo y asignación**
- <<= **desplazamiento de bits a izquierda y asignación**
- >>= **desplazamiento de bits a derecha y asignación**
- &= **AND lógico de bits y asignación**
- |= **OR lógico de bits y asignación**
- ^= **XOR lógico de bits y asignación**

El signo = debe interpretarse como asignación. La expresión `m=z` no se debe leer como "m igual a z" sino como "se asigna a m el valor de z". Los operadores dobles tipo `+=`, `-=`, `*=` etc., operan la expresión de la derecha del signo igual con la que aparece a la izquierda del operador. Por ejemplo `x*=y` es lo mismo que poner `x=x*y`. Cuando lo que aparece a la derecha del = es una expresión `x*=y+4` lo que significa es `x=x*(y+4)`.

Todas las asignaciones tienen dos partes, el lado izquierdo (left value) y el lado derecho (right value) de la asignación. En el lado derecho puede existir una expresión aritmética o lógica, pero en el lado izquierdo no puede haber una expresión, solo una variable.

```
//Ejemplo operadores asignacion

int a=8,b=3,c;
a=5;      //asigna el valor 5 a la variable 'a', luego a valdra 5
a*=4;     //multiplica por 4 a a. Es equivalente: a=a*4, 'a' valdra 20
a+=b;     //suma 'b' al valor de 'a'. Equivale a=a+b, luego 'a' sera 23
          //y 'b' no se modifica
a/=(b+1); //Equivale a=a/(b+1); division entera 'a' valdra 5
c=a%b;    //se asigna a 'c' el resto de dividir 'a' entre 'b'

a=b=c=7;  //asignacion simultánea de 7 a las tres variables
a+b=c;    //error de sintaxis, en el lado izquierdo no puede haber
          //una expresion.
```

### 2.5.7 Conversiones implícitas de tipo

Si los operandos son de distinto tipo, el operando de precisión menor promociona automáticamente al de precisión mayor.

Cuando se realizan operaciones con datos de distintos tipos, el ordenador iguala los rangos para poder dar un resultado. Los rangos son (de menor a mayor): 1-char, 2-short, 3-int, 4-long, 5-float, 6-double.

Se llama promoción al hecho de elevar el rango del dato que lo tiene menor. Al caso contrario se le llama pérdida. El resultado de la expresión se reconvierte (promoción o pérdida) al tipo de variable a que está siendo asignada.

```
int a=3, b=5, c;
double d=3.0 ,e=5.0, f;

f=a+e; //a se convierte automaticamente de 3 a 3.0 antes de ser sumado
       //el resultado final es f vale 8.0
c=d/b; //b se convierte de 5 a 5.0, el resultado de la division es 0.6
       //pero al asignarlo a un integer, al final c vale 0
```

### 2.5.8 Otros operadores

**? :**                    **operador condicional (ternario)**

El operador ? es el único operador ternario de C. Su sintaxis es la siguiente:

*resultado = condicion ? valor si cierto : valor si falso;*

Se evalúa la condición, que puede ser un operación de comparación mediante un operador relacional. Caso de que esta condición sea cierta, el resultado final es el primer valor antes de los dos puntos. Si la condición es falsa, el resultado final es el segundo valor después de los dos puntos.

```
int a=3,b=8,c;
c= a<b ? 6 : 7;    //como 3<8, el valor final de 'c' es 6
c= a==b ? a : a*b; //'a' no es igual a 'b', luego 'c' vale a*b=24
```

**sizeof**                    **tamaño de tipo o variable en número de bytes**

El operador `sizeof` permite conocer el tamaño que ocupan en memoria los diferentes tipos de datos y variables, etc. Su uso es muy simple, basta con poner entre paréntesis el tipo del elemento, o del identificador, del que se desea conocer su tamaño y el operador devuelve el tamaño en bytes.

*sizeof(tipo) ó sizeof(variable)*

Ejemplo:

```

int tam;
float a;
char b;
tam = sizeof (char);    //tam valdra 1
tam = sizeof (short);   //tam valdra 2
tam = sizeof (int);     //tam valdra 4
tam = sizeof (long);    //tam valdra 4
tam = sizeof (float);   //tam valdra 4
tam = sizeof (double);  //tam valdra 8
tam = sizeof (a);       //tam valdra 4
tam = sizeof (b);       //tam valdra 1

```

### **(tipo) cast o conversión forzada de tipo**

Cuando se realizan asignaciones u operaciones con distintos tipos de variables, se realizan conversiones implícitas de tipo en las que el compilador proporciona un aviso o “warning” de que puede haber pérdida de datos. Si realmente se quiere convertir un tipo a otro, y el programador puede forzar la conversión, sabiendo que se pueden perder datos en la misma.

```

int a=3;
float b=3.5;
a=(int) b;    //se fuerza la conversion de 'b' a entero (3)
              //luego 'a' valdra 3, y el compilador no da avisos

```

### **& dirección de \* indireccion**

Estos operadores están relacionados con los punteros, que serán tratados más profundamente en un tema posterior, y que se incluyen aquí únicamente por completitud, sin ánimo de que tengan que ser entendidos o manejados hasta llegar a dicho capítulo.

El operador & o dirección, aplicado a una variable devuelve la posición de memoria en la que se guarda o almacena dicha variable. El operador \* o indirección, aplicado a una dirección de memoria, devuelve el contenido almacenado en dicha dirección de memoria.

### **[ ] acceso a componente del vector . -> acceso a componente de una estructura mediante un puntero**

Estos operadores están relacionados con vectores o conjuntos de datos y con estructuras de datos. Estos tipos de datos avanzados se tratan en un capítulo posterior, donde se explicaran estos operadores.

## **2.5.9 Precedencia y asociatividad**

Cuando se combinan distintos operadores, el orden en el que se evalúan depende de la prioridad y asociatividad de los mismos. En el caso de los operadores aritméticos, la precedencia es intuitiva, ya que coincide con la notación matemática: la multiplicación y la división tienen prioridad respecto de la suma y de la resta.

Para modificar esta jerarquía natural o para aclarar la operación a realizar se emplean los paréntesis. Su uso correcto, aún en exceso, no afecta de forma negativa a la operación, pero su omisión puede resultar en un orden de evaluación no deseado por el



programador. Como se ve en la tabla, si en una expresión hay paréntesis, es lo primero que se hace.

```
int a=3,b=5,c=8,d;
d=a*b+3;           //primero a*b y luego +3, 'd' vale 18
d=a+c/2;           //primero c/2 y luego +a, 'd' vale 7
d=c/a+b;           //'d' vale 7, ojo a la division entera
d=c/(a+b);         //primero a+b por el parentesis, 'd' vale 1

d=(a+b)/(c-a);     //primero (a+b), luego (c-a), último la division
d=a+b/c+3;         //primero b/c, luego suma 'a' y por ultimo suma 3
```

En la siguiente tabla se ordenan de mayor a menor prioridad los operadores.

**Tabla 2-4. Precedencia y asociatividad de operadores**

Operadores	Asociatividad
() [] . ->	Izq a dcha ➔
- ~ ! ++ -- (cast) *(indireccion) &(dirección) sizeof	Dcha a izq ←
* / % (aritméticos)	Izq a dcha ➔
+ -	Izq a dcha ➔
<< >>	Izq a dcha ➔
< <= > >=	Izq a dcha ➔
== !=	Izq a dcha ➔
& (de bits)	Izq a dcha ➔
^	Izq a dcha ➔
	Izq a dcha ➔
&&	Izq a dcha ➔
	Izq a dcha ➔
?:	Izq a dcha ➔
= *= /= %= += -= <<= >>= &= ^=	Dcha a izq ←

La asociatividad establece como se agrupan operadores de la misma precedencia. La mayoría de los operadores se asocian de izquierda a derecha.

```
float a=3.0f, b=6.0f, c=5.0f, d;
d=a/b*3;           //asocitividad de izq a dcha, se empieza por la izq.
                    //primero a/b y luego multiplicado por 3, d=1.5
d=a/(b*3);         //d=0.16666667, por el parentesis
```

Algunos de ellos, principalmente las asignaciones, tienen asociatividad en el sentido contrario.

```
int a=3,b=5,c=7;
a=b=c=9;           //asociatividad de derecha a izq, se empieza por la dcha
                    //lo primero es c=9
                    //luego b=c, luego b=9
                    //por ultimo a=b, luego a=9

//el resultado final es que las tres variables acaban con valor 9
```

Ejemplo: Establecer el orden de evaluación de la siguiente expresión

```
r= x < 3.0f && 2 * x != 3 + y || n++;
```

solución:

1. n++
2. 2\*x
3. 3+y
4. x<3.0f
5. resultado paso 2 != resultado paso 3
6. resultado paso 4 && resultado paso 5
7. resultado paso 6 || resultado paso 1

### 2.5.10 Funciones matemáticas

Nótese bien que en los operadores de C no existe por ejemplo operaciones como la potenciación o la raíz cuadrada. Existe una librería matemática en C, con funciones que permiten hacer estas operaciones. Aunque las funciones se verán más adelante, se avanza aquí un pequeño ejemplo de cómo utilizarlas:

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

void main()
{
    float x=3.0f,r;
    r=sqrt(x); //r sera igual a la raiz cuadrada de x, o sea raiz de 3
}
```

Algunas otras funciones existentes en esta librería son las trigonométricas, directas e inversas, exponenciales, logaritmos, valores absolutos, redondeos, etc.

## 2.6 Entrada y salida por consola

Aunque la entrada y salida por consola tiene varias funciones, se resume a continuación el manejo básico de dos funciones, `printf()` y `scanf()`, para poder desarrollar algunos ejemplos sencillos con interacción con el usuario de los programas. Para poder utilizar estas funciones es necesario incluir el fichero `<stdio.h>`

La función `printf()` sirve para sacar texto por pantalla, tal y como se ha visto anteriormente, pero también puede servir para sacar por pantalla el contenido numérico de las variables, tal y como se muestra en el ejemplo siguiente:

```
#include <stdio.h> //este include es necesario

void main()
{
    int x=3;

    printf("El valor de x es %d \n",x);
}
```

El valor de x es 3

Nótese que en este caso, la función recibe 2 parámetros separados por una coma. El primer parámetro es la cadena de texto que va a sacar por pantalla, delimitada con comillas. En esa cadena de texto encontramos el símbolo % seguido de una letra, que indica el tipo de dato que va a sacar. así %d sirve para sacar números enteros y %f sirve para sacar números reales. El segundo parámetro es la variable que queremos sacar por pantalla.

En un mismo `printf()` se pueden sacar tantas variables por pantalla como se desee, por ejemplo:

```
#include <stdio.h>

void main()
{
    int a=3,b=5,c;
    c=a+b;
    printf("La suma de %d y %d es %d \n",a,b,c);
}
```

**La suma de 3 y 5 es 8**

Nótese que en este caso, en el primer parámetro que es la cadena de texto, aparecen tantos %d como variables queremos sacar, cada uno colocado en su sitio. A continuación, se ponen los nombres de las tres variables separados por comas. Nótese que el valor de la primera variable va al primer %d, el valor de la segunda va al segundo %d y así sucesivamente.

También se pueden poner expresiones matemáticas en el lugar correspondiente, si no queremos almacenar el valor intermedio en una variable auxiliar.

```
#include <stdio.h>

void main()
{
    float a=3.1f,b=5.1f;
    printf("La suma de %f y %f es %f \n",a,b,a+b);
}
```

**La suma de 3.100000 y 5.100000 es 8.200000**

Por ultimo, también podemos imprimir caracteres o letras, mediante %c. Este formato sacara por pantalla el caracter ASCII correspondiente al parámetro numérico.

```
#include <stdio.h>

void main()
{
    char letra=65; //totalmente equivalente a char letra='A';
    printf("Numero:%d Caracter:%c\n",letra,letra);
}
```

**Numero:65 Carácter:A**

La tarea opuesta es la entrada por consola: capturar lo que teclea el usuario y asignarlo a una variable. Esta labor se puede hacer con la función `scanf()`. La sintaxis es similar a la del `printf()`, excepto en los parámetros de las variables, que deben ir precedidos del símbolo &. Veamos un ejemplo que queremos preguntar la edad a una persona.

```
#include <stdio.h>

void main()
{
    int edad;
    printf("Introduzca su edad:"); //este es un mensaje sin variables
    scanf("%d",&edad); //se asigna lo que teclee el usuario a "edad"

    printf("El doble de su edad es: %d\n",edad*2); //tarea ejemplo
}
```

**Introduzca su edad: 10**  
**El doble de su edad es 20**

También se puede solicitar más de un dato a la vez. El siguiente ejemplo sirve para calcular el índice de masa corporal (IMC) de una persona.

```
#include <stdio.h>

void main()
{
    float peso, altura,imc;
    printf("Introduzca su peso en Kg y su altura en metros: ");
    scanf("%f %f",&peso, &altura); //se piden los dos datos

    imc=peso/(altura*altura);
    printf("Su IMC es: %f\n",imc);
}
```

**Introduzca su peso en Kg y su altura en metros: 76 1.82**  
**Su IMC es: 22.944088**

Por ultimo, también podemos pedir caracteres o letras, mediante %c.

```
#include <stdio.h>

void main()
{
    char character;
    int numero;
    printf("Introduzca un caracter: ");
    scanf("%c",&character);
    printf("Caracter:%c Numero:%d\n",character,character);

    printf("Introduzca un numero: ");
    scanf("%d",&numero);
    printf("Numero:%d Caracter:%c\n",numero,numero);
}
```

**Introduzca un caracter: a**  
**Caracter:a Numero:97**  
**Introduzca un numero: 98**  
**Numero:98 Caracter:b**

## 2.7 Estilo del código

Existen diferentes convenciones en cuanto al estilo del código, la forma en la que se utilizan las tabulaciones, los espacios, las líneas en blanco, las llaves, etc., siempre con el único objetivo de hacer el código más legible. Se recuerda además aquí en este punto la importancia de los comentarios: cualquier programa tiene que estar correctamente comentado para una fácil comprensión.

A continuación presentamos el estilo de código utilizado en este libro.

- Las llaves que abren y cierran un bloque de sentencias aparecen solas, cada una en un línea.
- El código dentro de un bloque de sentencias esta indentado una tabulación respecto a las llaves.
- Se deja una línea en blanco para separar partes de código dentro de cada bloque de sentencias.

```
#include <stdio.h>

//descripcion de mi programa
void main()
{
    //comentarios necesarios
    unsigned char c=3;
    int d=123;

    if(d<200)//otro comentario oportuno
    {
        float e=1.234f;
        int f=2;

        e=e*f+2.1f;
    }
}
```

Bloque de sentencias: 1 tabulacion

Linea en blanco después de las declaraciones

2 tabulaciones

## 2.8 Ejercicios resueltos

**Ej. 2.1) Escribir la salida por pantalla de los siguientes programas:**

<pre>#include &lt;stdio.h&gt; void main() {     int a=3,b=4,c;     c=a*4+b-2;     a=c*a;     b=c-a?a:c;     printf("%d %d %d\n",a,b,c); } //SOLUCION:</pre>	<pre>#include &lt;stdio.h&gt; void main() {     int a=3,b=4,c;     c=a&gt;b;     b=c*a+a;     a=a &amp;&amp; b;     printf("%d %d %d\n",a,b,c); } //SOLUCION:</pre>
<b>42 42 14</b>	<b>1 3 0</b>
<pre>#include &lt;stdio.h&gt; void main() {     float a=3.0f,b=4.0f;     int c,d;     c=a/b;     a=a/b+c;     d=a+b;     b=a*d;     printf("%f %f %d %d\n",a,b,c,d); } //SOLUCION:</pre>	<pre>#include &lt;stdio.h&gt; void main() {     float a=3.0f,b=4.0f;     int c,d;     c= (a&gt;b    a&gt;1);     c &amp;= 15;     d=c++;     a=b*c*d;      printf("%f %f %d %d",a,b,c,d); } //SOLUCION:</pre>
<b>0.750000 3.000000 0 4</b>	<b>8.000000 4.000000 2 1</b>

**Ej. 2.2) Desarrollar un programa para calcular la fuerza de atracción entre dos cuerpos, siendo sus masas y su distancia tecleados por el usuario. Las masas serán introducidas en toneladas, la distancia en cm., y el resultado se dará en Newtons**

La formula a aplicar es:

$$F = G \frac{M_1 M_2}{d^2}$$

Donde G es  $6,67 \times 10^{-11} \text{ Nm}^2/\text{kg}^2$ .

La salida por pantalla debe ser similar a la siguiente:

```
Introduzca masa cuerpo 1 (Ton):1
Introduzca masa cuerpo 2 (Ton):2
Introduzca distancia entre ellos (cm):2
Fuerza atraccion=0.333500 N
```

SOLUCION:

```
#include <stdio.h>
void main()
{
    const float G=6.67e-11f; //usamos una constante
    float masal,masa2,distancia,fuerza;
    printf("Introduzca masa cuerpo 1 (Ton):");
    scanf("%f",&masal);
    printf("Introduzca masa cuerpo 2 (Ton):");
    scanf("%f",&masa2);
    printf("Introduzca distancia entre ellos (cm):");
    scanf("%f",&distancia);

    masal*=1000;//en kg
    masa2*=1000;//en kg
    distancia/=100;//en metros
    fuerza=G*masal*masa2/(distancia*distancia);

    printf("Fuerza atraccion=%f N\n",fuerza);
}
```

**Ej. 2.3) Calcular el perímetro y área de un círculo cuyo radio se introduce por teclado.**

SOLUCION:

```
#include <stdio.h>
void main()
{
    const float pi=3.141592f;
    float radio,perimetro,area;
    printf("Introduzca radio:");
    scanf("%f",&radio);
    perimetro=2*pi*radio;
    area=pi*radio*radio;

    printf("Perimetro:%f Area:%f\n",perimetro,area);
}
```

## 2.9 Ejercicios propuestos

1. Escribe el siguiente mensaje en la pantalla: “Introducción al Lenguaje C”.
2. Multiplicar dos números enteros y mostrar el resultado en pantalla.
3. Multiplicar dos números reales ingresados por el teclado y mostrar el resultado en pantalla.
4. Realice un programa que imprima en pantalla las letras vocales y sus correspondientes caracteres ASCII.
5. Imprima en pantalla el tamaño en bytes que ocupa una variable tipo `int`, `char`, `double`, `float`, `long int`. (NOTA: utilizar el operador `sizeof()` )
6. Imprima diferentes formatos de valores reales y enteros.
7. Escriba un programa que permita averiguar qué acrónimo está representado por los códigos ASCII : 101 117 105 116 105
8. Calcule la superficie de una esfera.
9. Conversor de divisa: Se desea realizar un programa que permita convertir pesetas en euros y euros en dólares.
10. Realice un programa para obtener el resultado de las siguientes operaciones:

$\begin{aligned} &170 \& 155 \\ &235 \wedge 143 \\ &\sim 152 \end{aligned}$
---

11. Escriba un programa para detectar si un número es par o impar.
12. Escribir un programa para que el usuario introduzca tres valores enteros por teclado. Con dichos valores, realice las operaciones de suma, resta, multiplicación y cociente. Imprima por pantalla los resultados
13. Escribir un programa que pida al usuario las coordenadas de 2 puntos bidimensionales en coordenadas cartesianas e imprimir por pantalla la distancia entre ambos puntos.





# 3. Sentencias de control

## 3.1 Introducción

En los programas vistos en el capítulo anterior, la ejecución de sentencias es totalmente secuencial, se van ejecutando unas detrás de otras, hasta llegar al final de la función `main()`. No obstante, esto no es suficiente para crear programas de verdad, en los que hay que hacer tareas repetitivas o tomar decisiones, tareas que necesitan las denominadas sentencias de control, que son palabras claves de C que pueden alterar el ciclo de ejecución secuencial del programa. Estas sentencias de control se pueden agrupar en dos tipos:

- Condicionales. Las sentencias condicionales `if-else` (si) y `switch-case` (en caso de) sirven para tomar decisiones en función de unas determinadas condiciones, para ejecutar unas sentencias u otras.
- Bucles. Estas sentencias de control sirven para realizar tareas repetitivas, utilizando las instrucciones `for`, `while` y `do-while`.
- Saltos o rupturas del flujo de ejecución: `break`, `continue` y `goto`

## 3.2 Condicionales

La toma de decisiones condicionales se realiza fundamentalmente con la instrucción `if`, aunque en ciertos casos puede ser conveniente utilizar una estructura `switch-case`.

### 3.2.1 Instrucción if-else

Es una de las sentencias más usadas en cualquier lenguaje de programación y se emplea para tomar decisiones. Permite ejecutar un bloque u otro de sentencias dependiendo de una o varias condiciones, que deben escribirse entre paréntesis. Las condiciones suelen ser comparaciones entre variables, constantes y resultados de operaciones.

La sentencia `if` funciona como se muestra en la figura siguiente. Se ejecutara el bloque de sentencias A si se cumple la condición (si es distinta de cero). Nótese que el bloque de sentencias B se cumple en todo caso, ya que queda fuera de la condición.

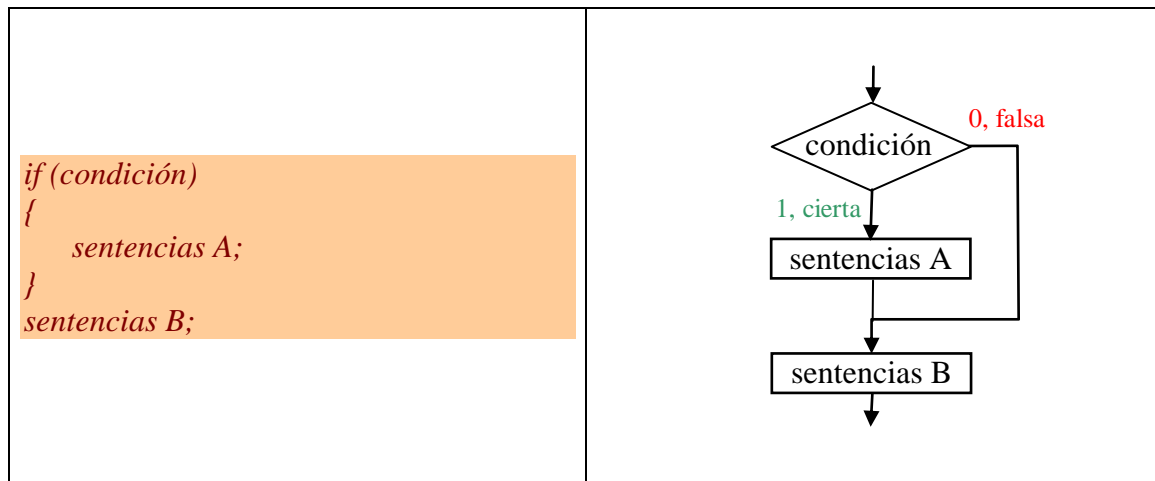


Figura 3-1. Estructura básica de un if

Veamos un ejemplo sencillo:

```
#include <stdio.h>

void main()
{
    int edad;
    printf("Introduzca su edad: ");
    scanf("%d",&edad);
    if(edad>=18)
    {
        printf("Es usted mayor de edad\n");
    }
    printf("Final del programa\n");
}
```

**Introduzca su edad: 23**  
**Es usted mayor de edad.**  
**Final del programa**

**Introduzca su edad: 12**  
**Final del programa**

Para ejecutar un bloque de sentencias en caso de que se cumpla la condición, o ejecutar otro bloque de sentencias en caso negativo, existe la instrucción `else` (sino) para formar la estructura `if-else` que funciona como sigue:

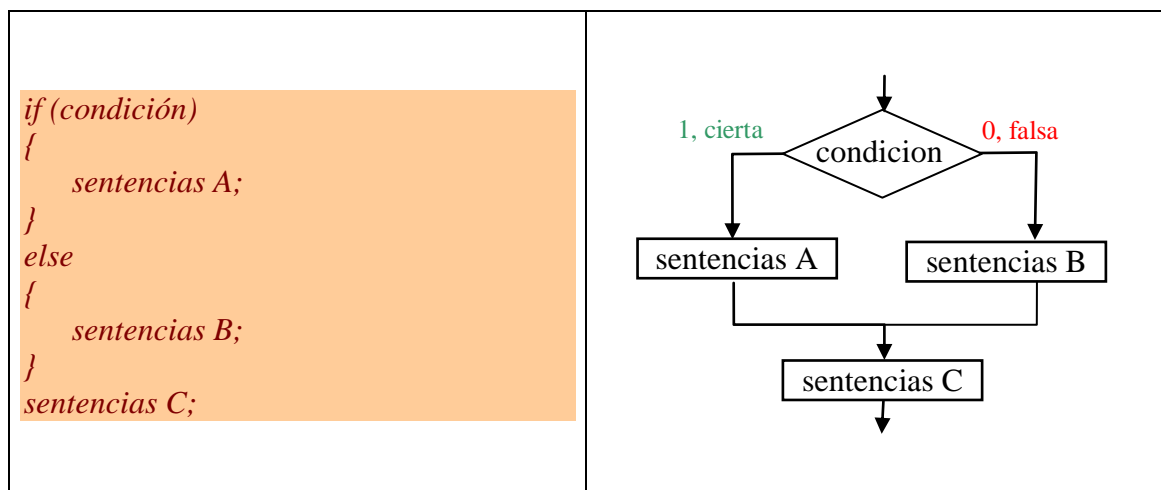


Figura 3-2. Estructura básica de un if-else

En el caso del ejemplo anterior, podríamos tener:

```
#include <stdio.h>

void main()
{
    int edad;
    printf("Introduzca su edad: ");
    scanf("%d",&edad);

    if(edad>=18)
    {
        printf("Es usted mayor de edad\n");
    }
    else //es decir edad<18
    {
        int dif;
        dif=18-edad;
        printf("Le faltan %d años para ser adulto\n",dif);
    }
}
```

**Introduzca su edad: 23**  
**Es usted mayor de edad.**

**Introduzca su edad: 12**  
**Le faltan 6 años para ser adulto**

Cuando la expresión entre paréntesis o condición se evalúa y es distinto de cero, se ejecuta el primer bloque de sentencias. Esa evaluación puede ser aplicada directamente a una variable, es decir es totalmente equivalente

***if(variable!=0) ⇔ if(variable)***

Veámoslo con un ejemplo.

```
#include <stdio.h>

void main()
{
    int numero,impar;
    printf("Introduzca un numero: ");
    scanf("%d",&numero);
    impar=numero%2;

    if(impar)
    {
        printf("%d es impar\n",numero);
    }
    else
    {
        printf("%d es par\n",numero);
    }
}
```

**Introduzca un numero: 25**  
**25 es impar**

**Introduzca un numero: 38**  
**38 es par**

Cuando el bloque de sentencias consta de una única sentencia, entonces las llaves son opcionales. Si además tenemos en cuenta que los separadores no son tenidos en cuenta, podríamos escribir el código anterior de las siguientes formas, todas ellas correctas y totalmente equivalentes, aunque las recomendables (por estilo) son las dos superiores:

```
if(impar)
{
    printf("%d es impar\n",numero);
}
else
{
    printf("%d es par\n",numero);
}
```

```
if(impar)
    printf("%d es impar\n",numero);
else
    printf("%d es par\n",numero);
```

```
if(impar) printf("%d es impar\n",numero);
else printf("%d es par\n",numero);
```

```
if(impar)
printf("%d es impar\n",numero);
else
printf("%d es par\n",numero);
```

### 3.2.1.1 If-else anidados

Los if-else pueden encadenarse o anidarse como se desee, tal y como se muestra en el ejemplo siguiente:

```
#include <stdio.h>
void main()
{
    int edad;
    printf("Introduzca su edad: ");
    scanf("%d",&edad);
    if(edad>=18)
    {
        if(edad<=65)
            printf("Esta usted en edad laboral\n");
        else
            printf("No esta usted en edad laboral\n");
    }
    else
    {
        printf("No esta usted en edad laboral\n");
    }
}
```

No obstante, en muchos casos se puede construir el programa con el mismo funcionamiento utilizando expresiones booleanas compuestas. En el ejemplo anterior tendríamos:

```
#include <stdio.h>
void main()
{
    int edad;
    printf("Introduzca su edad: ");
    scanf("%d",&edad);
    if(edad>=18 && edad<=65)
        printf("Esta usted en edad laboral\n");
    else
        printf("No esta usted en edad laboral\n");
}
```

Cuando se concatenan `if-else` hay que tener en cuenta el orden de encadenamiento. Si se utilizan llaves, el encadenamiento queda definido explícitamente.

Cuando no se utilizan llaves la regla aplicable es que cada `else` empareja con el `if` más cercano. Es importante tener presente que la tabulación a la hora de escribir un programa facilita la comprensión del mismo, pero no obliga a que el compilador empareje un `else` con el `if` que aparenta el formato.

Como ejemplo, véase la salida por pantalla de los siguientes fragmentos de código:

<pre>int x=3,y=5; if (x&gt;15) if (x&gt;y) x=8; else x=14; printf("%d",x);</pre>	<pre>int x=30,y=5; if (x&gt;15) if (x&gt;y) x=8; else x=14; printf("%d",x);</pre>	<pre>int x=30,y=50; if (x&gt;15) if (x&gt;y) x=8; else x=14; printf("%d",x);</pre>	<pre>int x=30,y=50; if (x&gt;15) { if (x&gt;y) x=8; } else x=14; printf("%d",x);</pre>
<b>3</b>	<b>8</b>	<b>14</b>	<b>3</b>

En el siguiente ejemplo se realiza un programa en C que permita comprobar si un año (dado por teclado) es bisiesto o no. Recuerdese que son bisiestos los años divisibles por 4, excepto los que son divisibles por 100 pero no por 400.

```
#include<stdio.h>
void main ()
{
    int a;
    printf ("Diga el año a considerar\n");
    scanf("%d",&a);
    if((a%4==0)&&((a%100!=0)|| (a%400==0)))
        printf ("El año %d es bisiesto\n",a);
    else
        printf("El año %d no es bisiesto\n",a);
}
```

### 3.2.1.2 Un error típico con if

Es un error muy común poner un solo signo `=` en lugar del doble signo `==` que requiere el operador condicional. Se produce entonces una curiosa acción. El signo `=` es de asignación, con lo que en lugar de preguntar se asigna el valor por el que se pregunta. Si el valor asignado es distinto de 0 se da por cierta la pregunta, si es cero se da por falsa.

**Ejemplo: Detectar si un número tecleado es cero o no.**

```
// PRUEBA DE MALA ASIGNACION EN UN if
#include <stdio.h>
void main()
{
    int n;
    printf("Escriba un número: ");
    scanf("%d",&n);
    if(n=0) //error logico, no de sintaxis
        printf("El número es cero");
    else
        printf("El número no es cero");
}
```

**Escriba un número: 0**  
**El numero no es cero**

### 3.2.1.3 If-else if

Otra forma de presentarse los if es mediante el else if. Es la forma general de presentarse una decisión de muchas alternativas. Las expresiones se evalúan en orden y en el momento que se cumpla una de ellas, se termina la cadena de decisiones en su totalidad. Su forma es:

```
if (condición1)
    instruccionesA
else if (condición2)
    instruccionesB
else if (condición3)
    instruccionesC
else
    instruccionesD
```

El ejemplo anterior de determinar si un año es bisiesto o no lo podíamos haber codificado de la siguiente forma, con una estructura de la forma `if-else if`

```
#include <stdio.h>
main()
{
    int a;
    printf ("Escriba el año: ");
    scanf ("%d",&a);
    if (a%4!=0)
        printf ("El año %d no es bisiesto",a);
    else if (a%400==0)
        printf ("El año %d es bisiesto",a);
    else if (a%100==0)
        printf ("El año %d no es bisiesto",a);
    else
        printf ("El año %d es bisiesto",a);
}
```

Un uso típico de la estructura `if-else if` es la clasificación en intervalos, tal como se muestra en este ejemplo en el que se pretende dada una nota numérica entera, escribir la calificación textual (aprobado, notable, etc.) que le corresponde. Nótese la forma de resolverlo mediante `if` independientes y mediante la estructura `if-else if`. Este último caso suele ser mucho más fácil y robusto frente a errores u omisiones en el código.

<pre>#include &lt;stdio.h&gt; //mejor solucion main() {     float nota;     printf ("Diga una nota: ");     scanf ("%f",&amp;nota);     if (nota&lt;5)         printf("Suspenso\n");     else if (nota&lt;7)         printf("Aprobado\n");     else if (nota&lt;9)         printf("Notable\n");     else if (nota&lt;10)         printf("Sobresaliente\n");     else         printf("Matricula\n"); }</pre>	<pre>#include &lt;stdio.h&gt; //peor solucion main() {     float nota;     printf ("Diga una nota: ");     scanf ("%f",&amp;nota);     if (nota&lt;5)         printf("Suspenso\n");     if (nota&gt;=5 &amp;&amp; nota&lt;7)         printf("Aprobado\n");     if (nota&gt;=7 &amp;&amp; nota&lt;9)         printf("Notable\n");     if (nota&gt;=9 &amp;&amp; nota&lt;10)         printf("Sobresaliente\n");     if (nota==10)         printf("Matricula\n"); }</pre>
---	--

### 3.2.1.4 El operador condicional ? :

Otra forma de escribir los `if-else` es con el operador condicional `?`: visto en el capítulo anterior, con lo que se consiguen expresiones de aspecto muy compacto, tal como se muestra en el código siguiente:

<pre>//con un if #include&lt;stdio.h&gt; main() {     int mayor,a,b;     printf("Introduzca a y b: ");     scanf("%d %d",&amp;a,&amp;b);     if (a&gt;b)         mayor=a;     else         mayor=b;     printf("Mayor: %d\n",mayor); }</pre>	<pre>//con el operador condicional #include&lt;stdio.h&gt; main() {     int mayor,a,b;     printf("Introduzca a y b: ");     scanf("%d %d",&amp;a,&amp;b);     a&gt;b ? mayor=a:mayor=b;     //tambien    mayor=(a&gt;b ? a:b);     printf("Mayor: %d\n",mayor); }</pre>
--	--

### 3.2.2 Instrucción switch-case

Su funcionamiento es parecido al del varios `if` encadenados, pero de más fácil comprensión y escritura. Permite la evaluación de una condición múltiple (bifurcación multidireccional). La estructura de un `switch - case` es:

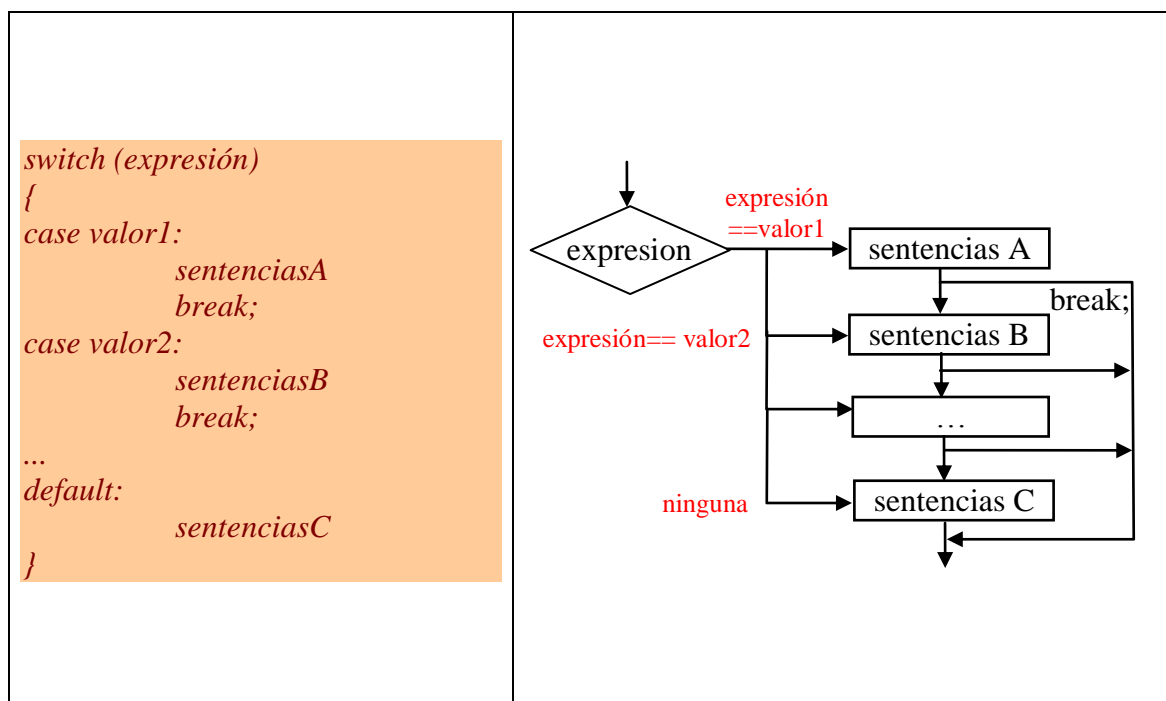


Figura 3-3. Estructura básica de la instrucción switch-case

La expresión debe ser (o tener como resultado) una variable entera (`char`, `int`, etc.), cuya igualdad será comprobada con los distintos valores (`valor1`, `valor2`, etc.) proporcionados en las distintas cláusulas o casos `case`, y cuyo tipo tiene que ser igualmente entero. No pueden existir dos `case` con el mismo valor. Si la expresión es igual a uno de los valores, comenzará a ejecutar el correspondiente bloque de sentencias.

Cada grupo queda limitado por la palabra clave `case` y, normalmente, por la palabra clave `break`, no precisando llaves, aunque estas pueden ser utilizadas si es conveniente. Con `break` se sale inmediatamente del `switch` (también vale para otros casos, como `do`, `while`, `for`, etc.). Si no hay `break` se sigue ejecutando secuencialmente el `switch` hasta detectar otro `break` o finalizar el `switch`.

A veces es difícil poder saber si se tienen controlados todos los posibles valores que puede portar la expresión. En este caso se deja una salida de emergencia para valores no controlados mediante la palabra `default`. Puede haber `switch` anidados, es decir, dentro de un `case` puede haber otro `switch`.

Una utilidad típica es la construcción de menús:

```
#include <stdio.h>

main()
{
    int opcion;
    printf ("Introduzca opcion:\n");
    printf("1-Imprimir\n");
    printf("2-Copiar\n");
    printf("3-Guardar\n");
    scanf ("%d",&opcion);

    switch (opcion)
    {
        case 1:printf ("Imprimiendo...");
               break;
        case 2:printf ("Copiando...");
               break;
        case 3:printf ("Guardando...");
               break;
        default:
            printf ("Opcion incorrecta\n");
    }
}
```

## 3.3 Bucles

### 3.3.1 Generalidades

Un bucle es una o varias sentencias que se repiten un número variable de veces dependiendo de una o varias condiciones. Sirven para realizar tareas repetitivas o iterativas de forma compacta.

De la estructura de un bucle se sale cuando se cumple una determinada condición, pero también se puede salir en cualquier momento (sin haber completado el bucle) mediante la orden `break`. El programa seguiría en la sentencia siguiente al bucle.

### 3.3.2 Bucle for

La estructura del bucle `for` es la siguiente:



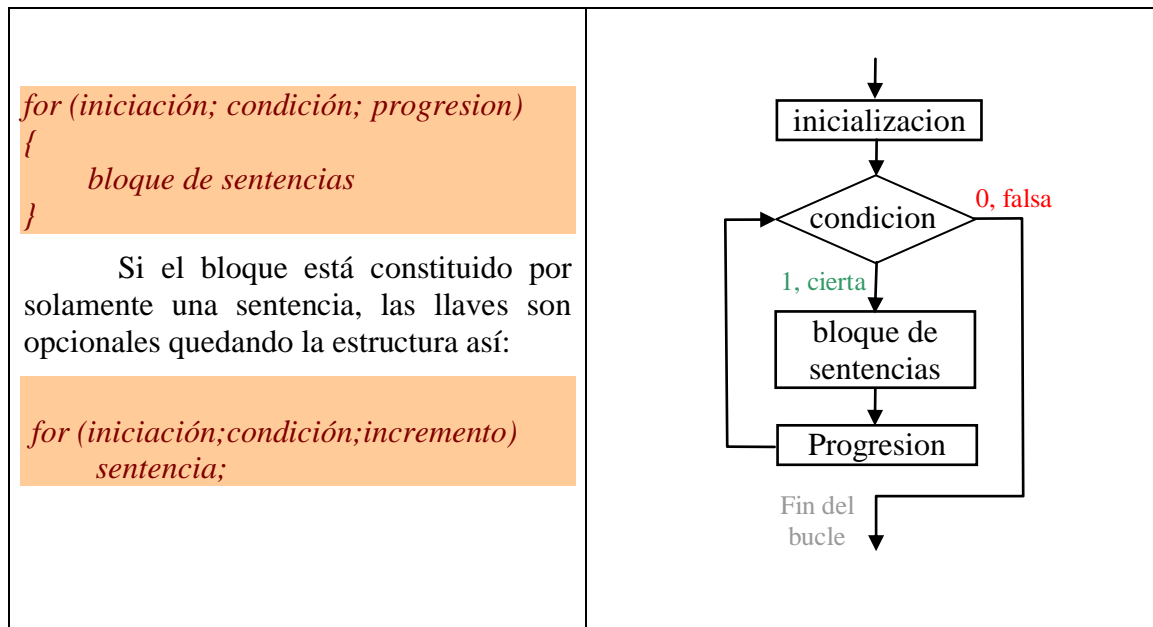


Figura 3-4. Estructura del bucle for

El `for` lleva asociada una variable entera, real o de caracteres, que controla el bucle mediante los tres parámetros indicados en el paréntesis y que se escriben separados entre sí por punto y coma. Iniciación es el valor inicial de la variable asociada. A continuación se establece la condición (mientras se cumpla se sigue ejecutando el bucle). Por último se indica el incremento de la variable en cada paso del bucle.

En el siguiente ejemplo se pretende imprimir en consola los números desde el 0 hasta el 9, usando un bucle `for`.

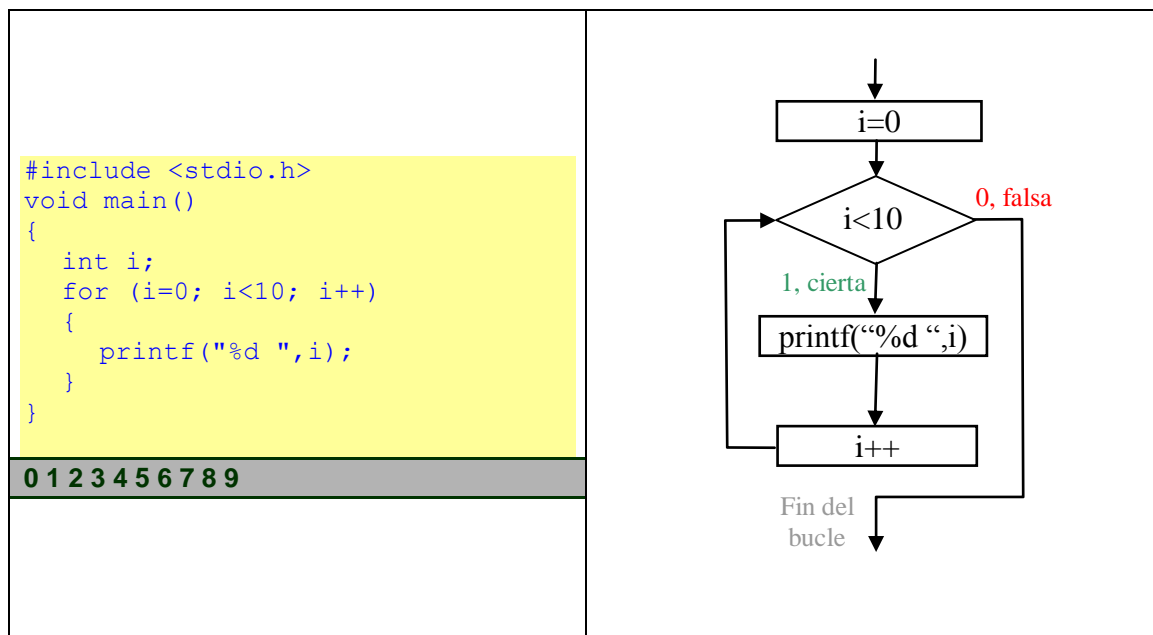


Figura 3-5. Ejemplo de funcionamiento del bucle for

**NOTA DE ESTILO:** En los bucles, las variables asociadas al bucle se suelen denominar como subíndices matemáticos, es decir, con las letras `i`, `j`, `k`.

**Ejemplo:** Escribir una lista de los cuadrados de los 10 primeros números naturales.

```
#include<stdio.h>
main()
{
    int i,c;
    for(i=1;i<=10;i++)
    {
        c=i*i;
        printf ("%d al cuadrado= %d",i,c);
    }
}
```

**Ejemplo:** Calcular el factorial del número ‘n’ introducido por el usuario

```
#include<stdio.h>
main()
{
    int n,i,fact=1;
    printf("Introduzca n: ");
    scanf("%d",&n);

    for(i=1;i<=n;i++)
        fact*=i;

    printf("Factorial de %d vale %d\n",n,fact);
}
```

Nótese que en la inicialización, condición y progresión puede existir cualquier expresión, ya sea una expresión vacía o una expresión compleja. En el caso anterior, para calcular el factorial podríamos haber escrito, con una inicialización múltiple:

```
#include<stdio.h>
main()
{
    int n,i,fact;
    printf("Introduzca n: ");
    scanf("%d",&n);

    for(i=1,fact=1;i<=n;i++)
        fact*=i;

    printf("Factorial de %d vale %d\n",n,fact);
}
```

También pueden realizarse iteraciones con un bucle `for` manejando caracteres de texto. El siguiente ejemplo muestra el alfabeto por pantalla:

```
#include<stdio.h>
main()
{
    int i;
    for (i='a';i<='z';i++)
    {
        printf("%c ",i);
    }
}
```

**abcdefghijklmnopqrstuvwxyz**

### 3.3.3 Bucle while

Se ejecutan unas sentencias solamente mientras se cumpla una condición establecida, escrita entre paréntesis. Las sentencias del bucle van siempre entre llaves a menos que sea una sola, en cuyo caso no son precisas. Su estructura es:

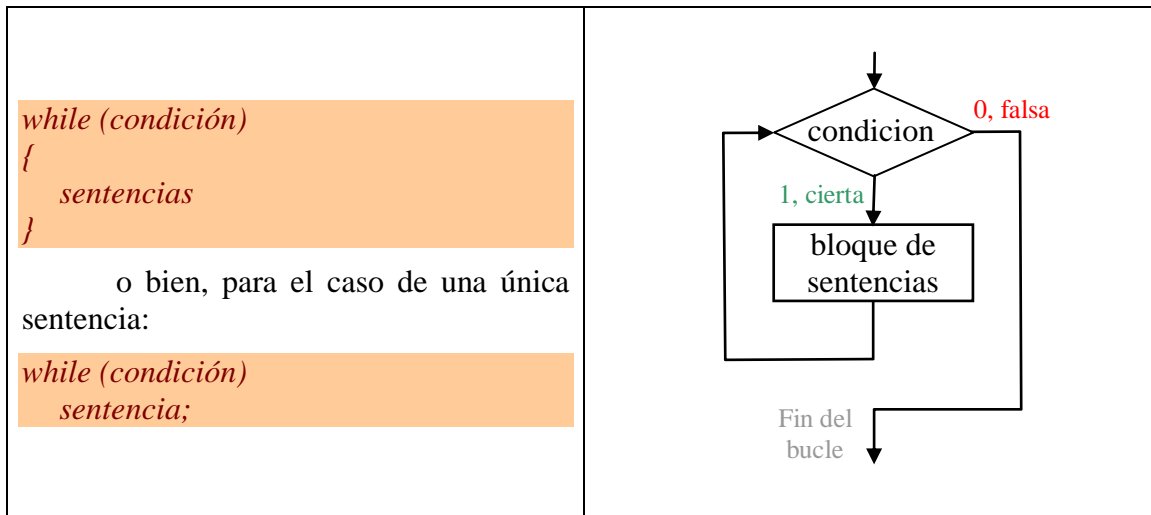


Figura 3-6. Estructura del bucle while

Para realizar la cuenta de 0 hasta 9, se implementaría con un `while` de la siguiente forma:

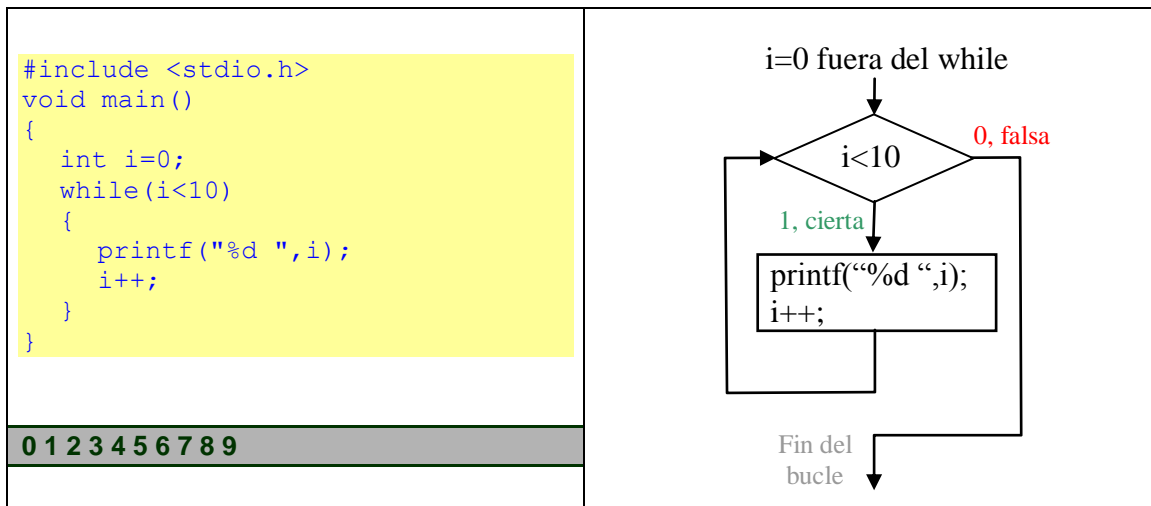


Figura 3-7. Ejemplo funcionamiento del bucle while

**Ejemplo:** Escribir el cuadrado de los 20 primeros números naturales ( `while` )

```
#include<stdio.h>
main()
{
    int x,c;
    x=1;
    while (x<=20)
    {
        c=x*x;
        printf("%d al cuadrado es %d\n",x,c);
        x++;
    }
}
```

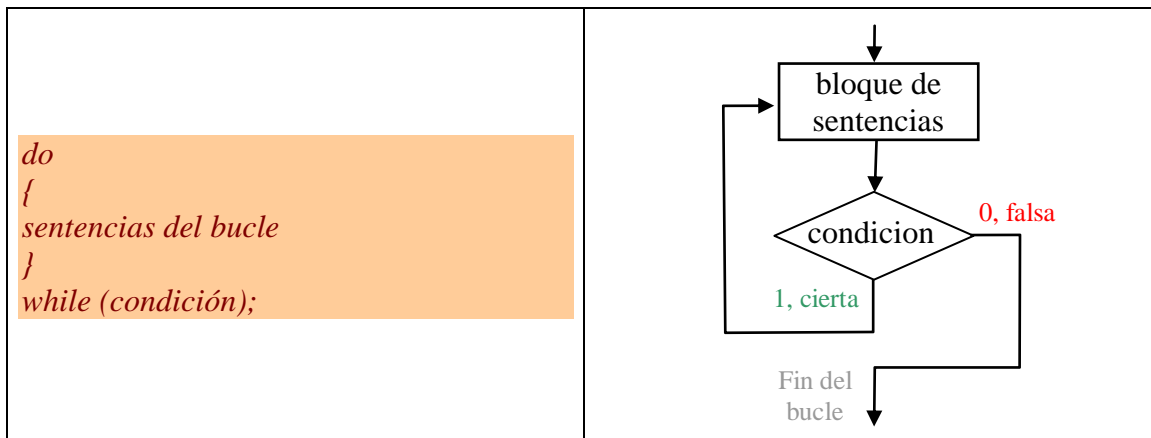
La condición establecida debe ser tal que por alguna circunstancia se produzca cambio en alguna de las variables que intervienen en la condición, si no el bucle sería infinito. Es específico del `while` que, antes de ejecutarse las sentencias que lo integran, se evalúa la condición, con lo que puede que no se ejecute su contenido.

**Ejemplo:** Calcular el factorial del número  $n$  (con `while`).

```
#include<stdio.h>
main()
{
    int n,x=1,fact=1;
    printf("Introduzca n: ");
    scanf("%d",&n);
    while (x<=n)
    {
        fact*=x;
        ++x;
    }
    printf("El factorial de %d vale %d\n",n,fact);
}
```

### 3.3.4 Bucle do - while

Es menos usada que las dos anteriores, pero muy útil, ya que, a diferencia del `while`, las sentencias del cuerpo del `do - while` al menos se ejecutan una vez, pues la primera vez aún no se ha evaluado la condición. Su estructura es:



**Figura 3-8. Estructura del bucle do-while**

La sentencia `do` no lleva punto y coma, en cambio la sentencia `while` si lo lleva. Si el cuerpo del `do - while` tiene varias sentencias, éstas irán entre llaves.

**Ejemplo:** Escribir el cuadrado de los 20 primeros números naturales.

```
#include<stdio.h>
main()
{
    int x=1,c;
    do
    {
        c=x*x;
        printf ("%d al cuadrado= %d\n",x,c);
        x=x+1;
    }
    while (x<=20);
}
```

**Ejemplo: Calcular el factorial del número n (con do-while).**

```
#include<stdio.h>

main()
{
    int n,x=1,fact=1;
    printf("Introduzca n: ");
    scanf("%d",&n);

    do
    {
        fact*=x;
        x++;
    }
    while (x<=n);

    printf("Factorial de %d vale %d\n",n,fact);
}
```

**Ejemplo: Dado un número positivo, escribirlo al revés**

```
#include<stdio.h>

main()
{
    int num,digito;
    printf("Escriba un número positivo: ");
    scanf("%d",&num);
    printf("El número invertido es: ");

    do
    {
        digito=num%10;
        printf("%d",digito);
        num=num/10;
    }
    while(num>0);

    printf("\n");
}
```

**3.3.5 ¿Que bucle elegir?**

Cualquier tarea repetitiva que deba ser implementada con un bucle puede serlo utilizando cualquiera de ellos: `for`, `while` o `do-while`. Cualquier bucle escrito con uno de ellos, puede ser reescrito de forma totalmente equivalente con cualquiera de los otros. Como muestra se representa aquí el código que cuenta desde 0 hasta n, siendo n un número introducido por el usuario:

```
#include <stdio.h>

void main()
{
    int i,n;
    printf("Introduzca n:");
    scanf("%d",&n);
    for (i=0; i<=n; i++)
    {
        printf("%d ",i);
    }
}
```

```
#include <stdio.h>

void main()
{
    int i,n;
    printf("Introduzca n:");
    scanf("%d",&n);

    i=0;
    while(i<=n)
    {
        printf("%d ",i);
        i++;
    }
}
```

```
#include <stdio.h>

void main()
{
    int i,n;
    printf("Introduzca n:");
    scanf("%d",&n);

    i=0;
    do
    {
        printf("%d ",i);
        i++;
    }
    while(i<=n);
}
```

Como reglas generales se pueden usar las siguientes:

1. Si la tarea repetitiva se ejecutara un número de veces conocido a priori con bastante seguridad, es típico utilizar una estructura `for`. En esos casos se utiliza el `break` para gestionar salidas excepcionales del bucle.
2. Si la tarea se repetirá un número de veces indeterminado o desconocido a priori, entonces la estructura típica es `while` o `do-while`
  - a. Si la tarea repetitiva tiene que ser realizada al menos 1 vez, entonces se usara la estructura `do-while`
  - b. Si la tarea repetitiva no es necesario que se realice al menos 1 vez, entonces se usa la estructura `while`

Ejemplos: Una tabla de multiplicar, de sumar, el factorial de un número, son tareas que sabemos el número de veces que se tienen que hacer y usaremos el `for`. Pedirle un número al usuario tantas veces como sea necesario para garantizar que ese número esta en un rango (ej. Pedir un número del 1 al 10). El usuario puede introducir un número negativo las veces que sea, así que usaremos un `while`.

### 3.3.6 Bucles anidados

Por supuesto, los bucles se pueden anidar unos dentro de otros para producir tareas múltiplemente repetitivas. Por ejemplo, un bucle `while` puede contener un bucle `for`, que a su vez tiene dentro un bucle `do-while`.

Por ejemplo, si se desean sacar por pantalla las tablas de multiplicar de todos los números del 1 al 9, la forma de hacerlo sería la siguiente:

<pre>#include&lt;stdio.h&gt;  main() {     int i,j;      for (i=1; i&lt;10; i++)     {         printf("Tabla del %d\n",i);         for (j=1; j&lt;10; j++)         {             printf("%d x %d = %d\n",i,j,i*j);         }     } }</pre>	<div> <b>Tabla del 1</b>  1 x 1 = 1  1 x 2 = 2  1 x 3 = 3  .....  <b>Tabla del 2</b>  .....  <b>Tabla del 9</b>  .....  9 x 6 = 54  9 x 7 = 63  9 x 8 = 72  9 x 9 = 81 </div>
--	---

**Ejemplo:** Realizar un programa que calcule el factorial de un número n, introducido por el usuario, tantas veces como se desee. La forma de acabar el programa es pulsando Ctrl+C

```
#include<stdio.h>

main()
{
    int n,i,fact;
    while(1)
    {
        printf("Introduzca n: ");
        scanf("%d",&n);

        fact=1;//Notese la inicializacion aqui.
        for(i=1;i<=n;i++)
            fact*=i;

        printf("Factorial de %d vale %d\n",n,fact);
    }
}
```

### 3.3.7 Algorítmica vs. Matemáticas

Supóngase que se desea calcular la suma de los n primeros números naturales, siendo n tecleado por el usuario. Con un bucle podríamos codificar la solución que sigue:

```
#include<stdio.h>

main()
{
    int i,suma=0,n;
    printf("Introduzca n:");
    scanf("%d",&n);

    for(i=1;i<=n;i++)
        suma+=i;

    printf("Suma= %d\n",suma);
}
```

No obstante, la solución de este problema tiene una forma cerrada, es decir, no algorítmica. Conociendo la fórmula de una progresión aritmética, se deduce fácilmente que dicha suma vale  $n(n+1)/2$ , lo que implicaría que podríamos hacer el programa de la siguiente forma:

```
#include<stdio.h>

main()
{
    int suma=0,n;
    printf("Introduzca n:");
    scanf("%d",&n);


    suma=n*(n+1)/2;

    printf("Suma= %d\n",suma);
}
```

No hay ninguna duda de que esta última solución es mucho mejor que la anterior.

### 3.3.8 Un error típico con los bucles

La sentencia `for` o `while` no llevan punto y coma final. Si lo llevan entonces realmente lo que tiene después es una sentencia vacía. Es decir, los dos fragmentos de código siguientes son totalmente equivalentes, y obtienen la siguiente salida por pantalla:

<pre>#include &lt;stdio.h&gt;  void main() {     int i;     for (i=0; i&lt;10; i++); //Ojo     printf("I vale=%d ",i); }</pre>		<pre>#include &lt;stdio.h&gt;  void main() {     int i;     for (i=0; i&lt;10; i++)     {         ; //sentencia vacia     }     printf("I vale=%d ",i); }</pre>
<b>I vale=10</b>		

## 3.4 Alteraciones del flujo de ejecución

Las sentencias `break`, `continue` y `goto` permiten alterar el flujo normal de ejecución de una estructura de control, ya sea un `switch-case` o uno de los bucles vistos anteriormente. Adelantamos ya que las dos primeras son utilizadas y validas, sin embargo, la instrucción `goto` NO debe ser utilizada en ningún caso.

El uso de `break` en un `switch-case` ya se vio en su momento. El `switch-case` no permite el uso del `continue`.

Cuando `break` y `continue` aplican a los bucles, permiten abandonarlo en cualquier momento o saltar a la siguiente iteración sin terminar el bloque de sentencias completo. Están ligados normalmente a condiciones, es decir, aparecen siempre condicionados mediante un `if`.



### 3.4.1 Sentencia break

La sentencia `break` interrumpe un bucle en el momento en el que se ejecuta y continua el programa por la primera línea fuera del bucle. En el caso siguiente, solo se ejecutarían las sentencias A.

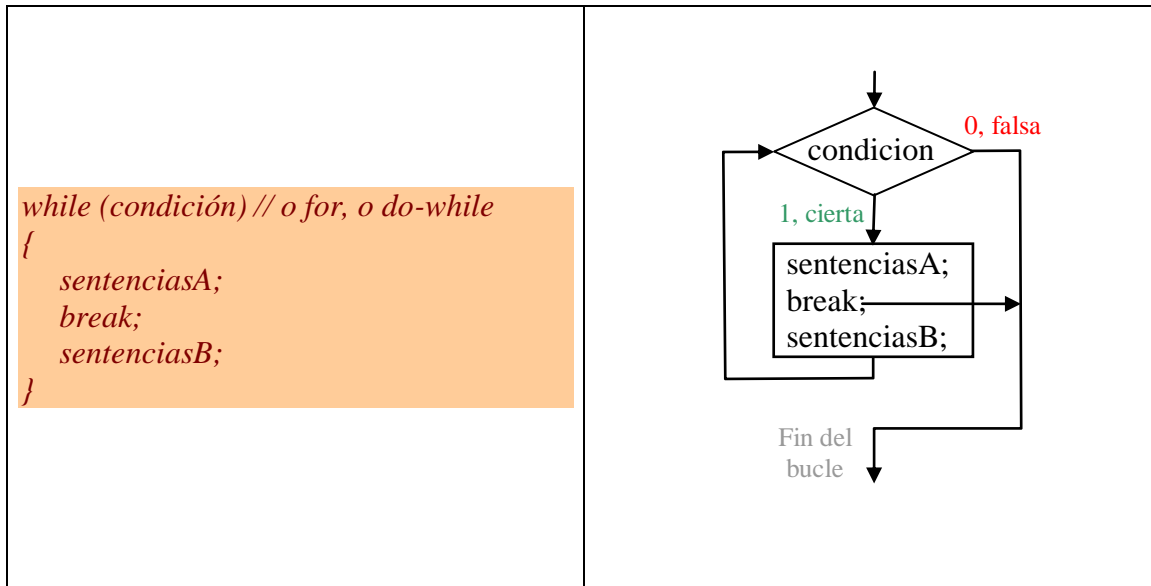


Figura 3-9. Uso del break en un bucle while

En el ejemplo anterior de contar hasta 9, podríamos tener (aunque se prefiere la solución original):

```
#include <stdio.h>
void main()
{
    int i=0;
    while(1)
    {
        if(i==10)
            break;
        printf("%d ",i);
        i++;
    }
}
```

0 1 2 3 4 5 6 7 8 9

Es importante resaltar que en el caso de bucles anidados, el `break` solo termina el bucle en el que se encuentra, el más interno.

### 3.4.2 Sentencia continue

La sentencia `continue` se utiliza para, sin abandonar un bucle, no se sigan ejecutando todas las sentencias de una determinada iteración. Su uso es aun más excepcional que el del `break`. El programa seguirá ejecutando el bucle, es solo una determinada iteración la que no se completa.

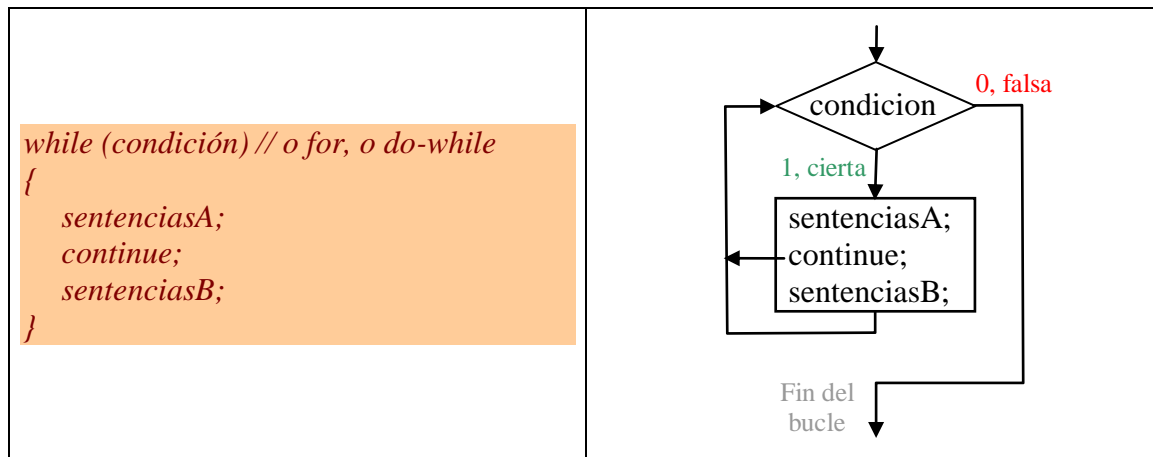


Figura 3-10. Uso del continue en un bucle while

El ejemplo siguiente serviría para mostrar los números impares menores que 20.

```
#include <stdio.h>

void main()
{
    int i;
    for (i=1; i<20; i++)
    {
        if (i%2==0)
            continue;
        printf("%d ", i);
    }
}
```

Es un ejemplo didáctico, si realmente se quisiera implementar ese programa, se haría más fácilmente con:

```
#include <stdio.h>

void main()
{
    int i;
    for (i=1; i<20; i+=2)
        printf("%d ", i);
}
```

**Ejemplo:** De un lote de 10 números enteros que se piden al usuario, contar cuantos números positivos introduce.

```
#include <stdio.h>

main()
{
    int num, i, num_positivos=0;
    for (i=0; i<10; i++)
    {
        scanf ("%d", &num);
        if (num<=0)
            continue;
        num_positivos++;
    }
    printf("Total:%d positivos\n", num_positivos);
}
```

Al igual que anteriormente, este es un ejemplo didáctico. La solución sin `continue` sería:

```
#include <stdio.h>

main()
{
    int num,i,num_positivos=0;

    for (i=0;i<10;i++)
    {
        scanf ("%d",&num);
        if (num>0)
            num_positivos++;
    }

    printf("Total:%d positivos\n",num_positivos);
}
```

### 3.4.3 Instrucción goto

En programación estructurada **NO DEBE UTILIZARSE GOTO**

El `goto` es una bifurcación incondicional que dirige la ejecución del programa desde el punto de detección hasta una etiqueta asociada sin necesidad de que se cumpla ninguna condición. Se entiende por etiqueta una referencia alfanumérica seguida de dos puntos (:). Su presentación es:

```
goto etiqueta;
```

donde “etiqueta” es un nombre de referencia que debe figurar en alguna parte del programa en la forma:

```
etiqueta:
```

Una de las utilidades del `goto` es abandonar estructuras de programación profundamente anidadas, rompiendo varios bucles a la vez.

**Ejemplo:** Escribir un programa que permita pasar de grados sexagesimales a radianes, el final se sobreentiende dando cero grados.

```
// PASO DE GRADOS SEXAGESIMALES A RADIANES

#include<stdio.h>

main()
{
    float g,r;

otro:
    scanf("%f",&g);
    if(g==0)
        goto fin;
    r=g*3.141592/180;
    printf("%f grados son %f radianes\n",g,r);
    goto otro;

fin:
    printf("Final del programa\n");
}
```

### 3.5 Ejercicios resueltos

**Ej. 3.1) Escribir la salida por pantalla de los siguientes programas:**

<pre>#include&lt;stdio.h&gt;  main() {     int i,j;     for (i=0,j=10;i&lt;=5;i++,j--)     {         printf("%d ",i*j);     } }</pre>	<pre>#include&lt;stdio.h&gt;  main() {     int i=1,s=0;     while(i)     {         i++;         if(i%2) s+=3;         else s+=5;         if(s&gt;20) break;         printf("%d ",s);     } }</pre>
<b>0 9 16 21 24 25</b>	<b>5 8 13 16</b>
<pre>#include&lt;stdio.h&gt;  main() {     int i=10,j=0;     do     {         i--;         j++;         printf("%d%d",i,j);         if(i*j&gt;10)         {             printf("Break");             break;         }     }while(i&gt;0 &amp;&amp; j&lt;10); }</pre>	<pre>#include&lt;stdio.h&gt;  main() {     int i=10,j=0;     for(i=0;i&lt;4;i++)     {         switch(i)         {             case 0:printf("a");             case 1:printf("b");             case 2:printf("c");             default:printf("E");         }     } }</pre>
<b>9182Break</b>	<b>abcEbcEcEE</b>

**Ej. 3.2) Calcular el máximo, mínimo y media de un conjunto de “n” valores reales introducidos por teclado, siendo “n” tecleado por el usuario**

**SOLUCION:**

```
// Cálculo del máximo, mínimo y media
#include<stdio.h>

main()
{
    int i,numero_datos;
    float media,min,max;

    printf("Numero de datos: ");
    scanf("%d",&numero_datos);

    media=0.0f;
    for (i = 1; i <= numero_datos; i++)
    {
        float dato;
        printf("Dato %d:",i);
        scanf("%f",&dato);
```

```

    media+=dato;
    if (max<dato || i==1) max=dato;
    if (min>dato || i==1) min=dato;
}
media=media/numero_datos;

printf("Min=%f Máx=%f Media=%f\n",min,max,media);
}

```

**Ej. 3.3) Calculadora:** Se desea programar una aplicación que funcione como calculadora capaz de realizar las cuatro operaciones básicas sobre números reales: suma, resta, multiplicación y división. El usuario introduce por orden el primer operando, el signo correspondiente a la operación y el segundo operando en una sola línea, sin espacios. El programa se ejecuta indefinidamente hasta que se pulsa Ctrl.+C. Los operandos tendrán precisión normal y se utilizara la sentencia switch-case. La salida por pantalla será similar a:

```

12+13.5
=25.500000
34.5/2
=17.250000
27-0.56
=26.440000
-3.2*-12
=38.400001

```

SOLUCION:

```

#include <stdio.h>

void main(void)
{
    float a,b; //los dos operandos
    unsigned char operador; //la operacion
    float resultado;

    while(1)
    {
        scanf("%f%c%f",&a,&operador,&b);

        switch(operador)
        {
            case '+': resultado=a+b;
                       break;
            case '-': resultado=a-b;
                       break;
            case '*': resultado=a*b;
                       break;
            case '/': resultado=a/b;
                       break;
            default: printf("Operador no conocido\n");
                     break;
        }
        printf("=%f\n",resultado);
    }
}

```

**Ej. 3.4) Adivinar un número:** Se desea programar un juego en el que el ordenador piensa un número entero del 0 al 99, y el usuario intenta adivinarlo. El ordenador informa al jugador si su estimación es menor, mayor o si ha acertado. Cuando acierta, le informa del número de intentos que le ha costado adivinar el número. Utilizar `srand(...)` para inicializar la semilla aleatoria y `rand()` para generar los números aleatorios. La semilla de generación de números aleatorios se inicializa solo una vez, al principio del programa. Se le pasa como parámetro el tiempo actual, para que cada vez que se juega sea una semilla diferente, tal como se muestra:

```
#include <stdlib.h> //para srand() y rand()
#include <time.h> //para time()
main()
{
    .....
    srand(time(NULL)); //Inicializar semilla aleatoria
    numero=rand()%100; //generar numero aleatorio
    .....
}
```

La salida por pantalla debe ser similar a:

```
Ya he pensado un numero del 0 al 99
Adivine. Diga un numero: 50
Se ha pasado
Adivine. Diga un numero: 25
Se ha pasado
Adivine. Diga un numero: 12
Se ha quedado corto
Adivine. Diga un numero: 20
Se ha quedado corto
Adivine. Diga un numero: 22
Se ha quedado corto
Adivine. Diga un numero: 23
Se ha quedado corto
Adivine. Diga un numero: 24
Lo adivino en 7 intentos!
```

**SOLUCION:**

```
#include <stdio.h>
#include <stdlib.h> //para srand() y rand()
#include <time.h> //para time()

void main(void)
{
    int numero,n;
    int contador_intentos=0;

    srand(time(NULL)); //Inicializar semilla aleatoria

    numero=rand()%100; //generar numero aleatorio
    printf("Ya he pensado un numero del 0 al 99\n");

    do
    {
        printf("Adivine.Diga un numero: ");
        scanf("%d",&n);

        contador_intentos++;
    }
```

```

    if(n<numero)
    {
        printf("Se ha quedado corto\n");
    }
    if(n>numero)
    {
        printf("Se ha pasado\n");
    }
}
while(numero!=n);

printf("Lo adivino en %d intentos!\n",contador_intentos);
}

```

**Ej. 3.5) Sumar 50 números dados por teclado. Si se detecta un número negativo, indicarlo y abandonar el bucle indicando cuantos valores se han leído hasta ese instante.**

**SOLUCION:**

```

#include<stdio.h>
main()
{
    int i,suma=0,num;

    for(i=1;i<=50;i++)
    {
        scanf("%d",&num);
        if (num<0)
        {
            printf("Se ha detectado un negativo\n");
            break;
        }
        suma+=num;
    }
    printf("Valores leidos %d. Total de la suma %d\n",i,suma);
}

```

**Ej. 3.6) Mostrar por pantalla y hallar la suma de los elementos de una progresión geométrica de razón  $r$ , entre los valores  $a$  y  $b$ , siendo estos valores tecleados por el usuario. Aunque existe solución cerrada a este problema, se pide hacerlo mediante un for.**

**SOLUCION:**

```

#include<stdio.h>
main()
{
    int i,a,b,r;
    int suma=0;
    printf("valores inicial final y razón ");
    scanf("%d %d %d",&a,&b,&r);

    for (i=a;i<=b;i*=r)
    {
        printf("%d ",i);
        suma+=i;
    }

    printf("\nLa suma vale %d\n",suma);
}

```

### **3.6 Ejercicios propuestos**

1. Realizar un programa que le pida un número entero 'n' al usuario, mostrándole las tablas de sumar y de multiplicar de dicho número n, así como su cuadrado y su cubo. Una vez mostrado todo esto, volverá a pedir otro número al usuario, y así indefinidamente hasta que se pulse ctrl.+C o se cierre la aplicación.
2. Realizar un programa que pida un número entero 'n' al usuario y le informe si dicho número es primo o no es primo. La forma más sencilla es dividir dicho número por todos los números menores que el (excepto el 1). Si no es divisible por ninguno, entonces es primo.
3. Realizar un programa que cuente desde el número A hasta el número B de N en N, siendo A, B y N introducidos por el usuario. Se tiene que permitir la cuenta hacia atrás, si N es negativo. En cualquier caso, el programa tiene que comprobar que A y B son compatibles con un salto dado, es decir, si el salto es positivo, A debe de ser mayor que B.
4. Realizar un juego en el que el jugador piensa un número del 1 al 100 y el ordenador trata de adivinarlo. El jugador debe informar al ordenador si ha acertado, se ha quedado corto o se ha pasado.



# 4. Tipos avanzados de datos

## 4.1 Vectores

Cuando en lugar de una sola variable se debe tratar con varias simultáneamente, la nomenclatura se torna más engorrosa ya que cada variable necesita ser declarada independientemente con un nombre identificativo propio.

Para salvar este inconveniente, el lenguaje C ofrece la posibilidad de agrupar variables del mismo tipo bajo un único nombre genérico, simplificando así la nomenclatura, pero para distinguir una variable de otra se utilizan otras variables enteras llamadas índices. A esta nueva estructura de datos se le denomina vector o arreglo (array).

### 4.1.1 Declaración de vectores

La declaración de un array es como la de cualquier otra variable. Debe indicarse su tipo y especificar entre corchetes cuantos elementos tendrá el vector.

*tipo nombre [cantidad de elementos];*

Por ejemplo, supóngase que tenemos 3 sensores térmicos que nos proporcionan un valor de temperatura en grados y con decimales. Podríamos declarar 3 variables separadas, pero en vez de eso declaramos un vector de 3 componentes como sigue:

```
#include <stdio.h>

void main()
{
    float temp[3];

    //sentencias del programa
}
```

Esta sentencia indica que se ha definido un vector que contendrá valores `float` llamado “temp”, con tres elementos, numerados del 0 al 2. Nótese bien esta numeración, empezando por 0 y terminando por el tamaño del vector menos una unidad.

Es importante resaltar que la cantidad de elementos no puede ser una variable, tiene que ser siempre una constante. Es decir, el siguiente código es un error de sintaxis:

```
void main()
{
    int num_sensores=3;
    float temp[num_sensores]; //error de sintaxis

    //sentencias del programa
}
```

No se puede poner el tamaño como variable, pero sí se pueden utilizar directivas del preprocesador (que realmente son constantes). Estas directivas son realmente una equivalencia total entre la etiqueta y el valor, hay que tener en cuenta que no son variables. En el siguiente ejemplo NUM\_SENSORES, realmente es la constante 3:

```
#define NUM_SENSORES 3

void main()
{
    float temp[NUM_SENSORES]; //esto es correcto

    //sentencias del programa
}
```

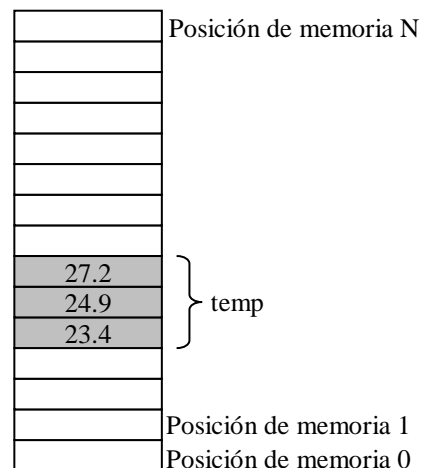
#### 4.1.2 Inicialización de un vector

Al igual que en los tipos básicos de datos, al declarar un vector, también se puede dar un valor inicial a cada una de sus componentes, sin embargo esto no es común salvo que el vector contenga muy pocos elementos.

Si quisiéramos dar un valor inicial a las componentes del vector del anterior ejemplo, podríamos hacerlo de la siguiente forma:

```
void main()
{
    float temp[3]={23.4, 24.9, 27.2};

    //sentencias del programa
}
```



**Figura 4-1. Reserva de memoria de un vector**

Como se observa en la inicialización anterior, cada elemento se separa mediante una coma “,” y se colocan entre llaves. Los valores dados a los elementos del vector se “guardan” en memoria tal como se muestra en la figura.

Otra opción de inicialización consiste en dejar que el compilador “cuente” por sí sólo la cantidad de elementos que contiene el vector:

```
float temp[]={23.4, 24.9, 27.2};
```

Como puede notarse no se coloca ningún número de elementos entre los corchetes, pero sí se dan los valores iniciales de los elementos. Este código es

totalmente equivalente al anterior. Al compilar el programa se cuentan el número de elementos entre llaves y se utiliza este número como si se hubiera puesto entre los corchetes.

Cada elemento del vector se almacena en posiciones consecutivas de memoria tal y como se refleja en la figura anterior. La posición de todo el vector es decidida por el sistema operativo, al igual que las variables normales, pero las componentes siempre estarán en posiciones consecutivas.

También existe la opción de inicializar solo la primera o primeras componentes de un vector. En el ejemplo anterior, podríamos poner:

```
float temp[3]={23.4}; //asigna la primera, resto a 0
float temp[3]={23.4, 24.9}; //inicializa la 1 y 2, la tercera a 0
```

En el primer caso inicializaría la primera componente a 23.4 y el resto les asignaría el valor 0 por defecto. En el segundo, inicializaría primera y segunda componente y el resto a 0. Nótese que aquí si es importante poner la dimensión del vector (3) si deseamos utilizar un vector de 3 componentes, porque en caso contrario, el vector tendría nada más que 1 componente.

### 4.1.3 Acceso a datos de un vector

Si se desea acceder individualmente a un dato sólo basta con colocar el nombre del vector y la posición que ocupa el dato en cuestión entre corchetes. Si en el ejemplo anterior se desea asignar valores a cada una de las componentes por separado, para posteriormente hallar la media de las componentes del vector, haríamos:

```
#include <stdio.h>

void main()
{
    float temp[3];
    float media;

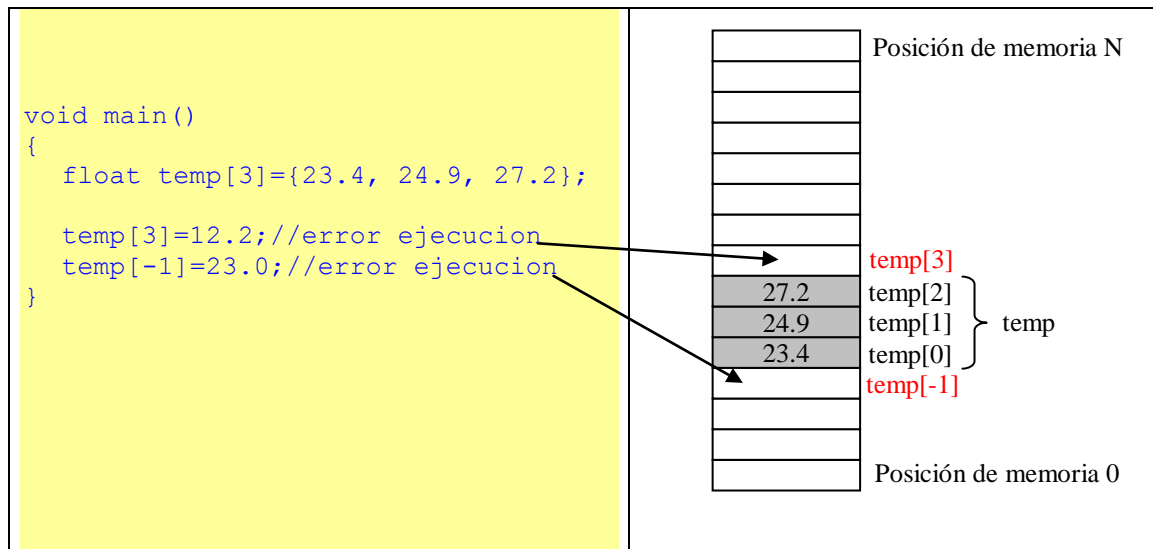
    //escribimos en las componentes
    temp[0]=13.5;
    temp[1]=24.5;
    temp[2]=31.2;

    //leemos las componentes
    media=(temp[0]+temp[1]+temp[2])/3.0f;

    printf("Temp media: %f\n",media);
}
```

El compilador no controla el tamaño de los vectores, si el programador intenta acceder a una componente que desborda el tamaño del vector, el compilador no produce errores, pero se producirá un error grave en la ejecución del programa. Nótese como el siguiente programa intenta acceder a unas componentes que quedan fuera del mismo, causando un error de ejecución.

**NOTA: En un vector de dimensión N, las componentes de dicho vector van numeradas de la 0 a la N-1**



**Figura 4-2. Error en ejecución por desbordamiento de vector.**

Por supuesto, también se pueden pedir los datos al usuario:

```
#include <stdio.h>
void main()
{
    float temp[3];
    float media;

    //pedimos los datos
    printf("Escriba las temperaturas: ");
    scanf("%f %f %f",&temp[0],&temp[1],&temp[2]);

    media=(temp[0]+temp[1]+temp[2])/3.0f;

    printf("Temp media: %f\n",media);
}
```

**Escriba las temperaturas: 12 15 16**  
**Temp media: 14.333334**

O también se podría haber pedido solo 1 componente:

```
#include <stdio.h>
void main()
{
    int i;
    float temp[3]={23.4, 24.9, 27.2};
    printf("Cual es la temperatura del sensor 1? ");
    scanf("%f",&temp[1]);
    printf("Las nuevas temperaturas son:\n");
    for(i=0; i<3; i++)
        printf("Sensor %d Temperatura %f\n", i, temp[i]);
}
```

El manejo por componentes visto anteriormente, no es realmente común. El acceso a componentes admite que se utilice una variable para indicar el número de componentes (recuérdese que eso NO se puede hacer al declarar el vector, pero sí al acceder a una componente). En el caso anterior, si se quieren sacar por pantalla los datos del vector (imagínese ahora que es de 5 componentes), en vez de hacerlo para cada componente `temp[0]`, `temp[1]`, etc., se recurre a un bucle que haga la tarea y utilizando el índice para determinar la componente:

```
#include <stdio.h>

void main()
{
    float temp[5]={12.1, 23.2, 45.1, 25.0, 31.3};
    int i;
    for(i=0; i<5; i++)
    {
        printf("Temperatura sensor %d=%f\n",i,temp[i]);
    }
}
```

```
Temperatura sensor 0=12.100000
Temperatura sensor 1=23.200001
Temperatura sensor 2=45.099998
Temperatura sensor 3=25.000000
Temperatura sensor 4=31.299999
```

Generalmente los datos, en vez de estar inicializados en el código, se solicitaran al usuario por teclado. Así un programa que solicite los datos de las temperaturas y las guarde en el vector sería:

```
#include <stdio.h>

void main()
{
    int i;
    float temp[5];

    for(i=0; i<5; i++)
    {
        printf("Temperatura sensor %d: ",i);
        scanf("%f",&temp[i]);
    }
}
```

Si además quisiéramos que el programa permitiera consultar la temperatura de un sensor particular, tantas veces como deseemos, podríamos hacer inmediatamente a continuación del `for`:

```
//for anterior
while(1)
{
    printf("Sensor: ");
    scanf("%d",&i);
    if(i>=0 && i<5)    //fijese en la comprobacion de indice
        printf("Sensor %d mide %f grados\n",i,temp[i]);
    else
        printf("Indice erroneo\n");
}
} //llave de cierre del main
```

```
Temperatura sensor 0: 12
Temperatura sensor 1: 23
Temperatura sensor 2 :34
Temperatura sensor 3: 45
Temperatura sensor 4: 56
Sensor: 14
Indice erroneo
Sensor: 1
Sensor 1 mide 23.000000 grados
Sensor: 4
Sensor 4 mide 56.000000 grados
```

**Ejemplo:** Introducir los días que tiene cada mes del año e imprimirlo después por pantalla.

```
#include<stdio.h>

void main()
{
    int mes[12],i;
    for(i=0;i<12;i++)
    {
        printf("Introduzca los dias del mes %d: ",i+1);
        scanf("%d",&mes[i]);
    }
    for(i=0;i<12;i++)
        printf("El mes %d tiene %d dias\n",i+1,mes[i]);
}
```

**Ejemplo:** Introducir 10 datos (enteros) por teclado e imprimir después la lista de dichos datos en orden inverso

```
#include <stdio.h>

void main()
{
    int i, vect[10];

    printf("Escriba 10 datos:\n");
    for(i=0;i<10;i++)
    {
        printf("Dato %d: ",i);
        scanf("%d",&vect[i]);
    }
    printf("Datos en orden inverso:");
    for(i=9;i>=0;i--)
        printf("%d ",vect[i]);
}
```

#### 4.1.4 Operaciones con vectores

En este apartado se muestran algunos ejemplos de operaciones típicas con vectores como el modulo, la media de los elementos de un vector, el valor absoluto de un vector, el producto escalar, etc. Los ejemplos manejan generalmente un vector inicializado en código, por simplicidad, pero lo normal es que los datos de los vectores fueran obtenidos de otra forma, por ejemplo pedidos al usuario. También se omite muchas veces el código para sacar por pantalla los vectores por el mismo motivo:

**Ejemplo:** Dado un vector de 5 números enteros que teclea el usuario, convertirlo a su valor absoluto

```
#include <stdio.h>
void main()
{
    int datos[5];
    int i;

    //primero pedimos los datos
    for(i=0;i<5;i++)
    {
        printf("Dato %d: ",i);
        scanf("%d",&datos[i]);
    }
}
```

```

//valor absoluto de cada componente
for(i=0;i<5;i++)
{
    if(datos[i]<0)
        datos[i]=-datos[i];
}

//sacamos el vector por pantalla
printf("Vector: ");
for(i=0;i<5;i++)
    printf("%d ",datos[i]);
printf("\n");
}

```

**Ejemplo:** Calcular el valor medio de los elementos de un vector de 10 elementos tecleados por el usuario.

```

#include <stdio.h>
#define N 10

void main()
{
    int i;
    float vect[N];
    float med, suma;

    //pedir los datos
    for(i=0;i<N;i++)
    {
        printf("Elemento %d: ",i);
        scanf("%f",&vect[i]);
    }
    //calcular la media
    suma=0;
    for(i=0;i<N;i++)
    {
        suma+=vect[i];
    }
    med=suma/N;
    printf("La media es: %f\n", med);
}

```

Nótese en este ejemplo el uso de `#define`. Recuérdese que a lo largo del programa `N` no es más que otra forma de llamar a la constante 10

**Ejemplo:** Realizar el producto escalar entre los dos vectores `v1` y `v2`

```

#include <stdio.h>
void main()
{
    float v1[5]={1,34,32,45,34};
    float v2[5]={12,-3,34,15,-5};
    float prod=0;
    int i;

    for(i=0;i<5;i++)
    {
        prod+=v1[i]*v2[i];
    }
    printf("El producto escalar es: %f\n",prod);
}

```

**Ejemplo: Realizar la suma de los 2 vectores v1 y v2 sobre v3 (v3=v1+v2)**

```
#include <stdio.h>

void main()
{
    float v1[5]={1,34,32,45,34};
    float v2[5]={12,-3,34,15,-5};
    float v3[5];

    int i;

    for(i=0;i<5;i++)
    {
        v3[i]=v1[i]+v2[i];
    }

    printf("Vector3: ");
    for(i=0;i<5;i++)
        printf("%f ",v3[i]);
    printf("\n");
}
```

**Ejemplo: Multiplicar el vector v1 por una constante “num”, para calcular v2=num\*v1**

```
#include <stdio.h>

void main()
{
    float v1[5]={1,34,32,45,34};
    float v2[5];
    float num=3;
    int i;

    for(i=0;i<5;i++)
    {
        v2[i]=num*v1[i];
    }

    printf("Vector1: ");
    for(i=0;i<5;i++)
        printf("%f ",v1[i]);
    printf("\n");

    printf("Vector2: ");
    for(i=0;i<5;i++)
        printf("%f ",v2[i]);
    printf("\n");
}
```

### 4.1.5 Vectores de dimensión variable

Hasta ahora hemos visto vectores en los que la dimensión o número de componentes del mismo era conocido y fijo. Si el vector tenía 10 componentes y se le pedían al usuario, había que pedirle siempre las 10 componentes. Imagínese ahora que se desea repetir el ejercicio en el que se pedía una serie de números al usuario y luego mostraba dicha lista en orden inverso, pero no se quiere fijar el número de componentes en el código, sino que se desea que el usuario pueda decir de cuantos datos consta esa lista.



Cuando declaramos un vector, siempre va explicito el número de componentes, que es constante y fijo, no puede ser cambiado. Lo que si que se puede hacer es declarar un vector de un elevado número de componentes ‘m’ y utilizar solo las “n” primeras. Obviamente “n” tiene que ser menor que el tamaño del vector. De esta forma se desaprovechan ‘m-n’ componentes. Esta forma de trabajar con la denominada memoria estática no es la forma más correcta de hacerlo, pero no obstante es sencilla y valida para algunos casos simples. Si se desea hacerlo más formalmente, hay que utilizar memoria dinámica.

```
#include <stdio.h>

void main()
{
    int i, vect[100];
    int n;
    printf("Cuantos datos: ");
    scanf("%d",&n); //el usuario debe teclear un n<100

    for(i=0;i<n;i++) //fijese como utilizamos solo las "n" primeras
    {
        printf("Dato %d: ",i);
        scanf("%d",&vect[i]);
    }
    printf("Datos en orden inverso:");
    for(i=n-1;i>=0;i--)
        printf("%d ",vect[i]);
}
```

De la misma forma, si quisiéramos calcular el modulo de un vector, y el número de componentes no se conoce a priori, se podría hacer:

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

void main()
{
    int i,n;
    float v[100];
    float modulo,suma;

    printf("Cuantos datos: ");
    scanf("%d",&n); //el usuario debe teclear un n<100

    for(i=0;i<n;i++)
    {
        printf("Dato %d: ",i);
        scanf("%f",&v[i]);
    }

    suma=0.0f;
    for(i=0;i<n;i++)
    {
        suma+=v[i]*v[i];
    }
    modulo=sqrt(suma);
    printf("El modulo es: %f\n",modulo);
}
```

### 4.1.6 Ordenamiento de un vector

Ordenar un vector es una tarea relativamente compleja. Se puede hacer de diversas formas, con distintos algoritmos, aunque algunos de ellos son más eficientes que los otros, es decir, realizan la tarea más rápido. Se muestra a continuación una manera de hacerlo, que no es la más eficiente, pero no es complicada de entender.

```
#include <stdio.h>

void main()
{
    int datos[5];
    int i,j;

    //primero pedimos los datos
    for(i=0;i<5;i++)
    {
        printf("Dato %d: ",i);
        scanf("%d",&datos[i]);
    }

    //ahora lo ordenamos
    for(i=0;i<4;i++)
    {
        for(j=i+1;j<5;j++)
        {
            if(datos[i]>datos[j])
            {
                int aux=datos[i];
                datos[i]=datos[j];
                datos[j]=aux;
            }
        }
    }

    //sacamos el vector por pantalla
    printf("Vector: ");
    for(i=0;i<5;i++)
        printf("%d ",datos[i]);
    printf("\n");
}
```

## 4.2 Matrices

La matriz es una estructura de datos común dentro de los lenguajes de programación y conceptualmente son idénticas a sus homónimas matemáticas. Por tanto una matriz es un conjunto de datos de un tamaño definido que se encuentran consecutivos en memoria y en la que es posible el acceso al elemento que deseemos simplemente con indicar su posición.

Las matrices son extremadamente útiles para trabajar con multitud de problemas matemáticos que se formulan de esta forma o para mantener tablas de datos a los que se accede con frecuencia.

### 4.2.1 Declaración de matrices

Para declarar un vector (1 dimensión) se indica entre corchetes el tamaño o número de componentes del vector. Para matrices bidimensionales y

multidimensionales, hay que indicar el tamaño o número de componentes de cada dimensión. La declaración de una matriz en lenguaje C es como sigue:

```
tipo nombre_matriz[tamaño1][tamaño2][tamaño3]....;
```

Si la matriz es bidimensional, tamaño1 y tamaño2 hacen referencia a las filas y columnas de dicha matriz:

```
tipo nombre_matriz[filas][columnas];
```

Al igual que sucede con los vectores, entre los corchetes solo puede existir una constante (número). Si se pone una variable sería un error de sintaxis. Como ejemplo de declaración se mencionan las siguientes:

```
// Matriz de números reales de 10x10
float matriz[10][10];
//Matriz tridimensional de números enteros 20x20x10
int Tridimensional[20][20][10];
```

## 4.2.2 Inicialización de matrices

Las matrices como el resto de las declaraciones, se pueden inicializar en el momento de su declaración, empleando llaves para separar cada fila y comas (,) para separar los elementos de la fila.

```
//matriz de 2 filas por 3 columnas de numeros enteros
int matriz[2][3] = { { 1,2,3 },
                    { 4,5,6 }
                  };
```

También se admite poner todos los elementos consecutivos. En este caso se van asignando los valores primero por filas y luego por columnas. Por lo tanto la inicialización anterior es equivalente a:

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

## 4.2.3 Acceso a datos de una matriz

El acceso a los datos de una matriz es idéntico al de un vector, solo que en este caso hay que referenciar la fila y la columna. Por ejemplo, para mostrar el contenido de la matriz anterior por pantalla, se necesitarían dos bucle `for` anidados, uno que va iterando sobre el número de filas y el otro (más interno) sobre el número de columnas. Así se van imprimiendo las filas consecutivamente.

```
#include <stdio.h>

void main()
{
    int i,j;
    int matriz[2][3]= { { 1,2,3 },
                        { 4,5,6 } };

    for(i=0;i<2;i++)//para cada fila
    {
        for(j=0;j<3;j++)//para cada columna
        {
            printf("%d\t",matriz[i][j]);
        }
        printf("\n");
    }
}
```

**Ejemplo: Suma de dos matrices m1 y m2 sobre m3 (m3=m1+m2)**

```
#include <stdio.h>

void main()
{
    int i,j;
    int m1[2][3]= { 1,2,3,4,5,6 };
    int m2[2][3]= { 4,5,12,23,-5,6 };
    int m3[2][3];

    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            m3[i][j]=m1[i][j]+m2[i][j];

    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("%d\t",m3[i][j]);
        }
        printf("\n");
    }
}
```

**Ejemplo: Algoritmo para multiplicar matrices de 3x3**

```
#include <stdio.h>

void main(void)
{
    int p[3][3] = { {1, 3, -4}, {1, 1, -2}, {-1, -2, 5} };
    int q[3][3] = { {8, 3, 0}, {3, 10, 2}, {0, 2, 6} };
    int r[3][3];

    int i, j, k; //Variables inices
    int sum;

    //Multiplica las matrices p y q
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            sum=0;
            for(k=0; k<3; k++)
                sum+=p[i][k]*q[k][j];
            r[i][j]=sum;
        }
    }

    //Imprime el resultado
    for (i=0; i<3; i++)
    {
        for (j=0; j<3; j++)
            printf("%d\t", r[i][j]);
        printf("\n");
    }
}
```

### 4.3 Cadenas de caracteres

Si el tipo del vector definido es de tipo `char` se trata de una cadena de caracteres. En una cadena cada caracter ocupa una posición de memoria (1 Byte) y debe terminar siempre con el caracter nulo (`\0`), que el programador debe incluir de forma obligada. Tal como sucede en los vectores, el nombre de la cadena de caracteres es un puntero al primer elemento.

Cada caracter se almacena como su código ASCII y el caracter nulo como 0. Una de las utilidades del caracter nulo radica en indicar a las funciones que manejan caracteres donde termina la cadena. Estas funciones se encuentran en el fichero de cabecera `string.h`. Si por error no se incluye el `\0` se tendrá error de ejecución.

#### 4.3.1 Declaración de cadenas

La declaración de una cadena de caracteres difiere considerablemente de la inicialización de un vector numérico. La formalización de la declaración debe seguir el siguiente formato:

```
char nombre_cadena [cantidad de elementos];
```

Supóngase que se desea declarar una cadena de caracteres que contenga un máximo de 20 elementos (incluido el caracter nulo de final de cadena, es decir 19 letras útiles), se puede proceder del siguiente modo:

```
#include <stdio.h>

void main()
{
    char cad[20];
}
```

#### 4.3.2 Inicialización de cadenas, entrada y salida por consola

Para inicializar una cadena de caracteres puede seguirse cualquiera de las siguientes opciones:

```
char texto[6]={76,73,66,82,79,0}; //valores ASCII
char texto[6]='L','I','B','R','O','\0'; //Caracteres ASCII
char texto[6]="LIBRO";
char texto[]="LIBRO";
```

En la primera sentencia se inicializa la cadena con la palabra deseada definida por los códigos de la tabla ASCII. En el segundo caso es igual, pero colocando cada letra, es decir cada elemento de la cadena, entre apóstrofes y separados por “,”. Tal como sucede con los vectores, aquí debe colocarse el valor inicial entre llaves y no debe olvidarse el caracter `\0`.

En la segunda y tercera se inicializa la cadena “`texto`” con la palabra `LIBRO` entre comillas y es el compilador quien agrega el `\0` y, además, en la tercera opción “cuenta” cuántos caracteres contiene la palabra.

Si se desea capturar una cadena desde el teclado se puede realizar, por ejemplo, de utilizando la función `gets()` o bien `scanf()` especificando `%s` para cadenas. La forma más sencilla de pedir una cadena de caracteres (incluyendo los espacios) y sacarla por pantalla sería:

```
#include <stdio.h>

void main()
{
    char frase[30];

    printf("Ingresa una frase: \n");
    gets(frase);
    printf("La frase es\n");
    puts(frase); //totalmente equivalente a printf("%s\n",frase);
}
```

Para leer palabras, se puede utilizar la función `scanf()` con el especificador `%s`. La función `scanf()` deja de leer si se introduce un espacio, una tabulación o un retorno de carro. Por ejemplo, el siguiente programa pregunta el nombre y apellido y luego lo muestra todo junto:

```
#include <stdio.h>

void main()
{
    char nombre[10], apellido[20];

    printf("Cual es tu nombre?");
    scanf("%s",nombre); //notese que no lleva &
    printf("Cual es tu apellido? ");
    scanf("%s",apellido);
    printf("Tu nombre completo es %s %s\n", nombre, apellido);
}
```

Nótese que el `scanf()` cuando aplica a cadenas de caracteres, no lleva el `&`

### 4.3.3 Operaciones con cadenas

La forma de trabajar con cadenas de caracteres es distinta a los vectores. Mientras que los vectores vienen definidos por el número de componentes, las cadenas de caracteres están delimitadas por el caracter `'\0'`. Para trabajar con una cadena, habitualmente se recorre uno a uno sus caracteres con un bucle `while()` hasta que se encuentra dicho caracter final de cadena.

**Ejemplo:** Cuento la cantidad de caracteres que hay en la frase: “Don Quijote de La Mancha.”

```
#include <stdio.h>

void main()
{
    char cad[]="Don Quijote de La Mancha";
    int i=0;
    while(cad[i]!='\0')
    {
        i++;
    }
    printf("La frase: %s tiene %d caracteres",cad,i);
}
```

**La frase Don Quijote de La Mancha tiene 24 caracteres**

**Ejemplo:** Realizar un programa que convierte una palabra (por ejemplo un nombre) de minúsculas a mayúsculas

```
#include<stdio.h>

void main()
{
    char nombre[75];
    int i;
    int desp='A'-'a';
    printf ("Escriba su nombre: ");
    scanf("%s",nombre);

    i=0;
    while(nombre[i]!='\0')
    {
        if(nombre[i]>='a' && nombre[i]<='z') //si no es minuscula, nada
            nombre[i]=nombre[i]+desp;
        i++;
    };

    printf("En Mayusculas: %s\n",nombre);
}
```

**Ejemplo:** Realizar un programa que cuente el número de veces que aparece una letra cualquiera (que se le pregunta al usuario) en una palabra cualquiera (que también se le pregunta al usuario)

```
#include<stdio.h>

void main()
{
    char palabra[75];
    char letra;
    int i,cont=0;
    printf("Letra:");
    scanf("%c",&letra);
    printf ("Escriba una palabra: ");
    scanf("%s",palabra);

    i=0;
    while(palabra[i]!='\0')
    {
        if(palabra[i]==letra)
            cont++;
        i++;
    };

    printf("Nnumero de veces: %d\n",cont);
}
```

**Ejemplo:** Concatenar dos cadenas

```
#include<stdio.h>

void main()
{
    char frase1[100],frase2[100];
    int i,j;

    printf ("Escriba una frase: ");
    gets(frase1);
    printf ("Escriba otra frase: ");
    gets(frase2);
}
```

```

i=0;
while(frase1[i]!='\0')
{
    i++;
};

j=0;
while(frase2[j]!='\0')
{
    frase1[i]=frase2[j];
    i++;
    j++;
};
frase1[i]='\0';

printf("Concatenadas: %s\n",frase1);
}

```

#### 4.3.4 Librería string.h

Muchas de las funciones de manejo común de cadenas de caracteres no es necesario implementarlas, ya que ya están programadas en una librería, cuyo fichero de cabecera que hay que incluir se llama `<string.h>`

```
#include <string.h>
```

##### 4.3.4.1 Longitud de una cadena

La función `strlen()` devuelve el número de caracteres que tiene la cadena sin contar el `'\0'`.

Ejemplo: Contar la cantidad de caracteres del nombre tecleado por el usuario.

```

#include <stdio.h>
#include <string.h>
void main()
{
    char nombre[100];
    int longitud;
    printf("Introduce tu nombre: ");
    scanf("%s",nombre);
    longitud = strlen(nombre);
    printf("Tu nombre tiene %d caracteres\n", longitud );
}

```

```

Introduce tu nombre: Pepe
Tu nombre tiene 4 caracteres

```

##### 4.3.4.2 Pasar a mayúsculas

La función `_strupr()` convierte a mayúsculas la cadena de caracteres.

```

#include <stdio.h>
#include <string.h>
void main()
{
    char nombre[100];
    printf("Introduce tu nombre: ");
    scanf("%s",nombre);
    _strupr(nombre);
    printf("En mayusculas %s\n", nombre);
}

```



#### 4.3.4.3 Copiar una cadena

La función `strcpy()` copia una cadena en otra. Se debe tener cuidado que la “cadena receptora” tenga espacio suficiente. Si la cadena origen es más larga que la cadena destino, los caracteres sobrantes serán eliminados.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char cadena_a_copiar[ ] = "Aprendo el Lenguaje C.";
    char destino[50];

    strcpy( destino, cadena_a_copiar );
    printf( "Valor final: %s\n", destino );
}
```

#### 4.3.4.4 Concatenar cadenas

La función `strcat()` copia una cadena a continuación de otra.

En el siguiente ejemplo se copia la cadena “de Arco” a la cadena “Juana”.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char nombre_completo[50];
    char nombre[ ]="Juana";
    char apellido[ ]="de Arco";
    strcpy( nombre_completo, nombre );
    strcat( nombre_completo, " " ); // Copia un espacio en blanco
    strcat( nombre_completo, apellido );
    printf( "El nombre completo es: %s.\n", nombre_completo);
}
```

**El nombre completo es: Juana de Arco.**

#### 4.3.4.5 Comparar cadenas

La función `strcmp()` devuelve un 0 si las dos frases son iguales, devuelve un 1 si dichas frases están ordenadas alfabéticamente y -1 si no lo están.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char frase1[100],frase2[100];
    int orden;
    printf( "Escriba una frase: " );
    gets(frase1);
    printf( "Escriba otra frase: " );
    gets(frase2);

    orden=strcmp(frase1,frase2);
    if(orden==0)
        printf("Iguales\n");
    if(orden==1)
        printf("Frase1>frase2 ordenadas alfabeticamente\n");
    if(orden==-1)
        printf("Frase1<frase2 no ordenadas alfabeticamente\n");
}
```

## 4.4 Estructuras

Las estructuras son un conjunto de datos de diferentes tipos agrupados bajo un único identificador.

Cada elemento de una estructura recibe el nombre de miembro o campo, y a diferencia de los vectores en los que cada elemento es del mismo tipo, cada miembro puede ser de un tipo diferente y se almacenan en posiciones contiguas de memoria.

### 4.4.1 Creación de una estructura

Antes de declarar una variable del tipo estructura, se debe definir previamente el modelo que tendrá dicha estructura de datos. Si bien no es obligatorio, se recomienda que la definición de este modelo se realice previa a la función `main()`.

Para declarar una estructura debe recurrirse a la palabra clave del Lenguaje C `struct`. La declaración de una estructura de datos es la siguiente:

```
struct nombre_estructura
{
    tipo1 miembro1;
    tipo2 miembro2;
    .....;
};
```

Supongamos que se desean almacenar los datos de un contacto para un listín de teléfonos. Para ello se procederá a definir una estructura de datos llamada `contacto` con los siguientes campos:

```
struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};
```

Como puede verse, la estructura contiene diferentes variables que permitirán almacenar datos de un contacto.

### 4.4.2 Declaración de una variable de tipo estructura

Debe remarcarse que una estructura de datos se comporta como un tipo nuevo de datos, esto implica que `struct contacto` es ahora un tipo de variable nueva por lo que el paso siguiente es declarar una variable tipo `contacto` denominada `amigo`. La forma de declarar una variable correspondiente a una estructura es:

```
struct nombre_estructura variable;
```

En el caso anterior para declarar una variable de tipo “contacto” llamada `amigo`, nuestro programa quedaría como:

```
struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};
```

```
void main()
{
    struct contacto amigo;
}
```

Obviamente, si tuviéramos que declarar varias variables, podríamos hacerlo en varias líneas, o incluso en una sola, tal y como se hace para variables de tipos de datos básicos:

```
struct contacto amigo1, amigo2;
```

No obstante, cuando hay que manejar un conjunto de datos (en este caso una agenda completa) de un determinado tipo (tipo “contacto”), lo normal es utilizar vectores de estructuras, tal y como se describirá posteriormente.

### 4.4.3 Inicialización de una estructura

Si el programador quiere asignar unos valores iniciales al declarar una variable de tipo estructura, puede hacerlo de forma similar a la de los vectores. En la línea de la declaración se añade una asignación y entre llaves se ponen los valores iniciales de cada uno de los miembros de la estructura. Estos valores tienen que ir en el mismo orden en el que han sido escritos en la definición de la estructura. Así, si en el ejemplo anterior quisiéramos que la variable “amigo” tuviera como contenidos iniciales el nombre “Pepe”, el teléfono 1234567 y una edad de 23 años, haríamos:

```
struct contacto amigo={"Pepe", 1234567, 23};
```

Se puede asignar, igual que se hacía con vectores, solo los primeros miembros. El resto de ellos quedarían asignados el valor 0 por defecto.

```
struct contacto amigo={"Pepe"}; //el telefono seria 0 y la edad 0
```

### 4.4.4 Acceso a miembros de una estructura

El acceso a un miembro cualquiera de una estructura se realiza a través del operador punto: “.” tal como se muestra a continuación:

*nombre\_variable.miembro*

En el ejemplo anterior, para asignar la edad del contacto denominado amigo se haría:

```
amigo.edad=25;
```

De esta forma, podemos hacer un pequeño programa que pida los datos de un contacto y que finalmente los saque por pantalla. Se pone en la columna el programa que realiza la misma tarea pero con variables independientes, no agrupadas en la misma estructura de datos. Aunque esta forma de trabajar produce el mismo resultado por pantalla, para realizar programas más avanzados no es nada adecuada, por lo que se recomienda el uso de las estructuras:

```
#include <stdio.h>

struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};
```

```
#include <stdio.h>

void main()
{
    char nombre[30];
    int telefono;
    char edad;
```

<pre> void main() {     struct contacto amigo;      printf("Nombre: " );     scanf("%s", amigo.nombre );     printf("Telefono: " );     scanf("%d", &amp;amigo.telefono );     printf("Edad: " );     scanf( "%d", &amp;amigo.edad );      printf("***Contacto**\n");     printf("%s\n",amigo.nombre);     printf("Tlf: %d\n",amigo.telefono);     printf("Edad: %d\n",amigo.edad); } </pre>	<pre> printf("Nombre: "); scanf("%s", nombre); printf("Telefono: "); scanf("%d", &amp;telefono); printf("Edad: "); scanf( "%d", &amp;edad);  printf("***Contacto**\n"); printf("%s\n",nombre); printf("Tlf %d\n",telefono); printf("Edad: %d\n",edad); } </pre>
<pre> Nombre: Pepe Telefono: 1234567 Edad: 23 **Contacto** Pepe Tlf: 1234567 Edad: 23 </pre>	

#### 4.4.5 Copia de estructuras

Las estructuras tienen una propiedad interesante que no tienen los vectores, la posibilidad de realizar una copia o asignación directa de una variable de tipo estructura a otra variable de tipo estructura (del mismo tipo). Cuando se realiza esta copia se hace una asignación directa miembro a miembro de todos los miembros de la estructura. Fíjese en el siguiente ejemplo:

<pre> struct contacto amigo= {"Pepe",1234567,23}; struct contacto amigo2;  amigo2=amigo; //copia  printf("%s\n",amigo2.nombre); printf("Tlf: %d\n",amigo2.telefono); printf("Edad: %d\n",amigo2.edad); </pre>
<pre> Pepe Tlf: 1234567 Edad: 23 </pre>

Se recuerda que los vectores para ser copiados, hay que realizar la copia componente a componente (o utilizar funciones como `memcpy()` que exceden el contenido de este libro)

#### 4.4.6 Vector de estructuras

Si se necesitan almacenar varios datos con el mismo formato de estructura se puede recurrir a la definición de un vector de estructuras, es decir un vector cuyos elementos sea una estructura definida según las necesidades. La verdadera utilidad de las estructuras es esta. En vez de manejar un vector de números, se puede manejar un vector de estructuras. En el ejemplo anterior, si queremos gestionar una agenda de contactos, esta agenda tendrá típicamente un número elevado, por ejemplo algunas centenas. Por supuesto, no se declararan centenas de estructuras separadas, sino un único vector, cuyo índice nos permitirá acceder a cada contacto particular.

#### 4.4.6.1 Declaración e inicialización de un vector de estructuras

La declaración de un vector de estructuras sigue los principios presentados en los apartados anteriores tanto para vectores como para estructuras. Una vez definido el modelo de estructuras, en la función `main()` se debe declarar un vector incluyendo la palabra clave `struct` tal como se muestra a continuación:

```
struct nombre_estructura nombre_variable[cant_elem];
```

En el ejemplo anterior, podríamos gestionar una agenda de por ejemplo, 100 contactos. La declaración quedaría:

```
struct contacto agenda[100];
```

De tal forma que la variable “agenda” es un vector de 100 componentes, y cada una de esas componentes es una estructura “contacto” que a su vez tiene nombre, número de teléfono y edad.

La inicialización de un vector de estructuras se hace mediante la combinación de la inicialización de un vector y la de una estructura, de forma similar a como se inicializan las matrices. Así, los datos de cada contacto irían entre llaves, tal y como se hacía para una estructura simple. Cada una de las componentes del vector, se separa por comas, y el vector a su vez también se delimita por llaves

```
struct contacto agenda[3]={ {"Pepe",1234567,23},
                             {"Maria",2345678,17},
                             {"Paco",9123456,34}      };
```

#### 4.4.6.2 Acceso a un vector de estructuras

Accediendo a través del índice del vector, se puede obtener cada uno de los contactos. En el caso anterior, si queremos mostrar el contenido de la agenda por pantalla, el programa quedaría como sigue:

```
#include <stdio.h>

struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};

void main()
{
    int i;
    struct contacto agenda[3]={ {"Pepe",1234567,23},
                                {"Maria",2345678,17},
                                {"Paco",9123456,34}      };

    for(i=0;i<3;i++)
    {
        printf("**Contacto**\n");
        printf("%s\n",agenda[i].nombre);
        printf("Tlf: %d\n",agenda[i].telefono);
        printf("Edad: %d\n",agenda[i].edad);
    }
}
```

Si en vez de simplemente mostrar la agenda, se codifica una comparación de alguno de los miembros, ya se dispone de una agenda con función de búsqueda.

**Ejemplo:** Modificar el programa anterior para que permita al usuario hacer una búsqueda inversa, es decir teclear un teléfono y que el programa muestre a todos los contactos que tienen ese número de teléfono (varios contactos pueden tener el mismo teléfono común).

```
#include <stdio.h>
struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};

void main()
{
    int i;
    int telefono;
    struct contacto agenda[3]={ {"Pepe",1234567,23},
                                {"Maria",1234567,17},
                                {"Paco",9123456,34}    };

    while(1)
    {
        printf("Telefono a buscar: ");
        scanf("%d",&telefono);

        for(i=0;i<3;i++)
        {
            if(telefono==agenda[i].telefono)
            {
                printf("**Contacto**\n");
                printf("%s\n",agenda[i].nombre);
                printf("Tlf: %d\n",agenda[i].telefono);
                printf("Edad: %d\n",agenda[i].edad);
            }
        }
    }
}
```

```
Telefono a buscar: 91111111
Telefono a buscar: 92332432
Telefono a buscar: 1234567
**Contacto**
Pepe
Tlf: 1234567
Edad: 23
**Contacto**
Maria
Tlf: 1234567
Edad: 17
Telefono a buscar: 9123456
**Contacto**
Paco
Tlf: 9123456
Edad: 34
```

En este ejemplo, si el teléfono no se encuentra, simplemente no se muestra nada por pantalla, aunque se podría mostrar un mensaje informando de ello. Modificar este programa para conseguir más funcionalidades como preguntar un nombre y mostrar el teléfono, u ordenar alfabéticamente la lista de contactos se consiguen fácilmente y se recomiendan al lector como ejercicio.

### 4.4.7 Estructuras que contienen otras estructuras y/o vectores

Como se ha visto en la creación de estructuras, cada uno de los miembros de una estructura va precedido de su tipo, que no necesariamente tiene que ser un tipo de datos básico. De hecho puede ser otra estructura o un vector de datos.

En el ejemplo anterior podríamos querer almacenar la fecha de nacimiento de nuestros contactos, para saber cuando es su cumpleaños. Aunque se podrían añadir tres miembros directamente a la estructura “contacto” para representar el día, el mes y el año, también se puede crear una estructura “fecha”, y utilizarla como sigue:

```
#include <stdio.h>

struct fecha
{
    char dia;
    char mes;
    int agno;
};

struct contacto
{
    char nombre[30];
    int telefono;
    struct fecha nacimiento;
};

void main()
{
    struct contacto amigo={"Pepe",1234567,{1,5,1970}};
    struct fecha hoy={1,5,2007};

    printf("%s nacio:\n",amigo.nombre);
    printf("Dia %d\n",amigo.nacimiento.dia);
    printf("Mes %d\n",amigo.nacimiento.mes);
    printf("Año %d\n",amigo.nacimiento.agno);

    if(hoy.dia==amigo.nacimiento.dia && hoy.mes==amigo.nacimiento.mes)
        printf("Hoy es el cumpleaños de %s!\n",amigo.nombre);
}
```

```
Pepe nacio:
Dia 1
Mes 5
Año 1970
Hoy es el cumpleaños de Pepe!
```

De la misma forma, uno de los miembros puede ser un vector de datos. Supóngase una estructura para manejar los datos de un alumno, y en concreto contiene el nombre, número de matrícula y un vector de cinco números reales representando las notas. Para sacar por pantalla dichas notas, el programa podría ser:

```
#include <stdio.h>

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

```

void main()
{
    int i;
    struct alumno alum={"Juan", 93321, {10, 5, 3, 4, 8}};

    printf("Notas de %s-%d:\n", alum.nombre, alum.matricula);
    for(i=0; i<5; i++)
    {
        printf("Nota %d: %f\n", i, alum.notas[i]);
    }
}

```

```

Notas de Juan-93321:
Nota 0: 10.000000
Nota 1: 5.000000
Nota 2: 3.000000
Nota 3: 4.000000
Nota 4: 8.000000

```

## 4.5 Uniones

Las uniones de datos tienen un tratamiento parecido a las estructuras, pero la diferencia radica en que en memoria se reserva espacio para la variable que más espacio requiera. El uso de uniones es mucho más esporádico que el de estructuras, y su interés es limitado, justificado únicamente por razones de eficiencia (y escapando realmente al alcance de este libro).

Las uniones son variables que pueden contener en diferentes momentos diferentes tipos de valores con diferentes tamaños. La presentación de una unión es como la de una estructura:

```

union nombre
{
    tipo1 nombre1;
    tipo2 nombre2;
    ...
};

```

El valor que se recupere de una unión debe ser del mismo tipo que el del último valor almacenado. Es por tanto responsabilidad del programador controlar el tipo de valor almacenado en todo momento en la unión. Si se define una unión número capaz de almacenar dos tipos diferentes: entero y decimal se pondría:

```

union numero
{
    int entero;
    float decimal;
};

```

La variable número será lo suficientemente grande para contener el mayor de los valores tipo que la integran. Se puede asignar a número cualquiera de los dos tipos y usar después en expresiones. Todos los miembros de una unión comienzan su almacenamiento en la misma dirección de memoria; si hay espacio sobrante quedará vacío. Si queremos una variable del tipo número podríamos declararla:

```

unión numero n;

```

Podremos, al igual que en las estructuras, seleccionar uno de los campos (sólo uno en un instante determinado) empleando el operador “.”:



```
n.entero=2;
```

Si se accede leyendo un miembro distinto al último tipo que ha sido asignado, su valor es indeterminado:

```
#include <stdio.h>

union numero
{
    int entero;
    float decimal;
};

void main()
{
    union numero n;
    n.entero=2;
    n.decimal=3.0f;
    printf("%d %f\n",n.entero,n.decimal);
}
```

```
1077936128 3.000000
```

Nótese en este ejemplo que el primer número mostrado por pantalla no es ni 2, ni 3, sino que el computador al coger unos bytes que han sido almacenados como de coma flotante e interpretarlos como un entero, utiliza un convenio de codificación diferente, con un resultado “basura”.

## 4.6 Enumeraciones

Con la construcción `enum` se pueden definir tipos de datos enteros que tengan un rango limitado de valores, y darle un nombre significativos a cada uno de los posibles valores.

La forma como se define una enumeración es de forma parecida a como se hace con las estructuras, usando la palabra clave `enum` para el comienzo de un tipo de enumeración. Su formato es:

```
enum nombre_enum { lista_de_enumeración };
```

Por ejemplo, para gestionar y almacenar la información del día de la semana, podríamos utilizar una variable de tipo entero y utilizar los números 1 para el lunes, 2 para el martes, y así hasta el 7 para el domingo. En vez de eso, se puede utilizar una enumeración como sigue:

```
#include <stdio.h>

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

void main()
{
    enum dia hoy;
    hoy = sabado;

    printf("%d\n",hoy); //Por pantalla saldra '5'
}
```

En el ejemplo anterior los identificadores `lunes=0`, `martes=1`, etc. Si se desea que la cuenta comience en 1, se indica:

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

En este caso los valores van del 1 al 7. También se pueden dar valores individuales:

```
enum codigo_telefonico
{
    Andalucia=95, Madrid=91, Barcelona=93
};
```

## 4.7 Tipos definidos por el usuario

Con `typedef` se pueden crear nuevos nombres de tipos de datos. Por ejemplo si se desea que la palabra `ENTERO` sea equivalente a `int`, se pondría:

```
typedef int ENTERO;
```

Se podría usar en adelante `ENTERO` en declaraciones como si fuese `int`. El utilizar mayúsculas es conveniente para resaltar el significado de la palabra definida. Así en nuestro programa podríamos hacer:

```
ENTERO a=3; //totalmente equivalente a "int a=3;"
```

La ventaja del `typedef` cuando se manejan estructuras es que se evita tener que poner la palabra `"struct"` cuando declaramos las variables. Así, podríamos declarar el tipo definido por el usuario `"fecha"` con la estructura vista anteriormente:

```
#include <stdio.h>

typedef struct fecha
{
    int dia;
    int mes;
    int agno;
} fecha;

void main()
{
    fecha hoy; //Notese como no es necesaria "struct"

    hoy.agno=2007;
    hoy.dia=27;
    hoy.mes=11;

    printf("Hoy es: %d-%d-%d\n", hoy.dia, hoy.mes, hoy.agno );
}
```

## 4.8 Ejercicios resueltos

**Ej. 4.1) Hacer un programa en el que se sustituya la componente de valor mínimo de un vector por el valor 0, y mostrar el vector por pantalla después**

SOLUCION:

```
#include <stdio.h>

void main()
{
    int datos[5]={1,3,-2,4,5};
    int i;
    int minimo;
    int indice_minimo;

    //para el minimo
    minimo=datos[0];
    indice_minimo=0;

    for(i=1;i<5;i++)
    {
        if(minimo>datos[i])
        {
            minimo=datos[i];
            indice_minimo=i;
        }
    }
    printf("Min %d y esta en %d\n",minimo,indice_minimo);
    datos[indice_minimo]=0;

    //sacamos el vector por pantalla
    printf("Vector: ");
    for(i=0;i<5;i++)
        printf("%d ",datos[i]);
    printf("\n");
}
```

**Ej. 4.2) Solicitar un número N de datos enteros al usuario, donde N es tecleado por el usuario y será siempre menor que 500. Después de pedir los datos el programa cambiara los datos que tengan un valor impar por el valor par inmediatamente inferior. Finalmente, el programa mostrar la lista final, en la que todos los números tienen que ser pares.**

**SOLUCION:**

```
#include <stdio.h>
void main()
{
    int datos[500];
    int i,n;

    printf("Cuantos datos:");
    scanf("%d",&n);

    //primero pedimos los datos
    for(i=0;i<n;i++)
    {
        printf("Dato %d: ",i);
        scanf("%d",&datos[i]);
    }

    for(i=0;i<n;i++)
    {
        if(datos[i]%2==1)
            datos[i]-=1;
    }
}
```

```
//sacamos el vector por pantalla
printf("Vector: ");
for(i=0;i<n;i++)
    printf("%d ",datos[i]);
printf("\n");
}
```

**Ej. 4.3) Pedir al usuario una letra y una palabra. El programa primero mostrar dicha palabra en las que todas las ocurrencias de dicha letra hayan sido sustituidas por el caracter '\*'. Finalmente se omitirán de la cadena (eliminándolas) dichas ocurrencias. Se muestra como ejemplo una salida típica por pantalla:**

```
Letra:a
Escriba una palabra: Palabra
palabra: P*l*br*
palabra: Plbr
```

**SOLUCION:**

```
#include<stdio.h>

void main()
{
    char palabra[75];
    char letra;
    int i,j;
    printf("Letra:");
    scanf("%c",&letra);
    printf ("Escriba una palabra: ");
    scanf("%s",palabra);

    i=0;
    while(palabra[i]!='\0') //sustituir por '*'
    {
        if(palabra[i]==letra)
            palabra[i]='*';
        i++;
    }
    printf("palabra: %s\n",palabra);

    i=0;
    j=0;
    while(palabra[i]!='\0') //eliminar los '*'
    {
        palabra[i]=palabra[j];
        if(palabra[i]!='*')
            i++;
        j++;
    }
    printf("palabra: %s\n",palabra);
}
```

**Ej. 4.4) Hacer un programa que permita buscar en una agenda de contactos mediante el nombre de la persona. Los miembros de la estructura que se debe manejar serán nombre, teléfono y edad. Los datos de la agenda serán escritos en código directamente en la inicialización de un vector de estructuras, utilizar 3 contactos de ejemplo. El programa informara si no se ha encontrado ningún contacto, mostrara todos los contactos**

**que coincidan con el nombre e informara del número de contactos encontrados.**

**SOLUCION:**

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

struct contacto
{
    char nombre[30];
    int telefono;
    char edad;
};

void main()
{
    int i;
    char nombre[30];
    int encontrado;
    struct contacto agenda[3]={ {"Pepe",1234567,23},
                                {"Maria",1234567,17},
                                {"Paco",9123456,34}    };

    while(1)
    {
        printf("Persona a buscar: ");
        scanf("%s",nombre);

        encontrado=0; //hemos encontrado 0, de momento
        for(i=0;i<3;i++)
        {
            int iguales;
            iguales=strcmp(nombre,agenda[i].nombre);
            if(iguales==0) //si son iguales
            {
                encontrado++;
                printf("**Contacto**\n");
                printf("%s\n",agenda[i].nombre);
                printf("Tlf: %d\n",agenda[i].telefono);
                printf("Edad: %d\n",agenda[i].edad);
            }
        }
        if(encontrado==0)
            printf("Contacto no encontrado\n");
        else
            printf("Se han encontrado %d contactos\n",encontrado);
    }
}
```

```
Persona a buscar: Pepe
**Contacto**
Pepe
Tlf: 1234567
Edad: 23
Se han encontrado 1 contactos
Persona a buscar: Marcos
Contacto no encontrado
```

**Ej. 4.5) Hacer un programa que permita obtener las notas de un colectivo de alumnos (un máximo de 30) que se introducen por teclado, el nombre y la nota de cada uno de ellos. El programa muestra por pantalla el tanto por ciento de aprobados y al alumno de máxima nota:**

```
Numero de alumnos: 3
Nombre: Paco
Nota: 6
Nombre: Maria
Nota: 7.5
Nombre: Pepe
Nota: 3
Aprobados 66.666667
Mejor: Nombre: Maria Nota: 7.500000
```

**SOLUCION:**

```
#include<stdio.h>

struct ficha
{
    char nombre[60];
    float nota;
};

void main()
{
    struct ficha alumnos[30];
    int num_alumnos;
    int num_aprobados=0;
    int i;
    struct ficha mejor;//el mejor alumno

    printf("Numero de alumnos: ");
    scanf("%d",&num_alumnos);

    for(i=0;i<num_alumnos;i++)
    {
        printf("Nombre: ");
        scanf("%s",alumnos[i].nombre);
        printf("Nota: ");
        scanf("%f",&alumnos[i].nota);
    }

    mejor.nota=0;
    for(i=0;i<num_alumnos;i++)
    {
        if(alumnos[i].nota>=5)
            num_aprobados++;

        if(alumnos[i].nota>mejor.nota)
            mejor=alumnos[i];
    }
    printf("Aprobados %f\n",num_aprobados*100.0/num_alumnos);
    printf("Mejor: Nombre: %s Nota: %f\n",mejor.nombre,mejor.nota);
}
```

# 5. Punteros

## 5.1 Organización de la memoria

Antes de definir qué es un puntero conviene recordar cómo está organizada la memoria del ordenador, ya que ambos conceptos están íntimamente relacionados. Normalmente, la memoria del ordenador contiene  $2^{32}$  posiciones o direcciones (unas cuatro mil millones), cada una ocupada por un byte. Podemos imaginar esta memoria como una matriz compuesta por  $N=2^{32}$  filas con una anchura de 8 bits, tal y como se representa en la figura siguiente:

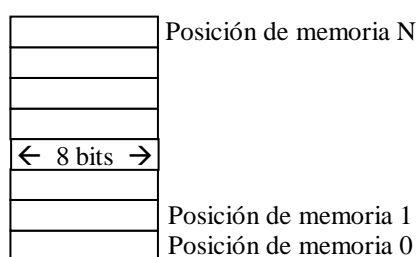


Figura 5-1 Organización de la memoria del ordenador

Los bytes se organizan en grupos, por ejemplo, un dato de `int` ocupa cuatro posiciones de memoria. Si en un programa hacemos la siguiente declaración e inicialización:

```
int x=27;
```

Podemos imaginar que la variable `x` se almacena en la memoria como se representa en la figura siguiente:

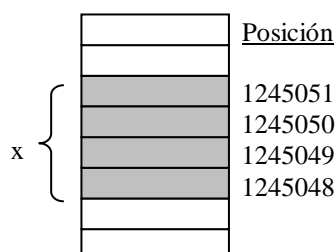
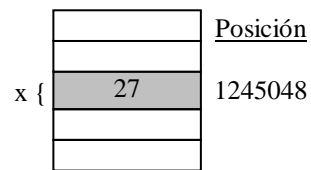


Figura 5-2. Almacenamiento en memoria de la variable `x`

En general, cuando se hace referencia a la posición de memoria ocupada por una variable se sobreentiende que es la dirección de memoria que ocupa el primer byte de dicha variable. Por ejemplo, en el caso de `x`, su dirección de memoria sería la 1245048.

A partir de ahora cada variable se va a representar, independientemente de su tamaño, como si estuviese almacenada en una única posición de memoria. Por ejemplo, la variable `x` la representaríamos en la memoria como:

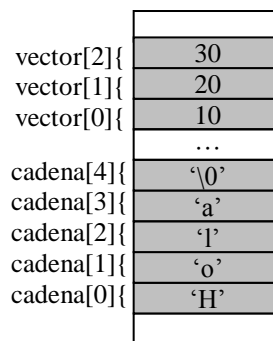


**Figura 5-3. Esquema simplificado de almacenamiento de la variable `x`**

Los vectores y las cadenas de caracteres están formados por un conjunto de variables que se guardan en posiciones consecutivas de memoria. Por ejemplo, si hacemos la siguiente declaración:

```
int vector[3]={10,20,30};
char cadena[]="Hola";
```

El vector y la cadena quedarían almacenados de forma parecida a ésta:



**Figura 5-4. Ejemplo de almacenamiento de un vector y una cadena de caracteres**

En general, los elementos de cualquier matriz se guardan en posiciones consecutivas de memoria. Por ejemplo, si declaramos la matriz siguiente:

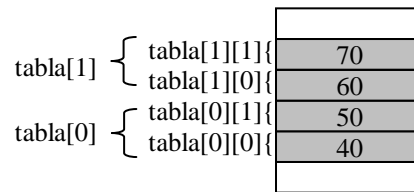
```
int tabla[2][2]={
    {40,50},
    {60,70}
};
```

Para nosotros es más fácil imaginarla tal y como se representa en matemáticas:

$$\begin{pmatrix} \text{tabla}[0][0] & \text{tabla}[0][1] \\ \text{tabla}[1][0] & \text{tabla}[1][1] \end{pmatrix} = \begin{pmatrix} 40 & 50 \\ 60 & 70 \end{pmatrix}$$

Sin embargo, realmente los elementos de la matriz se almacenan en la memoria del ordenador de la siguiente forma:





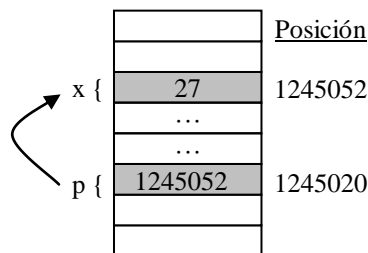
**Figura 5-5 Ejemplo de organización de la memoria con una matriz de dos dimensiones**

Es más, cada fila de la matriz se almacena como un vector que se puede manejar como tal. En el ejemplo anterior, los dos vectores a los que da lugar la declaración de la matriz son `tabla[0]` (con los elementos `tabla[0][0]` y `tabla[0][1]`) y `tabla[1]` (con los elementos `tabla[1][0]` y `tabla[1][1]`).

## 5.2 ¿Qué es un puntero?

Un puntero es una variable que contiene la dirección de memoria de un objeto, por ejemplo, de otra variable. El espacio de memoria ocupado por un puntero es el número de bytes necesarios para especificar una dirección de 32 bits, por tanto, un puntero ocupará 4 bytes.

Por ejemplo, en la figura siguiente se representa una supuesta memoria en la cuál hay almacenada una variable llamada `x`, de tipo entero, cuyo valor es 27 y otra variable llamada `p`, que es un puntero cuyo valor es precisamente la dirección de memoria ocupada por la variable `x`. En la jerga de los programadores se dice entonces que el puntero `p` apunta a `x`, lo cuál suele representarse gráficamente con una flecha.



**Figura 5-6 Puntero p apuntando a la variable x**

En este capítulo aprenderemos, entre otras cosas, cómo se le asigna a un puntero la dirección de memoria donde está almacenado un objeto o cómo podemos, a partir de su puntero, obtener el valor del objeto apuntado.

## 5.3 Declaración de un puntero

Un puntero se declara indicando el tipo de datos apuntados seguido de un nombre precedido por un asterisco, es decir:

```
tipo *nombre_puntero;
```

Por ejemplo, si queremos declarar un puntero que apunte a datos de tipo `char` y que se llame `pchar` escribiríamos:

```
char *pchar;
```

También se pueden declarar punteros genéricos que pueden apuntar a diferentes tipos de datos escribiendo la palabra reservada `void` en lugar del tipo, por ejemplo:

```
void *pgen;
```

Cuando se declara un puntero, tal como se ha hecho en los ejemplos anteriores, éste todavía no apunta a ningún objeto. Más adelante veremos cómo se inicia o se asigna una dirección de memoria a un puntero, pero antes vamos a presentar sus dos operadores básicos.

## 5.4 Operadores

Con los punteros se utilizan dos operadores: `&` y `*`.

El operador `&`, también conocido como “dirección de”, proporciona la dirección de memoria en la que está almacenado su operando.

Por ejemplo, si `x` es una variable, `&x` es la posición de memoria donde está almacenada dicha variable. Esta posición de memoria se asigna en el momento de la declaración y no cambia durante la ejecución del programa. Por ejemplo, el siguiente programa muestra en pantalla la dirección de memoria donde está almacenada una variable:

```
#include <stdio.h>

void main()
{
    int x;

    printf("La direccion de x es %d\n", &x);
}
```

Cuya salida en pantalla podría ser:

```
La direccion de x es 1245052
```

El operador `*`, también conocido como “indirección”, proporciona el contenido de una dirección de memoria a partir de su puntero.

Por ejemplo, si `x` es una variable y `p` es un puntero que apunta a `x` (es decir, `p=&x`) se cumple que `*p` es el contenido de la posición de memoria donde se guarda el valor de `x`. Es más, la variable `x` puede sustituirse por `*p` en cualquier lugar del programa y éste funcionará exactamente igual.

Veamos un ejemplo:

```
#include <stdio.h>
void main ()
{
    int x=27, *p;

    p=&x;
    printf("p apunta a la direccion %d\n", p);
    printf("cuyo contenido es %d\n", *p);
}
```

Cuya salida por pantalla podría ser:

```
p apunta a la direccion 1245052
cuyo contenido es 27
```

Analicemos cada una de las líneas del programa:

```
int x=27, *p;
```

Se declaran dos variables de tipo `int`, `x`, que se inicializa con el valor 27, y `p`, un puntero que todavía no apunta a ningún dato.

```
p=&x;
```

Al puntero `p` se le asigna la dirección de memoria ocupada por `x`, es decir, ahora `p` apunta a `x`.

```
printf("El puntero apunta a la direccion %d\n", p);
```

Se muestra en pantalla el valor del puntero `p`, es decir, la dirección a la que apunta, que en el ejemplo es la 1245052.

Finalmente, la instrucción:

```
printf("cuyo contenido es %d\n", *p);
```

Muestra en pantalla el contenido de la dirección a la que apunta `p`, es decir, el valor de la variable `x`, que vale 27. Como se ha contado anteriormente, si `p` apunta a `x`, `*p` puede utilizarse en lugar de `x` en cualquier lugar del programa. Por ejemplo:

```
#include <stdio.h>
void main()
{
    int x=27, y=3, suma, *p;

    p=&x;
    suma=*p+y;
    printf("La suma de x e y vale %d\n", suma);
}
```

Cuya salida es:

```
La suma de x e y vale 30
```

## 5.5 Iniciación de un puntero

Un puntero puede iniciarse en el momento de su declaración como cualquier otra variable. Por ejemplo:

```
int x, *p=&x;
```

El asterisco que aparece en la declaración del puntero no debe confundirse con el operador “indirección de”. Es decir, la declaración anterior es equivalente a las dos instrucciones siguientes:

```
int x, *p;
p=&x;
```

Un puntero también se puede iniciar con el valor especial `NULL` (puntero nulo):

```
char *pchar=NULL;
```

`NULL` es una constante simbólica que se utiliza en lugar de cero como mnemotécnico para indicar que se trata de un valor especial para un puntero. De hecho, no tiene sentido intentar escribir en la posición cero de memoria a la que apunta un puntero nulo, ya que ésta es una posición reservada para el sistema operativo.

El valor `NULL` suele utilizarse, además de para iniciar un puntero, para realizar operaciones de comprobación. Por ejemplo, una comprobación muy común consiste en

verificar si el puntero que devuelve una función tiene el valor NULL, lo que suele ser indicativo de que se ha producido un error.

Como hemos visto en el apartado anterior, a un puntero también se le puede asignar una dirección de memoria después de declararlo, por ejemplo:

```
int x, *p;
p=&x;           //p apunta a x
```

O cambiársela después:

```
int x, y, *p;
p=&x;           //p apunta a x
p=&y;           //p apunta ahora a y
```

En general, a un puntero se le asigna la dirección de memoria de un objeto declarado previamente. En los programas del apartado anterior, y únicamente con fines didácticos, se han realizado operaciones para mostrar que una variable tiene asociada una posición de memoria cuyo valor numérico se puede conocer.

Sin embargo, el programador no necesita saber cuáles las posiciones de memoria ocupadas por un objeto, simplemente asigna al puntero el valor devuelto por el operador &. De hecho, no tiene sentido que el programador asigne directamente direcciones de memoria al puntero, por ejemplo `p=1245052`, ya que es el sistema operativo el que gestiona la memoria.

## 5.6 Operaciones con punteros

### 5.6.1 Asignación

A un puntero se le puede asignar otro puntero del mismo tipo con el resultado de que ambos punteros apuntan al mismo objeto. Por ejemplo, el siguiente programa:

```
#include <stdio.h>

void main()
{
    int x=7, *p1, *p2;

    p1=&x;
    p2=p1;
    printf ("*p1 vale %d\n", *p1);
    printf ("*p2 vale %d\n", *p2);
}
```

Imprime en la pantalla:

```
*p1 vale 7
*p2 vale 7
```

A un puntero también se le puede asignar un puntero genérico (`void *`), en cuyo caso se produce una conversión implícita del puntero genérico para que apunte al tipo de datos al que apunta el otro puntero antes de hacer la asignación. Más adelante veremos un ejemplo.

### 5.6.2 Aritmética y comparación

Entre dos punteros que apuntan a elementos de una misma matriz, por ejemplo, un vector o una cadena de caracteres, se pueden realizar las operaciones aritméticas o de comparación que se describen a continuación.

A un puntero se le puede sumar o restar un número entero. Por ejemplo, si *p* es un puntero que apunta a un elemento de un vector y *n* un número entero, la operación de asignación:

```
p+=n;
```

Hace que el puntero apunte al elemento situado *n* posiciones más adelante. Por ejemplo, si *n*=1 el puntero apunta al siguiente elemento del vector. Veamos un ejemplo:

```
#include <stdio.h>

void main()
{
    int vector[5]={7, 1, -3, 4, 2};
    int *p;

    p=vector;           // Equivalente a p=&vector[0]
    printf("%d\n", *p); // Imprime 7
    p+=2;               // Ahora p=&vector[2]
    printf("%d\n", *p); // Imprime -3
}
```

Los operadores de incremento (++) y decremento (--) también se pueden utilizar para hacer que el puntero apunte siguiente al elemento siguiente o al anterior, respectivamente, al que estaba apuntando. Estos dos operadores tienen la misma prioridad que \* y & (si no se recuerda el orden de la asociatividad de los operadores, que en este caso particular es derecha a izquierda, conviene utilizar paréntesis cuando se utilicen juntos en la misma expresión para indicar qué operación se realiza primero).

El ejemplo siguiente muestra varias operaciones realizadas con un puntero después de haberle asignado la dirección de memoria de uno de los elementos de un vector:

```
void main()
{
    int vector[10], x, *p;

    vector[5]=17;

    p=&vector[5]; // p apunta a vector[5]

    x=*p+3;       // x=vector[5]+3=20
    x=*(p+1);     // x=vector[6], valor indeterminado
    (*p)++;       // vector[5]=18
    p++;          // p apunta ahora a vector[6]
}
```

Dos punteros que apuntan a los elementos de un mismo vector se pueden comparar, en el sentido de comparar sus direcciones de memoria. Por ejemplo, si *p* apunta al primer elemento de un vector y *q* al sexto, el resultado de la comparación *p*<*q* es verdadero. También se pueden restar dos punteros, lo cual da como resultado el número de elementos que existen entre ellos.

Veamos un programa en el que se realizan las dos operaciones anteriores:

```
#include <stdio.h>

void main()
{
    int vector[10];
    int *p, *q;
```

```

p=vector;           // Equivalente a p=&vector[0]
q=&vector[5];       // q apunta a vector[5] (sexto elemento)

printf("%d\n", q-p); // Resta de punteros (Imprime 5)

if (p<q)            // Comparación de punteros
{
    p+=3;           // p apunta ahora a vector[3]
    *p=27;          // vector[3]=27
}
printf("%d\n", vector[3]); // Imprime 27
}

```

En resumen, las operaciones válidas con punteros son la asignación de punteros del mismo tipo, sumar o restar un número entero al puntero, y restar o comparar punteros que apuntan a elementos de la misma matriz.

Como ya vimos anteriormente, también es válido asignar a un puntero el valor cero usando el mnemotécnico NULL:

```
p=NULL;
```

O compararlo con este valor, por ejemplo:

```

if (p == NULL)
    ...

```

## 5.7 Punteros a vectores

En C, existe una estrecha relación entre punteros y vectores. Tanto es así, que para acceder a los elementos de un vector puede hacerse indistintamente utilizando la notación con subíndices, que ya se ha visto en un capítulo anterior, o la notación con punteros que hemos aprendido en este capítulo. El uso de una notación u otra depende de la preferencia del programador, sin embargo el acceso a los elementos del vector con punteros es más rápido y los programadores profesionales suelen decantarse por esta última opción.

Por ejemplo, supongamos que hacemos la siguiente declaración:

```
int v[10], *p=&v[0];
```

En ella se declara un vector llamado *v* que tiene diez elementos (*v*[0], ... , *v*[9]) y un puntero, llamado *p*, que apunta al primer elemento del vector.

El nombre de un vector es un puntero que apunta al primer elemento del vector. En el ejemplo anterior *v* y *&v*[0] tienen por tanto el mismo valor, de manera que también podíamos haber escrito la declaración anterior como se hace habitualmente:

```
int v[10], *p=v;
```

Hay que tener cuidado porque el nombre de un vector es un puntero constante, cuya dirección no se puede modificar, y no está permitido realizar algunas de las operaciones descritas en los apartados anteriores. Por ejemplo:

```

int v[10], *p=v;

v++;           // Operación ilegal (p++ sí sería válido)
v=&v[3];       // Operación ilegal (p=&v[3] sí sería válido)

```

Siguiente con el ejemplo, si `p` apunta a `v[0]`, por definición `(p+1)` apunta a `v[1]`, `(p+2)` apunta a `v[2]`, y así sucesivamente. Además, si `p` apunta a `v[0]`, `*p` es el contenido de `v[0]`, y si `(p+1)` apunta a `v[1]`, `*(p+1)` es el contenido de `v[1]`, etc.

En general, se cumple que `(p+i)` es la dirección de `v[i]` y que `*(p+i)` es el contenido de `v[i]`. De esta forma, si asignamos al puntero `p` la primera dirección del vector `v`, en cualquier lugar donde aparezca `v[i]` podemos escribir `*(p+i)` y el programa funcionará exactamente igual.

El programa siguiente imprime los elementos de un vector accediendo a ellos con notación de subíndices y de vectores:

```
#include <stdio.h>

void main()
{
    int i, vector[3]={10,20,30}, *p=vector;

    for (i=0; i<3; i++)
        printf("%d\n", vector[i]);    //Notación con subíndices

    for (i=0; i<3; i++)
        printf("%d\n", *(vector+i)); //Notación con punteros

    for (i=0; i<3; i++)
        printf("%d\n", *(p+i));      //Notación con punteros

    for (i=0; i<3; i++)
        printf("%d\n", p[i]);        //Notación con subíndices
}
```

El programa siguiente calcula el módulo y el mínimo de un vector de cinco números enteros utilizando notación de punteros aplicada al puntero `p`.

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

void main()
{
    int v[5]={-1,2,4,-8,0}, *p=v;
    int i;
    int minimo,suma=0;
    float modulo;

    // Módulo del vector
    for(i=0;i<5;i++)
        suma+=(*(p+i))*(*(p+i));

    modulo=(float)sqrt(suma);
    printf("El modulo del vector vale %f\n", modulo);

    // Mínimo del vector
    minimo=*p; //minimo=v[0] o minimo=p[0]
    for(i=0;i<5;i++)
    {
        if (*(p+i)<minimo)
            minimo=*(p+i);
    }
    printf("El minimo del vector es %d\n", minimo);
}
```

Se puede resaltar que si se tiene:

*tipo vector[TAM];*  
*tipo\* p=vector; //o tipo\* p=&vector[0], es lo mismo*

Entonces es totalmente equivalente, para un índice “i”:

`vector[i] ⇔ *(p+i) ⇔ p[i] ⇔ *(vector+i)`

Figura 5-7. Equivalencia entre punteros y vectores

## 5.8 Punteros a cadenas de caracteres

Todo lo dicho para punteros y vectores puede aplicarse a cadenas de caracteres teniendo en cuenta que los elementos de la cadena son caracteres individuales en lugar de números y que el último elemento de la cadena es el caracter nulo (‘\0’).

Los programas siguientes ilustran el manejo de cadenas de caracteres utilizando nomenclatura de punteros. El primero de ellos imprime una cadena de caracteres, caracter a caracter, manejando el puntero de dos formas distintas:

```
#include <stdio.h>

void main()
{
    char mensaje[]="Hola Mundo";
    char *p=mensaje;           // p apunta a mensaje[0]
    int i;

    // Manteniendo el puntero apuntado a mensaje[0]
    i=0;
    while (*(p+i)!='\0')
    {
        printf("%c", *(p+i));
        i++;
    }
    printf("\n");

    // Moviendo el puntero por la cadena de caracteres
    while (*p!='\0')
    {
        printf("%c", *p);
        p++;
    }
    printf("\n");
}
```

El siguiente ejemplo cuenta e imprime en la pantalla el número de caracteres de una palabra introducida desde el teclado:

```
#include <stdio.h>

void main()
{
    char palabra[51], *p=palabra;
    int contador=0;

    printf("Introduzca una palabra\n");
    scanf("%s", palabra);
    while (*p!='\0')
    {
```



```

    contador++;        //Incrementa el número de caracteres leídos
    p++;              //Apunta el puntero al siguiente caracter
}
printf("%s tiene %d caracteres\n", palabra, contador);
}

```

## 5.9 Punteros a estructuras

Un puntero a una estructura se declara igual que cualquier otro puntero. Primero el tipo de datos (en este caso una estructura) y luego el nombre del puntero precedido por un asterisco, es decir:

```
struct nombre_estructura *nombre_puntero;
```

Por ejemplo:

```

struct fecha
{
    int dd;
    int mm;
    int aaaa;
};

void main()
{
    struct fecha *p;
    // Otras sentencias
}

```

También puede declararse mediante un tipo de datos sinónimo de la estructura creada con `typedef`, por ejemplo:

```

typedef struct
{
    int dd;
    int mm;
    int aaaa;
}fecha;

void main()
{
    fecha *p;
    // Otras sentencias
}

```

En ambos casos, se está declarando una estructura que tiene tres miembros de tipo `int` (día, mes y año) y un puntero, llamado `p`, que puede apuntar a variables de dicha estructura. El programa siguiente declara e inicializa el puntero para que apunte a una variable de tipo estructura, llamada `f`, declarada previamente:

```

typedef struct
{
    int dd;
    int mm;
    int aaaa;
}fecha;

void main()
{
    fecha f, *p=&f;
    // Otras sentencias
}

```

Para acceder a un miembro de una variable de tipo estructura a través de su puntero hay que escribir el nombre del puntero seguido del operador -> (símbolos “menos” y “mayor que” juntos) y del miembro de la estructura, es decir:

*nombre\_puntero->miembro*

Por ejemplo, en el programa siguiente asigna valores a los miembros de la estructura a través de su puntero y luego imprime en pantalla los valores asignados:

```
#include <stdio.h>

typedef struct
{
    int dd;
    int mm;
    int aaaa;
} fecha;

void main()
{
    fecha f, *p=&f;

    p->dd=29;
    p->mm=7;
    p->aaaa=2007;
    printf("%d-%d-%d\n", f.dd, f.mm, f.aaaa);
}
```

### 5.10 Punteros a punteros

Un puntero también puede apuntar a otro puntero. La declaración es idéntica a la de cualquier otro puntero con la salvedad de que en lugar de un asterisco, \*, se utilizan dos, \*\*, un símbolo que significa “doble indirección”.

Por ejemplo, en el siguiente programa, se declaran tres variables de tipo `int`: `x`, un puntero llamado `p`, y un puntero a un puntero llamado `pp`. A continuación, a la variable `x` se le asigna el valor 1, a `p` la dirección de `x` (`p` apunta a `x`) y a `pp` la dirección de `p` (`pp` apunta a `p`). Finalmente, la última línea del programa imprime el valor de `x` aplicando el operador de doble indirección al puntero `pp`.

```
#include <stdio.h>

void main()
{
    int x, *p, **pp;

    x=1;
    p=&x;
    pp=&p;
    printf("El valor de x es %d\n", **pp);
}
```

Gráficamente, podemos imaginar la relación entre las tres variables del programa anterior como se muestra en la figura siguiente:

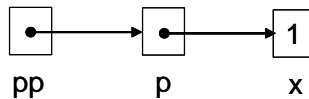


Figura 5-8. Ejemplo de un puntero apuntando a otro puntero

### 5.11 Asignación dinámica de memoria

En todos los programas que hemos visto hasta ahora, cuando se declara un objeto (una variable, un vector, una estructura, etc.) se le asigna automáticamente un espacio fijo de memoria que permanece ocupado por dicho objeto durante la ejecución del programa. A esta forma de asignar la memoria se le denomina estática porque el tamaño de los objetos se fija antes de ejecutar el programa.

Por ejemplo, cuando se usa un vector con una dimensión fija para manejar un número de datos variable puede ocurrir, por un lado, que si el número de datos es menor que el máximo que se puede almacenar se esté desaprovechando la memoria. Por el otro, la dimensión del vector puede ser insuficiente para almacenar todos los datos que se necesiten en un determinado momento.

Cuando los programas aumentan de tamaño y procesan gran cantidad de datos esto puede ser un serio problema. Para evitarlo, la memoria se puede gestionar de forma más eficiente utilizando lo que se denomina asignación dinámica de memoria, la cuál permite asignar espacios de memoria para almacenar los datos durante la ejecución del programa y liberar esos espacios cuando ya no son necesarios, los cuáles pueden ser reutilizados posteriormente.

La asignación dinámica de memoria, en combinación con los punteros, permite crear estructuras de datos muy potentes como listas enlazadas, árboles binarios, etc., cuya descripción va más allá de los objetivos de este libro. No obstante, a continuación se realiza una introducción básica a esta forma de gestionar la memoria y se presentan algunos ejemplos con estructuras y cadenas de caracteres creados de forma dinámica.

En C, la asignación dinámica de memoria se realiza con funciones especiales de la librería estándar `<stdlib.h>`, de las cuáles sólo vamos a presentar aquí las dos básicas: `malloc()` y `free()`.

La función `malloc()` permite asignar, durante la ejecución del programa, un bloque de memoria de `n` bytes consecutivos para almacenar los datos. Formalmente, su prototipo es el siguiente:

```
void *malloc (unsigned int n_bytes);
```

El parámetro formal de la función de la función, `n_bytes`, es un número entero sin signo que indica el número de bytes que se quieren asignar. La función devuelve un puntero genérico (`void *`) que apunta a la primera posición de memoria del bloque asignado o un puntero nulo (`NULL`) si no hay suficiente memoria disponible.

Para poder utilizar la función, hay que declarar un puntero que apunte al tipo de objetos que se almacenarán en el bloque de memoria reservado y al cuál se le asigna el puntero genérico devuelto por la función `malloc()`. Por ejemplo, si queremos reservar memoria para 10 números enteros podemos escribir:

```
int *p;
p=malloc(10*sizeof(int));    //También: p=malloc(40);
```

Cuando a `p`, que es un puntero que puede apuntar a datos enteros, se le asigna el puntero genérico (`void *`) devuelto por la función `malloc()` se produce una conversión implícita de este último a uno de tipo `int *` antes de hacer la asignación.

También se puede realizar una conversión explícita o forzada para expresarlo claramente o si se quiere que el código sea compatible con C++, en el que la conversión explícita de punteros es obligatoria. Por ejemplo:

```
p=(int *)malloc(10*sizeof(int));
```

En general, aunque el programador conozca el tamaño de los datos, por ejemplo, que un dato de tipo `int` ocupa 4 bytes, conviene para evitar errores y garantizar la portabilidad del código utilizar la función `sizeof()` que proporciona directamente el número de bytes que ocupa un objeto o un tipo de datos.

Normalmente, después de asignar la memoria suele comprobarse que todo ha ido bien, es decir, comprobar si el puntero devuelto por `malloc()` es `NULL` (lo que indicaría que la asignación de memoria ha fallado) y asegurar en este caso que no se realiza ninguna operación de lectura o escritura con el puntero.

Por ejemplo:

```
int *p;
p=malloc(10*sizeof(int));

if(p==NULL)
    printf("No hay memoria suficiente\n");
else
    printf("Memoria correctamente asignada\n");
```

Habitualmente, si el puntero devuelto por `malloc()` es `NULL`, además de escribir un mensaje de aviso, también suele detenerse la ejecución del programa. Para ello puede utilizarse, por ejemplo, la función `exit()` de la biblioteca `<stdlib.h>` cuyo prototipo es el siguiente:

```
void exit(int estado);
```

Para indicar que el programa se ha detenido normalmente escribiríamos `exit(0)` y en caso contrario `exit(1)`.

La función `free()` permite liberar un bloque de memoria previamente asignado. Su prototipo es el siguiente:

```
void free (void *q);
```

A la función simplemente hay que pasarle como argumento el puntero que apunta al bloque de memoria previamente asignado por la función `malloc()`.

Así, para liberar los 40 bytes de memoria reservados en los ejemplos anteriores escribiríamos:

```
free(p);
```

El programa siguiente rescribe un programa anterior que utilizaba una estructura para imprimir una fecha utilizando asignación dinámica de memoria:

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

typedef struct
{
    int dd, mm, aaaa;
}
fecha;

void main ()
{
    fecha *p;

    p=malloc(sizeof(fecha));
    if (p==NULL)
    {
        printf("No hay memoria disponible");
        exit(1);
    }
    p->dd=29;
    p->mm=7;
    p->aaaa=2007;
    printf("%d-%d-%d\n", p->dd, p->mm, p->aaaa);

    free(p);
}
```

El programa siguiente imprime al revés una palabra introducida desde el teclado:

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

void main()
{
    char *p;
    int i, longitud;

    p=malloc(51);
    if(p==NULL)
    {
        printf("No hay memoria disponible\n");
        exit(1);
    }
    printf("Introduzca una palabra\n");
    scanf("%s", p);

    longitud=strlen(p);
    for(i=longitud-1;i>=0;i--)
        printf("%c", *(p+i));        // También: printf("%c", p[i]);
    printf("\n");

    free(p);
}
```



# 6. Estructura de un programa

En los capítulos anteriores de este libro se han abordado conceptos fundamentales del lenguaje C tal como variables, construcciones condicionales e iterativas, vectores y cadenas entre otros. Sin embargo poco se ha dicho de cómo se debe estructurar un programa para que sea fácilmente interpretado por otra persona que no sea el programador y de como se puede manejar un programa complejo con miles de líneas de código de forma sencilla. La respuesta es el uso de funciones, subdividiendo el programa en trozos de código casi independientes que permiten el desarrollo rápido y libre de errores, además de permitir la reutilización del código generado en otros programas diferentes, acelerando el proceso de desarrollo software.

Se ha visto que la función principal de un programa en C (`main()`) utiliza funciones de la librerías estándar para realizar algunas tareas comunes (`printf()`, `scanf()`). No obstante el lenguaje C permite también la definición de funciones por parte del programador lo que permite estructurar el código facilitando la modularidad y portabilidad del mismo.

## 6.1 Directivas del preprocesador

Antes de comenzar con las funciones, se van a describir las directivas del preprocesador. Estas directivas realmente no forman parte del lenguaje C, ni son entendidas por el compilador, sino por un programa previo que se denomina preprocesador. Las directivas del preprocesador se distinguen de las líneas de código C porque comienzan con el símbolo `#`. A continuación se presentan las directivas del preprocesador más utilizadas

### 6.1.1 `#include`

La directiva `include` permite incluir en el código generado por el programador otros archivos de cabecera. Esta directiva puede usarse dos formas distintas:

```
#include <fichero.h>
#include "miFichero.h"
```

La diferencia entre utilizar los símbolos `< >` o las comillas `" "` radica en la ubicación de dicho fichero de cabecera. Cuando se utilizan `< >` se indica al preprocesador que busque directamente en los directorios que tenga configurados por defecto el entorno de desarrollo y en los que se suelen encontrar ficheros de librerías estándar, por ejemplo `<stdio.h>` o `<math.h>`. Cuando se utilizan comillas dobles, se le indica al preprocesador que primero busque en la carpeta actual de trabajo, en la que se suelen encontrar los archivos fuente del usuario. Si no lo encuentra aquí, buscara

después en la carpeta anteriormente descrita. Las comillas se usan por lo tanto generalmente para ficheros de cabecera del usuario.

Claramente la directiva `#include` es lo primero que se escribe en el código que se está desarrollando. El lenguaje C contiene una amplia variedad de archivos cabecera (.h) compuestos por diversas funciones que resultan de utilidad a la hora de realizar un programa. Se puede mencionar a modo de ejemplo, la librería `math.h` en la que están definidas varias funciones matemáticas y trigonométricas o la librería `string.h` en la que se dispone de un conjunto de funciones específicas para trabajar con cadenas de caracteres.

### 6.1.2 #define y #undef

Como su nombre indica, la directiva `define` permite definir símbolos que no son más que un nombre alternativo para una constante. Por su parte, la directiva `undef` permite eliminar símbolos previamente definidos. El uso de estas directivas es el siguiente:

*`#define nombre valor`*

Obsérvese que las directivas no llevan punto y coma (;) al final. Considere el siguiente ejemplo en el que se calcula:

```
#include <stdio.h>
#define PI 3.141592

void main()
{
    float rad;
    printf("Introduzca el radio de la circunferencia");
    scanf("%f",&rad);
    printf("El area de la circunferencia es %f\n", PI*rad*rad);
}
```

En el ejemplo anterior, se ha definido el símbolo `PI` a través de la directiva `#define` y se le da el valor numérico correspondiente. Si bien no es obligatorio, los símbolos definidos empleando esta directiva suelen escribirse en mayúsculas. De esta forma se facilita la lectura del programa y se lo diferencia claramente del resto de las variables. Lo que realiza el preprocesador es recorrer todo el archivo, sustituyendo literalmente las ocurrencias de “`PI`” por su valor numérico. Nótese que `PI` no es una variable, por lo que no se puede hacer en código

`PI=3.14159;`

Si en algún momento se desea que un símbolo definido deje de estarlo se hace con la directiva:

*`#undef nombre_símbolo`*

### 6.1.3 Macros

La directiva `#define` también permite definir macros, que son operaciones sobre datos o variables. La sintaxis de una macro es la siguiente:

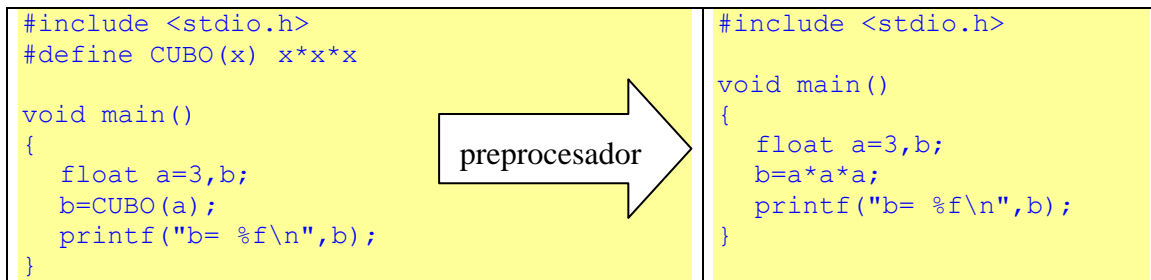
*`#define nombre_macro(param1, ..., paramN) código`*

Donde código es un conjunto válido de sentencias en C, y los parámetros (`param1, ..., paramN`) son símbolos que aparecen en código. Cuando el preprocesador



se ejecuta, sustituye cada llamada a la macro por el texto escrito en código, y también sustituye los parámetros por los valores que tengan cuando se llama a la macro.

Veamos un ejemplo:



En el código anterior se ha definido la macro `CUBO`, que tiene un único parámetro llamado “x”. Cuando el preprocesador encuentra esta macro, la sustituye directamente por el código definido, sustituyendo la “x” por lo que se encuentre entre paréntesis.

En la práctica, la utilización de las macros está desaconsejada a menos que sea estrictamente necesario. No obstante, en el caso de utilizarla, se recomienda que el código de la macro sea lo más sencillo posible.

## 6.2 Funciones. Estructura de un programa

Un programa que realice una tarea muy simple puede ser codificado dentro del `main()`, tal y como hemos trabajado anteriormente. Sin embargo, para problemas reales, es necesario subdividir el programa en partes razonablemente independientes que permitan un desarrollo más rápido y libre de errores, una depuración eficaz y un código que es fácilmente reutilizable en otras aplicaciones o programas.

La estructuración de un programa se hace dividiendo el código en trozos independientes (o casi) denominados funciones. Ya hemos visto anteriormente como se utilizan algunas funciones, como `printf()` y `scanf()` que son funciones incluidas en las librerías del compilador (en concreto mediante inclusión `<stdio.h>`), y otras matemáticas como `sqrt()` (de la librería `<math.h>`).

Este capítulo se centra en el desarrollo de funciones por parte del usuario, siendo este el encargado de crearlas, programarlas y utilizarlas en sus programas. Como punto de partida se propone el programa “Hola mundo” codificado con una función. Obviamente este es un ejemplo didáctico, pero sirve para presentar la estructura general de un programa con funciones.

```
//directivas de inclusion include
#include<stdio.h>

//directivas de sustitucion define
#define PI 3.14159

//prototipos de funciones
void Saluda();

//funcion main
void main()
{
    Saluda(); //llamada a la funcion
}
```

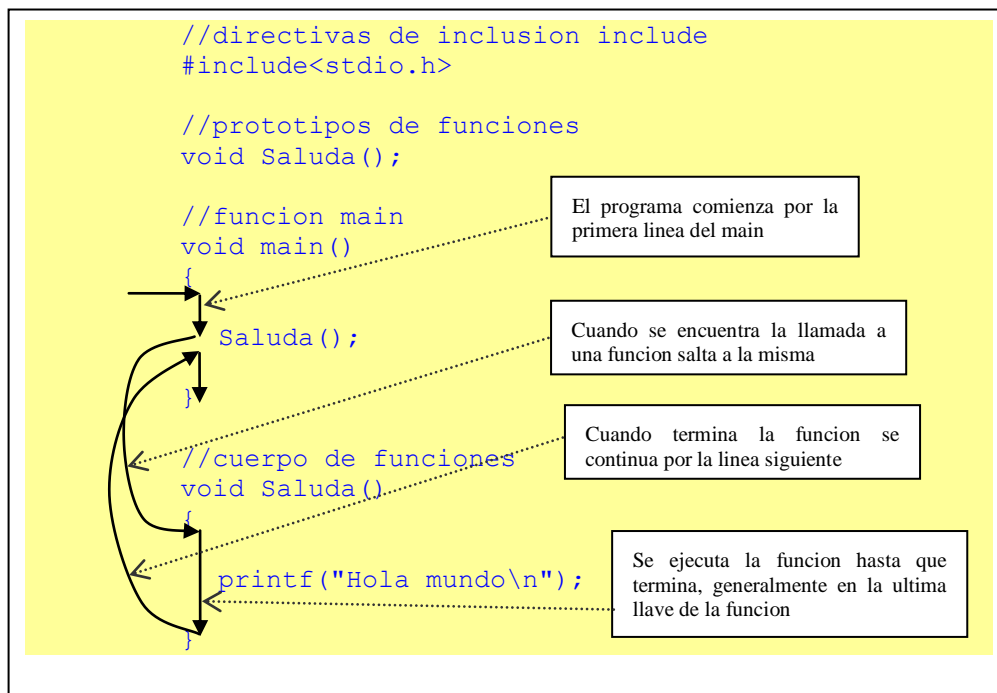
```
//cuerpo de funciones
void Saluda()
{
    printf("Hola mundo\n");
}
```

**Hola mundo**

Nótese bien que el cuerpo de la función `Saluda()` está **fuera** del `main()`, mientras que dentro del `main()` existe lo que se denomina llamada o invocación de la función.

### 6.2.1 Flujo de ejecución

Veamos a continuación el flujo de ejecución del programa anterior:



**Figura 6-1. Flujo de ejecución de un programa con funciones**

Como ya se describió en el capítulo 2, la función `main()` siempre debe existir en un programa, y una y solo una de las funciones puede ser denominada así.

Al llamar a una función se ejecutan las sentencias que la componen y, una vez finalizada la ejecución de la misma, la ejecución del programa continúa desde el punto en que se hizo la llamada a la función.

### 6.2.2 Declaración de una función: prototipo

Al igual que para utilizar una variable hay que declararla previamente, para programar una función, es necesario establecer su prototipo o declaración. El prototipo de una función consta de 3 partes principales:

1. El nombre de la función. Es un identificador que debe seguir las reglas establecidas para los identificadores (sin espacios, no comenzar por un número, no caracteres especiales, etc.). Además, el nombre debe de ser significativo e identificar la tarea que hace la función.

2. Una lista de parámetros a la función, siendo especificado el tipo (tipos básicos, vectores, estructuras) de cada uno de los parámetros. Esta lista de parámetros puede ser vacía, es decir no tener ningún parámetro.
3. El tipo de la función o tipo del valor de retorno de la función. Representa un valor que puede ser devuelto (aunque no es obligatorio devolver algo) como resultado de la ejecución de dicha función.

La sintaxis del prototipo de una función sería:

```
tipo nombre (tipo1 param1, tipo2 param2, ..., tipoN paramN);
```

Como ejemplo se pueden analizar algunas de las funciones de la librería `<math.h>`. Por ejemplo el prototipo de la función `sqrt()` para calcular la raíz cuadrada es básicamente el siguiente (que se puede encontrar en el fichero `math.h`):

```
double sqrt(double);
```

Este prototipo especifica que la función tiene un parámetro (de entrada a la función) que es el que está especificado entre paréntesis. Este parámetro es el número del que queremos hallar la raíz cuadrada, y que es un número real de precisión doble. Además se observa que la función es de tipo `double`, es decir devolverá como resultado un número real de precisión doble. Obviamente este número devuelto por la función es el que ha calculado como la raíz cuadrada del parámetro.

Nótese bien que en el prototipo no se especifica como se hace la operación de la raíz, sino solo la sintaxis de cómo se puede utilizar la función, que necesita como entrada y que da como salida.

En el caso anterior no se ha dado un nombre al parámetro, por ser obvio e innecesario. Realmente los nombres de los parámetros NO son necesarios en los prototipos pero SI muy convenientes. Supóngase que el usuario quiere ahora programar una función suya que le sirva para calcular el área de un círculo cuyo radio conoce. El prototipo de la función podría ser el siguiente:

```
float AreaCirculo(float radio);  
// equivalente a float AreaCirculo(float);
```

El usuario especifica que la función tiene 1 parámetro (efectivamente para calcular el área de un círculo solo necesitamos 1 dato), y la función devuelve un dato `float`, que será el área deseada. Aunque el nombre “radio” es opcional en los prototipos, es muy informativo, ya que especifica al programador que el dato de entrada es el radio y no el diámetro por ejemplo.

Si no se especifica ningún valor de retorno de una función, C asume que la función devuelve un valor de tipo `int`. Si no se desea que una función devuelva ningún valor, se puede establecer explícitamente con el tipo `void` (vacío). Esto es igualmente válido para los parámetros, se puede especificar explícitamente que la función no tiene parámetros mediante `void` o simplemente no poniendo ningún parámetro entre los paréntesis. En el apartado anterior, se vio la función Saluda con el siguiente prototipo:

```
void Saluda(); //equivalente a void Saluda(void);
```

Como se ve, esta función no necesita ningún parámetro de entrada ni devuelve ningún valor.

### 6.2.3 Definición de una función

El prototipo anteriormente descrito sirve para establecer la sintaxis de una función pero no como realiza la función su tarea respectiva. El conjunto de sentencias que realizan dicha tarea se denomina **cuerpo, definición o implementación** de la función.

El cuerpo de una función tiene la siguiente sintaxis:

```
tipo nombre (tipo1 param1, tipo2 param2, ..., tipoN paramN)
{
    //cuerpo de la función;
    //sentencias que operan sobre los parametros param1-paramN
}
```

Nótese como la primera línea coincide con el prototipo de la función, exceptuando el punto y coma final. Entre llaves aparece el conjunto de sentencias que realizan la tarea. Tal y como se ha descrito antes, en el prototipo no es obligatorio el uso de un nombre o identificador para cada parámetro. No obstante, en el cuerpo SI es obligatorio el uso de dichos nombres, ya que son los identificadores que van a ser utilizados dentro del cuerpo de la función. Además, se recomienda utilizar siempre idéntica nomenclatura tanto en el prototipo como en la definición, es decir que la primera línea coincida al 100% con la declaración de la función, exceptuando el punto y coma final.

Nótese que el cuerpo de la funciones NO esta dentro de la función `main()`, sino a nivel global. La función se define típicamente al finalizar la llave de cierre de la función `main()` tal como se muestra a continuación:

```
void main()
{
    //Sentencias del programa;
}

//Definición de la función
tipo nombre (tipo1 var1, ....., tipoN varN)
{
    //Cuerpo de la función;
}
```

Aunque todavía no se entienda en su totalidad, se presenta aquí el cuerpo de la función del ejemplo anterior que sirve para calcular el área de un círculo:

```
float AreaCirculo(float radio) //radio es una variable
{
    float area;
    area=3.14159*radio*radio; //se calcula el area
    return area; //se devuelve el valor calculado
}
```

El parámetro de la función es el radio, que se utiliza como dato de entrada. Nótese que no sabemos a priori lo que vale, ni se tiene que pedir su valor al usuario, es simplemente una variable. La función realiza el cálculo del área y finalmente devuelve este resultado mediante la palabra clave `return`.

### 6.2.4 Llamada a una función

Finalmente, la llamada a una función se realiza con el nombre de la misma y entre paréntesis se listan los parámetros que requiere la función para operar. El número y tipo de los parámetros empleados en la llamada a la función debe coincidir con el número y tipo de los parámetros formales del prototipo. La llamada a la función se puede realizar en cualquier línea de programación en la que se necesite. Adicionalmente, si la función devuelve algún valor (es decir, no es de tipo `void`) la llamada a la función debe estar incluida en una expresión que recoja el valor devuelto:

```
void main()
{
    float r,a;
    printf("Radio: ");
    scanf("%f",&r);
    a=AreaCirculo(r);           //llamada a la funcion, pasando "r" como
                                //radio y asignando el area a "a"
    printf("Area: %f\n",a);
}
```

Los datos empleados en la llamada a una función reciben el nombre de parámetros reales (en el ejemplo anterior “r”), ya que se refieren a la información que se transfiere a la función para que ésta se ejecute. Los nombres de los parámetros formales no tienen por qué coincidir con los nombres de las variables usadas como parámetros reales en el momento de la llamada.

### 6.2.5 ¿Donde se declara el prototipo de la función?

Lo realmente necesario es que cuando el compilador se encuentra una llamada a una función, conozca el prototipo de la misma, lo que se puede conseguir con la declaración de la función, o cambiando el orden establecido hasta ahora y poniendo en la parte superior (después de las directivas de preprocesador `#include`) el cuerpo de la función directamente y omitiendo el prototipo, ya que el cuerpo hace sus funciones. Aunque de esta forma el programa también compila y funciona correctamente, se prefiere el uso de los prototipos con la estructura del programa presentada anteriormente y que será utilizada a lo largo de este capítulo.

<pre>//MEJOR  #include &lt;stdio.h&gt;  float AreaCirculo(float radio);  void main() {     ...     a=AreaCirculo(r);     ... }  float AreaCirculo(float radio) {     //Cuerpo de la funcion }</pre>	<pre>//PEOR, aunque funciona  #include &lt;stdio.h&gt;  float AreaCirculo(float radio) {     //Cuerpo de la funcion }  void main() {     ...     a=AreaCirculo(r);     ... }</pre>
---	--

## 6.3 *Ámbito de una variable*

Las variables de un programa pueden clasificarse en función del ámbito en el cual son conocidas y por tanto accesibles. Así pues, se distinguen los siguientes tipos: variables locales y variables globales.

### 6.3.1 Variables Globales

Una variable global se halla declarada fuera de toda función del programa al inicio del programa principal. Su ámbito se extiende a lo largo de todas las funciones del fichero de código fuente. El “tiempo de vida” de una variable global está limitado por el tiempo que dura la ejecución del programa, por lo que se crean al comenzar a ejecutar el programa y se destruyen cuando éste concluye la ejecución.

En el siguiente ejemplo existe una variable global denominada “a” que es accesible desde todas las funciones, tanto el `main()` como la denominada `funcion()`. Nótese que estas funciones no tienen que declarar la variable, solo utilizarla. Cualquier cambio que hagan sobre ella lo reflejarán las otras funciones, ya que solo hay una variable común a todas las funciones.

```
#include <stdio.h>

int a=3; //variable global

void funcion();

void main() //main tiene acceso a "a"
{
    printf("A vale: %d\n",a);
    a=7;
    funcion();
    printf("A vale: %d\n",a);
}

void funcion()//tambien tiene acceso a "a"
{
    printf("A vale dentro de la funcion: %d\n",a);
    a=10;
}
```

La salida por pantalla de este programa, es por lo tanto:

```
A vale: 3
A vale dentro de la funcion: 7
A vale: 10
```

El uso de variables globales debe hacerse con precaución y evitarse siempre que sea posible. Es una buena práctica de programación no emplear variables globales salvo en casos muy excepcionales.

Cuando el código fuente queda repartido por varios ficheros diferentes y se quiere utilizar una única variable global a todos ellos debe de hacerse con la palabra clave `extern`, aunque su uso queda fuera del alcance de este libro.

### 6.3.2 Variables Locales

Una variable local se halla declarada, por lo general, al inicio del cuerpo de una función. Su ámbito se reduce al bloque de sentencias que componen el cuerpo de la función, por lo que sólo son conocidas dentro de él. Por otra parte, su “tiempo de vida”

va desde que se entra en la función hasta que se sale de ella, por lo que las variables locales se crean al comenzar la ejecución de la función y se destruyen al concluir dicha ejecución. Se denominan variables locales automáticas.

```
#include <stdio.h>

void funcion();

void main()
{
    int a=3; //"a" es local a main
    b=28; //Esto es un error de sintaxis, "b" no existe aqui
}

void funcion()
{
    int b=7; //"b" es local a la funcion
    a=18; //Esto es un error de sintaxis, "a" no existe aqui
}
```

Como se ve, no se puede acceder a las variables fuera de su ámbito. En el caso anterior resulta bastante evidente, dado que los nombres de las variables son diferentes. Sin embargo y dado que cada función tiene por así decirlo su propio espacio de memoria, dos o más funciones pueden tener una variable local con el mismo nombre sin que interfiera con las otras funciones. Cada función tiene su propio dato, y las modificaciones que haga sobre el no tienen ningún efecto sobre los otros, tal y como se muestra en el ejemplo siguiente:

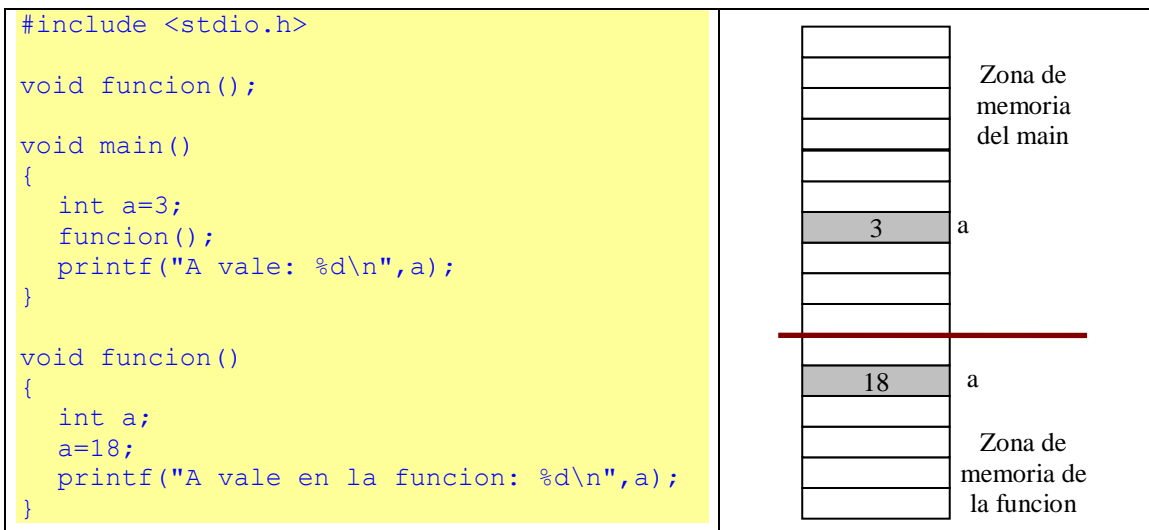


Figura 6-2- Variables locales

**A vale dentro de la funcion: 18**  
**A vale: 3**

## 6.4 Parámetros de una función

Cuando se realiza la llamada a una función que tiene parámetros, además de invocar su nombre, hay que escribir dentro de los paréntesis una lista de argumentos, tantos como parámetros tenga la función. Un argumento no es más que el valor que recibe el parámetro correspondiente cuando se llama a la función.

Por ejemplo, en el siguiente programa se ha programado una función con un único parámetro de tipo `float`. La función imprime un mensaje con dicho valor y el

doble de ese valor. Nótese que este es un ejemplo didáctico, no tiene una utilidad real importante.

```
#include <stdio.h>

void ImprimeDoble(float x);

void main()
{
    float a=1.5f;

    ImprimeDoble(3.0f); //llamada con una constante
    ImprimeDoble(a); //llamada con una variable
}

void ImprimeDoble(float x)
{
    printf("Doble = %f\n",2*x);
}
```

```
Doble = 6.000000
Doble = 3.000000
```

La función `ImprimeDoble()` tiene un parámetro “x” que es una variable local a la función, solo existe dentro de la función `ImprimeDoble()`. Cuando se realiza la llamada a la función, se copia el valor del argumento entre paréntesis al parámetro “x”. Por lo tanto, la primera vez que se llama la función, “x” toma el valor 3.0 y la segunda toma el valor 1.5.

Si el tipo de datos que se pasa como argumento no coincide con el tipo del parámetro correspondiente, se produce una conversión implícita de tipo en el caso de que sea posible. Por ejemplo la llamada siguiente (con el prototipo del ejemplo anterior) pasa como argumento una variable de tipo entero, mientras que la función espera una variable de tipo `float`. Lo que sucede es que el número entero se convierte implícitamente a `float` (el compilador emite un “warning” avisando de esta conversión), en este caso pasa de ser 3 a 3.000000, y el programa funciona sacando por pantalla 6.000000

```
int a=3;
ImprimeDoble(a);
```

También es posible lo contrario, que el parámetro de la función fuera de tipo entero, y se le pasara una variable de tipo `float` como argumento. En este caso la variable se convertiría implícitamente a entero, truncando sus decimales (el compilador también emitiría un “warning” avisando de esto).

```
#include <stdio.h>

void ImprimeDoble(int x);

void main()
{
    float a=1.5f;
    ImprimeDoble(a); // "a" se convierte a int, pasando a valer 1
}

void ImprimeDoble(int x) // "x" vale 1
{
    printf("Doble = %d\n",2*x);
}
```

```
Doble = 2
```



A esta forma de pasar los argumentos se le denomina paso por valor (o por copia) porque realmente a la función `ImprimeDoble()` no se le pasan la variable “a”, sino una copia de su valor, el cual es recibido por el parámetro “x”. Además del paso por valor, existe otra forma de pasar los argumentos a una función que se denomina paso por referencia. A continuación se describen más en profundidad ambos métodos.

### 6.4.1 Paso por valor

Este método copia el valor del argumento en el parámetro formal correspondiente. A la función se le pasa una copia de los argumentos pero no los propios argumentos, de manera que la función no puede modificar el valor de estos últimos.

En el ejemplo anterior, supongamos que el código de la función modifica el valor de su parámetro x, y que el `main()` saca por pantalla el valor de su variable “a” después de llamar a la función:

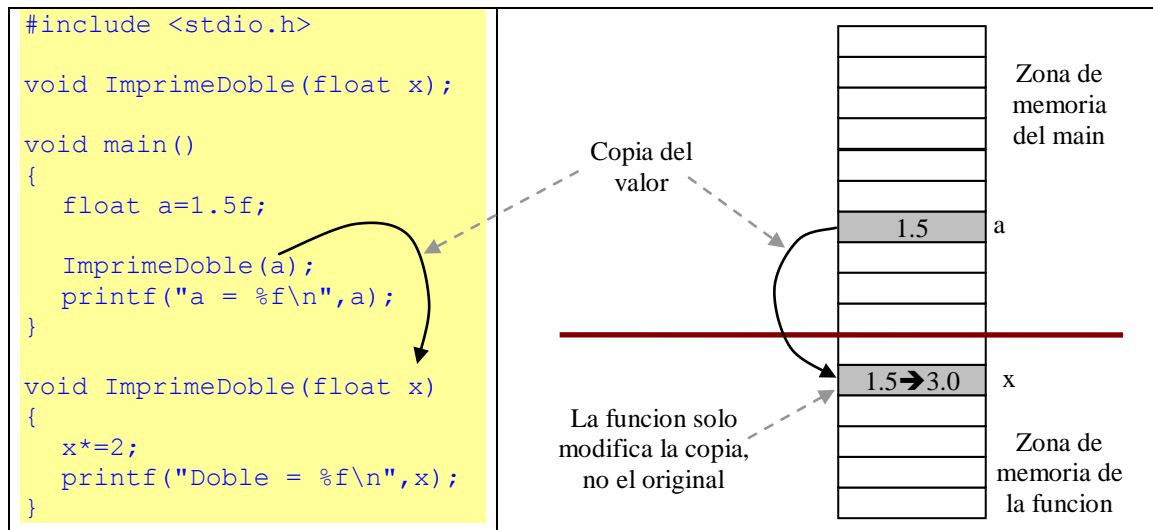


Figura 6-3. Paso de parámetro por valor

Como se muestra en la figura, el parámetro “x” es una copia de “a” y por lo tanto contiene inicialmente el valor 1.5. Cuando la función modifica la “x”, esta modificando solo la copia local del dato, pero la variable original “a” sigue sin modificar, por lo que la salida por pantalla será la siguiente:

```
Doble = 3.000000
a = 1.500000
```

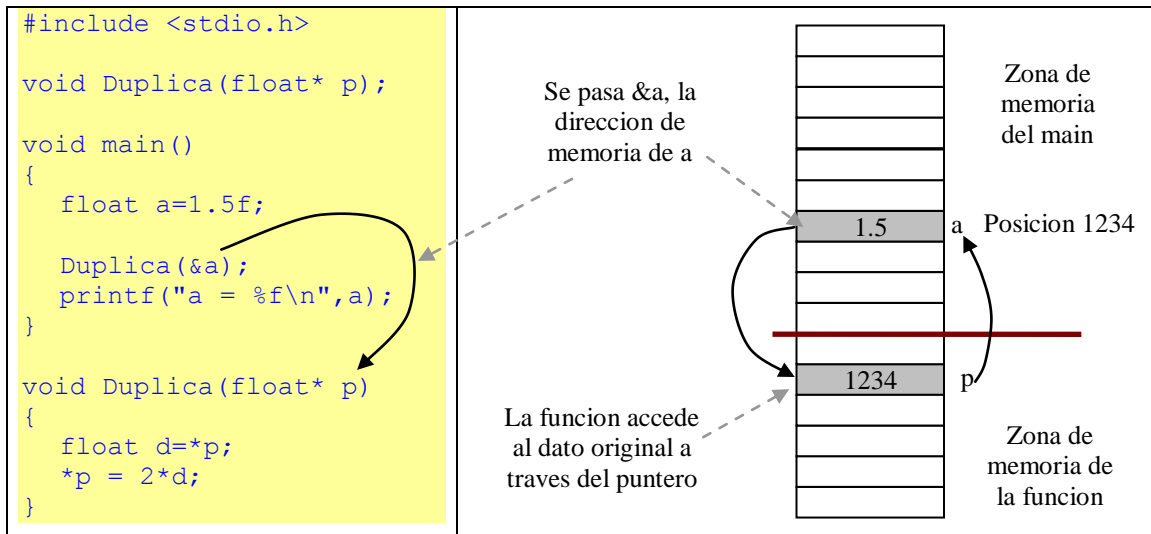
Esto sucede incluso si la variable local de la función `main()` se denominara “x” en lugar de “a”. Entonces se aplicaría el razonamiento visto en el apartado anterior, en el que existen dos variables denominadas “x”, cada una local a su función. Cuando `ImprimeDoble()` cambia el valor de “x” solo cambia el local, mientras que la variable “x” de la función `main()` seguiría manteniendo su valor de 1.5.

### 6.4.2 Paso por referencia

Este método copia la dirección de memoria del argumento en el parámetro formal correspondiente. Para poder recibir una dirección de memoria, el parámetro formal de la función tiene que ser un puntero. A la función se le pasa la dirección de

memoria de los argumentos, de manera que la función puede modificar su valor accediendo a ellos indirectamente a través de los punteros.

Supóngase que se quiere programar una función que duplique (multiplique por 2) el valor de una variable que se le pasa como parámetro, se haría de la siguiente forma:



**Figura 6-4. Paso de parámetro por referencia**

En el ejemplo anterior, cuando se pasa como parámetro &a, se está pasando y asignando al puntero “p” la dirección de memoria de “a”, en este caso el valor 1234 (por ejemplo). La función ahora dispone de un puntero que apunta a la variable original “a”. Cuando hace:

```
float d=*p; //equivalente a d=*(1234)=a=1.5
```

realmente obtiene el valor de “a”, en este caso 1.5. A continuación establece

```
*p = 2*d; //equivalente *(1234)=2*1.5, es decir duplica "a" del main
```

Lo que equivale a decir que el “a” del `main()` vale  $2 \times 1.5$ , es decir, esta duplicando el “a” del `main()`.

Puede verse que función `Duplica()` del programa anterior puede modificar el valor de “a” a pesar de que realmente no conoce a ninguna variable con ese nombre. Sin embargo, dispone de un puntero llamado “p” que recibe la dirección de memoria en la que está almacenada dicha variable (al llamar a la función es como si hiciéramos la operación `p=&a`), es decir “p” apunta a “a”. Por tanto, cualquier operación que hagamos con `*p` es como si la estuviéramos haciendo con “a”, por ejemplo, duplicarla.

En el siguiente ejemplo, se hace uso de una función llamada `cambio()` que intercambia el valor de dos variables de la función `main()`:

```
#include <stdio.h>

void cambio (int *p, int *q);

void main()
{
    int a=1, b=7;
    cambio(&a, &b);
    printf("a vale %d y b vale %d\n", a, b);
}
```

```
void cambio (int *p, int *q)
{
    int aux;

    aux=*p;
    *p=*q;
    *q=aux;
}
```

Cuya salida por pantalla sería:

```
a vale 7 y b vale 1
```

Nótese que este resultado de intercambiar los dos valores mediante una función NO es posible sin el uso del paso por referencia.

Hasta ahora hemos visto que una función puede devolver únicamente un valor, el que se especifica después de la palabra `return`. En un ejemplo anterior vimos que la función `AreaCirculo()` devolvía un valor de retorno. Pero, ¿y si quisiéramos hacer que una función haga varias operaciones y devuelva varios valores? Por ejemplo, que sume, multiplique y divida dos números. ¿Cómo recuperamos la suma, la multiplicación y la división calculadas por la función si sólo puede devolver un valor?

La solución consiste en pasarle a la función tantas variables por referencia como cálculos queramos recuperar. Siguiendo con el ejemplo, podemos declarar una función con el siguiente prototipo:

```
void operaciones (float x, float y, float *s, float *p, float *d);
```

En este prototipo hemos decidido que la función no devuelve ningún valor, aunque podíamos haberlo aprovechado para alguna de las tres operaciones. De los cinco parámetros que tiene la función, los dos primeros se pasan por valor (los operandos) y los tres últimos por referencia (las operaciones que queremos recuperar).

Un programa que hace uso de esta función se muestra a continuación:

```
#include <stdio.h>

void operaciones (float x, float y, float *s, float *p, float *d);

void main()
{
    float a=1.0f, b=2.0f, suma, producto, division;

    operaciones(a, b, &suma, &producto, &division);

    printf("La suma vale %f\n", suma);           //Imprime 3
    printf("El producto vale %f\n", producto);   //Imprime 2
    printf("La division vale %f\n", division);   //Imprime 0.5
}

void operaciones (float x, float y, float *s, float *p, float *d)
{
    *s=x+y;
    *p=x*y;
    *d=x/y;
}
```

Nótese que el paso por referencia ya ha sido utilizado con anterioridad sin entender realmente lo que se estaba haciendo. Es el caso de la función `scanf()`.

Cuando se quería que el usuario tecleara el valor de un `float` y asignarlo a una variable se hacía:

```
float x;
scanf("%f",&x);
```

Nótese que a la función `scanf()` para que pueda modificar la variable “x” y escribir en ella el valor tecleado, se le pasa la variable “x” por referencia, es decir un puntero. Sin embargo la función `printf()` recibía los parámetros por valor (copia) ya que no necesitaba cambiar su contenido, solo lo mostraba por pantalla.

## 6.5 Retorno de una función

### 6.5.1 Funciones que no devuelven resultados (tipo void)

Cuando una función finaliza su ejecución, devuelve el control a la parte del código desde donde se ha realizado la llamada. Cuando una función es de tipo `void`, es decir, no devuelve nada, su ejecución termina cuando llega a la llave de cierre de la función. Sin embargo, si se desea terminar la función por algún motivo antes de llegar a dicha llave, se puede utilizar la sentencia `return`. Esta sentencia finaliza la función en el momento que es ejecutada, volviendo el control a la línea siguiente de donde se realizó la llamada a la función. Véase el ejemplo siguiente, en el que se implementa una función que sirve para contar desde 0 hasta el número entero que se le pasa como parámetro:

```
#include <stdio.h>

void ContarHasta(int n);

void main()
{
    ContarHasta(-3);
    ContarHasta(5);
}

void ContarHasta(int n)
{
    int i;
    if(n<=0)
    {
        printf("Error, parametro negativo\n");
        return; //terminamos la funcion en este punto
    }
    for(i=0;i<=n;i++)
        printf("%d ",i);
    printf("\n");
    //return es opcional aqui
}
```

Si se introduce como parámetro un número negativo se informa al usuario y se termina la función, ya que no se quiere seguir ejecutando nada. Si el parámetro es positivo, la función termina cuando se llega a la llave de cierre. Se podría haber puesto también `return` justo antes de esta llave, pero si el tipo de la función es `void`, esto es opcional. La salida por pantalla sería:

```
Error, parametro negativo
0 1 2 3 4 5
```

### 6.5.2 Funciones que devuelven resultados

El uso de funciones con un tipo de retorno diferente a `void` es más común, ya que la devolución de resultados por una función es una característica muy útil que ayuda a la creación de software reutilizable. El caso más común de función es aquel que tiene unos datos (parámetros) de entrada, efectúa una operación con dichos parámetros y devuelve un valor como resultado. Siguiendo el ejemplo anterior, supóngase que lo que se desea es una función que calcule el doble de un número que se le pasa como parámetro, pero que no modifique dicho parámetro, sino que devuelva el doble por otro medio. Este medio es el tipo de retorno. Véase el código siguiente:

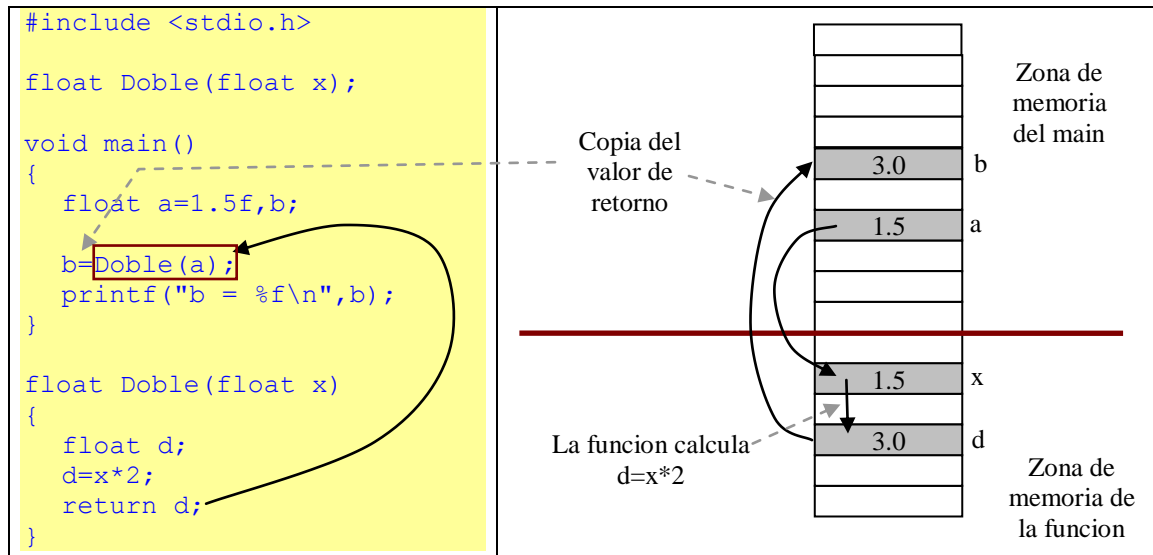


Figura 6-5. Retorno de una función

Como se ve en el ejemplo, la función `Doble()` calcula en una variable local llamada “d” el doble del parámetro “x”. Cuando la función realiza el “`return d;`” lo que se hace es asignar el valor de “d” a la variable que esta en el lado izquierdo de la igualdad `b=Doble(a);`, en este caso la variable “b”. Es como si la llamada a la función, es decir “`Doble(a)`” se sustituyera por el retorno, en este caso “d”.

El mismo resultado se podría haber obtenido con una función `Doble()` más compacta:

```

float Doble(float x)
{
    return x*2;
}
  
```

Además, también se puede compactar la función `main()` (aunque esto no quizás la mejor opción, ya que el programa pierde algo de legibilidad) de la siguiente manera, en la que también se muestra como se puede utilizar la función para calcular el cuádruple de la variable “a”.

```

void main()
{
    float a=1.5f;

    printf("Doble de %f es = %f\n", a, Doble(a));
    printf("Cuadruple de %f es = %f\n", a, Doble(Doble(a)));
}
  
```

En el programa anterior, el valor devuelto por la función `máximo` no se asigna a ninguna variable, sino que la llamada se realiza directamente dentro de la función `printf()`. Cuando la esta función encuentra el especificador de formato `%f` imprime el primer elemento de la lista que viene a continuación, que en este caso es el valor devuelto por la función. Lo mismo sucede para calcular el cuádruple. Primero se calcula el `Doble(a)`, y el resultado se introduce como parámetro otra vez a la función `Doble()` más externa.

Cuando una función tiene tipo de retorno, el uso de `return` es siempre obligatorio y siempre debe devolver un valor del tipo adecuado. Si el valor que se devuelve no es del tipo adecuado, se realiza una conversión implícita, caso de que esta sea posible. Si por ejemplo, la función es de tipo `float`, pero se devuelve un entero, este entero promocionara automáticamente a `float`. Si por el contrario la función es de tipo entero y se intenta devolver un número `float`, el compilador nos avisara con un “warning” de posible pérdida de datos, la correspondiente de truncar los decimales del número real que se devuelve.

Anteriormente hemos visto algunas funciones que no devuelven datos, sino que imprimen algunos resultados por pantalla con `printf()`. La utilidad de estas funciones es muy limitada, ya que no sirven más que para eso, pero no pueden ser utilizadas en programas más complejos. Por ejemplo, se puede hacer una función que calcule el máximo de dos números. Si hacemos que esta función sea de tipo `void` y no devuelva nada, su utilidad se limita a mostrar mensajes por pantalla, lo que es muy limitado. Es la solución peor. Mucho mejor opción es devolver como resultado de la función el valor máximo deseado, y que luego sea el usuario el que decida como utilizarlo, si quiere imprimirlo por pantalla o hacer más cálculos con el.

<pre>//Retornando valor, MEJOR #include &lt;stdio.h&gt; int maximo(int a, int b);  void main() {     int a, b, max;     printf("Introduce numeros: ");     scanf("%d %d",&amp;a, &amp;b);     max=maximo(a,b);     printf("El maximo es %d",max); }  int maximo(int a, int b) {     if(a&gt;b)         return a;     else         return b; }</pre>	<pre>//Sin retornar valor, PEOR #include &lt;stdio.h&gt; void maximo(int, int);  void main() {     int a, b;     printf("Introduce numeros: ");     scanf("%d %d",&amp;a, &amp;b);     maximo(a,b); }  void maximo(int a, int b) {     if(a&gt;b)         printf("El maximo es %d",a);     else         printf("El maximo es %d",b); }</pre>
---	--

Si se utiliza la forma de la izquierda y se desea calcular el máximo de cuatro números introducidos por el usuario, se puede realizar de la forma siguiente, mientras que utilizando la versión de la derecha, calcular el máximo de cuatro números no sería posible:

```
void main()
{
    int a,b,c,d,max;
    printf("Introduce 4 numeros: ");
```

```
scanf("%d %d %d %d",&a,&b,&c,&d);
max=maximo(maximo(a,b),maximo(c,d));
printf("El maximo es %d\n",max);
}
```

**Ejemplo:** función que suma dos números de tipo float que se pasan como parámetro

```
#include <stdio.h>

float suma (float x, float y);

void main()
{
    float a=1.0f, b=2.0f, s;

    s=suma(a,b);    // Llamada a la función y asignación del valor
                   // devuelto por ella a la variable s.

    printf("La suma vale %f\n", s);
}

float suma (float x, float y)
{
    return (x+y);
}
```

La función suma tiene dos parámetros, x e y, que son dos variables locales de esta función. Cuando se realiza la llamada a la función se copia el valor de los argumentos, variables a y b, en el parámetro correspondiente. Es decir, x toma el valor de a (x=1.0) e y toma el valor de b (y=2.0).

El hecho de que el valor de retorno tenga la misma nomenclatura que la variable sobre la que se asigna no tiene ningún inconveniente. No obstante, hay que recordar que el hecho de que las variables se llamen igual no tiene ningún efecto si no se realiza la asignación correspondiente haciendo la llamada a la función adecuadamente:

```
#include <stdio.h>

int cuadrado(int x);

void main()
{
    int x=4, y;

    y=cuadrado(x); //la "y" del main coge su valor del retorno

    printf("x vale %d\n", x);
    printf("El cuadrado de x vale %d\n", y);
}

int cuadrado(int x)
{
    int y;
    y=128; //no afecta a la "y" del main, ya que esta "y" es otra
    x=x*x;
    return x;
}
```

Cuya salida por pantalla es:

```
x vale 4
El cuadrado de x vale 16
```

Como puede apreciarse, el valor de la variable local “x” de la función `main()` no cambia de valor después de llamar a la función porque el parámetro formal de la función `cuadrado()`, que también se llama “x”, es una variable local de dicha función y sólo recibe de la otra “x” una copia, es decir, el valor 4. Por tanto, cualquier operación que se realice dentro de la función no afecta por al valor de la variable “x” de la función `main()` que se pasa como argumento en la llamada. Lo mismo sucede con la variable “y”. La función `cuadrado()` tiene una variable local llamada “y” a la que asigna el valor 128. Sin embargo, esta variable no tiene nada que ver con la variable “y” de la función `main()` que coge su valor del retorno de la función, en este caso “x” al cuadrado que vale 16. El uso de la variable “y” en la función no tiene ningún sentido práctico en este ejemplo y se puede suprimir, se ha usado únicamente por motivos didácticos.

### 6.5.3 Funciones con retorno booleano

Es muy común el uso de funciones que devuelven un número entero, cuyos valores son exclusivamente 0 (falso) o 1 (verdadero). En este caso se pueden utilizar estas funciones directamente como proposiciones lógicas. En el siguiente ejemplo se muestra una función que admite un parámetro de tipo entero y devuelve un 0 si el número no es impar y un 1 si el número es impar. Denominamos a la función `EsImpar()`, que realmente es la tarea que esta realizando:

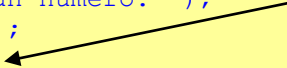
```
#include <stdio.h>

int EsImpar(int n);

void main()
{
    int num;
    printf("Teclee un numero: ");
    scanf("%d",&num);
    if(EsImpar(num))
        printf("Su numero es impar\n");
    else
        printf("Su numero no es impar\n");
}

int EsImpar(int n)
{
    if(n%2==1)
        return 1; //impar
    else
        return 0; //par
}
```

```
//NOTESE el uso de la funcion
//equivalente a
int ret=EsImpar(num);
if(ret==1)
{
}
```



Obsérvese como se utiliza la función de forma compacta, y que se lee muy intuitivamente como “Si el número `num` es impar”.

Otra tarea típica de los programas que es preguntar al usuario si desea continuar la ejecución del mismo, se puede hacer mediante una función, que se encarga de todo, preguntar al usuario, mostrar el menú, repetir la pregunta si la respuesta del usuario no es adecuada. Por otro lado, lo que se consigue es un `main()` muy sencillo e intuitivo:

```
#include <stdio.h>

int QuiereContinuar();
```



```

void main()
{
    while(QuiereContinuar())
    {
        //tarefas a hacer
    }
}

int QuiereContinuar()
{
    int resp;
    while(1)
    {
        printf("Quiere continuar?\n 0-No\n 1-Si\n");
        scanf("%d",&resp);
        if(resp==1)
            return 1;
        if(resp==0)
            return 0;
        printf("Respuesta no valida\n");
    }
}

```

## 6.6 Funciones que llaman a otras funciones

Considere el siguiente ejemplo en el que se realiza un programa para calcular las combinaciones de “n” elementos tomados de “k”, o lo que es lo mismo el número combinatorio “n sobre k”:

$$\binom{n}{k} = \frac{n!}{(n-k)! k!}$$

Para calcular dicho valor se puede realizar en la función `main()` como sigue:

```

#include <stdio.h>

int factorial(int a);

void main()
{
    int n, k, i;
    int factN=1, factNK=1, factK=1;
    int n_sobre_k;

    printf("Introduzca n y k: ");
    scanf("%d %d", &n, &k);

    for(i=1; i<=n; i++)
        factN*= i;

    for(i=1; i<=(n-k); i++)
        factNK*= i;

    for(i=1; i<=k; i++)
        factK*= i;

    n_sobre_k = factN/(factNK*factK);

    printf("%d sobre %d = %d\n", n, k, n_sobre_k);
}

```

Como puede observarse, en el programa anterior se utilizan tres bucles `for` para calcular el factorial de tres números, lo que significa repetir el código tres veces. Esto se puede hacer de una forma mucho mejor con una función. A continuación, se creará una función llamada `factorial()` que calculará el factorial del número que reciba como parámetro:

```
#include <stdio.h>

int factorial(int a);

void main()
{
    int n, k;
    int n_sobre_k;

    printf("Introduzca n y k: ");
    scanf("%d %d", &n,&k);

    n_sobre_k = factorial(n)/(factorial(n-k)*factorial(k));

    printf("%d sobre %d = %d\n", n, k, n_sobre_k);
}

int factorial(int a)
{
    int i, res=1;
    for(i=1; i<=a; i++)
        res*=i;
    return res;
}
```

La función `factorial()` es llamada tres veces desde la función `main()`, en la primera se le pasa como parámetro el valor correspondiente. Supóngase ahora que se quiere utilizar dicho número combinatorio para otro calculo, por ejemplo elevar un binomio a una potencia (binomio de Newton). Con este objetivo resulta evidente la utilidad de crear una función específica para este cálculo:

```
#include <stdio.h>

int factorial(int a);
int combinaciones(int n,int k);

void main()
{
    int n, k;
    int n_sobre_k;

    printf("Introduzca n y k: ");
    scanf("%d %d", &n,&k);

    n_sobre_k = combinaciones(n,k);

    printf("%d sobre %d = %d\n", n, k, n_sobre_k);
}

int combinaciones(int n,int k)
{
    int result;
    result = factorial(n)/(factorial(n-k)*factorial(k));
    return result;
}

//funcion factorial va aqui, se ha omitido por brevedad
```

Nótese como la función `combinaciones()` hace uso de la función `factorial()` para realizar sus cálculos. Rematamos este ejemplo con una función más que imprime el triángulo de pascal (con un determinado número de filas, que se le pasa como parámetro), haciendo uso a su vez de la función `combinaciones()`. Nota: Este programa no funciona para números muy grandes, ya que el cálculo del factorial está limitado a tipo `int`, lo que es rápidamente superado para factoriales de números mayores que el 12. Se podría ampliar este rango mediante variables de tipo doble, aunque se corre el riesgo de realizar aproximaciones, ya que aunque el rango de estas variables es mayor, el número de sus cifras significativas tampoco es infinito.

```
#include <stdio.h>

int factorial(int a);
int combinaciones(int n,int k);
void triangulo_pascal(int n);

void main()
{
    triangulo_pascal(10);
}
void triangulo_pascal(int n)
{
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<=i;j++)
            printf("%d ",combinaciones(i,j));
        printf("\n");
    }
}
//cuerpo de las funciones factorial y combinaciones aqui
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

## 6.7 Funciones y vectores

En C, cualquier matriz y en particular un vector siempre se pasa por referencia. Es decir, cuando un vector se pasa como argumento a una función no se pasa el vector completo, sino su dirección de memoria (más concretamente, la dirección de memoria de su primer elemento). Además, la función puede modificar el contenido de los elementos del vector ya que conoce la dirección de memoria donde están almacenados.

Conviene recordar que el nombre de un vector es un puntero que apunta a su primer elemento. Por ejemplo, si hacemos la siguiente declaración:

```
int vector[3]={10,20,30};
```

Se cumple que `vector` y `&vector[0]` tienen el mismo valor, aunque habitualmente los programadores utilizan el primero.

Si queremos declarar una función que pueda recibir un vector como argumento podemos declarar el parámetro formal de tres formas: como un vector sin dimensión, un vector con dimensión o un puntero. Por ejemplo, los tres prototipos siguientes son equivalentes, aunque lo más común es utilizar el primero o el tercero:

```
void funcion (int vector[]);
void funcion (int vector[3]);
void funcion (int *p);
```

El segundo prototipo no suele usarse ya que, como se ha contado anteriormente, lo que se recibe es una dirección de memoria y no un vector completo. De hecho, al compilador le da exactamente igual el número que se ponga entre los corchetes y en cualquiera de los tres prototipos la función se prepara para recibir una dirección de memoria. Si desea que la función conozca la dimensión del vector que se le pasa como argumento hay que añadir un parámetro a la función que reciba ese dato. Por ejemplo:

```
void funcion (int vector[], int dimension);
```

Los programas siguientes imprimen los elementos de un vector utilizando la misma función escrita con subíndices y con punteros:

```
// Con subíndices
#include <stdio.h>

void imprime(int v[], int n);

void main()
{
    int v1[3]={10,20,30};

    imprime(v1,3);
}

void imprime(int v[], int n)
{
    int i;

    for(i=0;i<n;i++)
        printf("%d\n",v[i]);
}
```

```
// Con punteros
#include <stdio.h>

void imprime(int *p, int n);

void main()
{
    int v1[3]={10,20,30};

    imprime(v1,3);
}

void imprime(int *p, int n)
{
    int i;

    for(i=0;i<n;i++)
        printf("%d\n",*(p+i));
}
```

Los programas anteriores funcionarían igual incluso utilizando la notación de punteros en el prototipo y en la cabecera de la función y la notación de subíndices en el cuerpo de la función, o viceversa.

El programa siguiente hace uso de una función que convierte un vector que se le pasa como argumento a valor absoluto:

```
#include <stdio.h>

void abs(int vector[], int n);

void main()
{
    int datos[5]={-1,3,-5,7,-9};
    int i;

    abs(datos,5);

    // Imprimimos el vector
}
```

```

    for(i=0;i<5;i++)
        printf("%d ",datos[i]);
    printf("\n");
}
void abs(int vector[], int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(vector[i]<0)
            vector[i]=-vector[i];
    }
}

```

**1 3 5 7 9**

El programa siguiente utiliza dos funciones, una que calcula el producto escalar de dos vectores y otra que calcula el mínimo.

```

#include <stdio.h>

float ProductoEscalar(float v1[], float v2[], int n);
float Minimo(float *p, int n); //es lo mismo que (float p[],int n)

void main()
{
    float v1[5]={1,2,3,-4,5};
    float v2[5]={-6,-7,8,9,10};
    float producto_escalar, min1, min2;

    producto_escalar=ProductoEscalar(v1,v2,5);
    printf("El producto escalar vale %f\n", producto_escalar);

    min1=Minimo(v1,5);
    min2=Minimo(v2,5);

    printf("El minimo de v1 vale %f\n", min1);
    printf("El minimo de v2 vale %f\n", min2);
}

float ProductoEscalar(float v1[], float v2[], int n)
{
    int i;
    float prod=0;

    for(i=0;i<n;i++)
    {
        prod+=v1[i]*v2[i];
    }
    return prod;
}

float Minimo(float *p, int n)
{
    float min=p[0];
    int i;

    for(i=0;i<n;i++)
    {
        if(p[i]<min)
            min=p[i];
    }
    return min;
}

```

A continuación se muestra un ejemplo en el que una función calcula la media de los elementos de un vector entero:

```
#include <stdio.h>

#define N 4

float media(int vec[], int n);

void main()
{
    float med;
    int i, v[N];

    printf( "Introducir elementos del vector\n" );
    for (i=0; i<N; i++)
        scanf( "%d", &v[i]);

    med = media(v,N);
    printf( "La media es: %f", med );
}

float media(int vec[], int n)
{
    int j;
    float sum=0.0;
    for (j=0; j<n; j++)
        sum = sum + vec[j];
    return ((float)sum/n);
}
```

## 6.8 Funciones y matrices

En C, una matriz siempre se pasa por referencia. Cuando una matriz se pasa como argumento a una función realmente no se pasa la matriz completa sino la dirección del primer elemento de dicha matriz.

Al igual que ocurre con los vectores, el nombre de una matriz es un puntero que apunta a su primer elemento. Por ejemplo, en la siguiente declaración:

```
//declaracion e inicializacion de matriz
int tabla[2][2]={
    {1,2},
    {3,4}
};
```

Los valores de `tabla` y `&tabla[0][0]` son iguales. Si se quiere declarar una función que pueda recibir una matriz bidimensional como argumento podemos declarar el parámetro formal como una matriz en la que se puede omitir la primera dimensión (número de filas) pero no la segunda (número de columnas). Por ejemplo, los dos prototipos siguientes son equivalentes:

```
void funcion (int matriz[][3]);
void funcion (int matriz[3][3]);
```

En este caso, puede seguirse el criterio de poner siempre todas las dimensiones de la matriz sabiendo que la primera es innecesaria.

El siguiente programa imprime el elemento seleccionado de una matriz utilizando una función:

```
#include <stdio.h>

void imprime_matriz(int matriz[2][2], int filas, int columnas);

main()
{
    int tabla[2][2]={1,2}, {3,4}};

    imprime_matriz(tabla, 2, 2);
}

void imprime_matriz(int matriz[2][2], int filas, int columnas)
{
    int i, j;

    for(i=0;i<filas;i++)
    {
        for(j=0;j<columnas;j++)
            printf("%d ", matriz[i][j]);

        //Salta una línea después de imprimir cada fila
        printf("\n");
    }
}
```

Cuya salida por pantalla sería:

```
1 2
3 4
```

## 6.9 Funciones y cadenas

En C, una cadena de caracteres siempre se pasa por referencia. Todo lo dicho para vectores es válido para cadenas de caracteres teniendo en cuenta que estas últimas están formadas por caracteres individuales en lugar de números y que terminan con el caracter nulo ('`\0`'). No obstante, se repiten a continuación como se pasa una cadena como argumento a una función.

Cuando una cadena de caracteres se pasa como argumento a una función se pasa la dirección de memoria de su primer elemento, la cual suele especificarse con el nombre de la cadena.

Por ejemplo, si hacemos la siguiente declaración:

```
char saludo[]="Hola";
```

El nombre de la cadena, saludo, es un puntero que apunta al primer elemento de la cadena, es decir, su valor coincide con `&saludo[0]`.

Si queremos declarar una función que pueda recibir una cadena de caracteres como argumento podemos declarar el parámetro formal de tres formas: como una cadena con dimensión, como una cadena sin dimensión o como un puntero. Sin embargo, lo habitual es utilizar una de las dos últimas opciones puesto que el compilador ignora el número escrito entre corchetes y prepara a la función para recibir una dirección de memoria y no una cadena de caracteres con una determinada dimensión. Por ejemplo, los siguientes prototipos son equivalentes:

```
void funcion (char cadena[]);
void funcion (char *p);
```

En el caso de las cadenas de caracteres no es necesario pasarle a la función la dimensión de la cadena, ya que se manejan de manera distinta a los vectores. El recorrido secuencial por los elementos de la cadena se realiza pasando por cada uno de ellos hasta encontrar el caracter nulo que indica el final de la cadena. Los programas siguientes imprimen los elementos de una cadena, caracter a caracter, utilizando la misma función escrita con subíndices y con punteros:

```
#include <stdio.h>

void imprime(char cadena[]);

void main()
{
    char saludo[]="Hola";
    imprime(saludo);
}

void imprime(char cadena[])
{
    int i=0;
    while(cadena[i]!='\0')
    {
        printf("%c", cadena[i]);
        i++;
    }
    printf("\n");
}
```

```
#include <stdio.h>

void imprime(char *p);

void main()
{
    char saludo[]="Hola";
    imprime(saludo);
}

void imprime(char *p)
{
    int i=0;
    while(*(p+i]!='\0')
    {
        printf("%c", *(p+i));
        i++;
    }
    printf("\n");
}
```

Como ocurría con los vectores también pueden mezclarse las notaciones. Por ejemplo el prototipo y la cabecera pueden escribirse con punteros y en el cuerpo de la función utilizar subíndices, o viceversa.

El programa siguiente hace uso de una función llamada `longcad()` que devuelve la longitud de una cadena de caracteres, sin contar el caracter nulo:

```
#include <stdio.h>

int longcad (char *cadena);

void main()
{
    char mensaje[]="Hola";
    printf("%s tiene %d caracteres\n", mensaje, longcad(mensaje));
}

int longcad (char *cadena)
{
    int n=0;

    while (*cadena!='\0')
    {
        n++;
        cadena++; //El puntero cadena apunta al siguiente elemento
    }
    return n;
}
```

La función se podría haber programado igualmente, sin necesidad de modificar el puntero que se pasa como parámetro, de la siguiente forma:



```
int longcad (char cadena[])
{
    int n=0;
    while (cadena[n]!='\0')
    {
        n++;
    }
    return n;
}
```

Cuya salida por pantalla sería:

**Hola tiene 4 caracteres**

Ejemplo: Función que devuelve el número de veces que aparece un caracter.

```
#include <stdio.h>
#define N 50

int cuenta(char cad[], char c);

void main()
{
    char palabra[N]="Informatica", c='a';
    int b;

    b=cuenta(palabra,c);
    printf("En la palabra %s hay %d letras %c\n", palabra, b, c);
}

int cuenta(char cad[], char c)
{
    int i=0, cont=0;
    while(cad[i]!='\0')
    {
        if(cad[i]==c)
            cont++;
        i++;
    }
    return cont;
}
```

## 6.10 Funciones y estructuras

Como se describió en un capítulo anterior, cuando se ha definido una estructura de datos, esta puede ser utilizada de forma análoga a como son utilizados los tipos básicos de datos. Sucede exactamente igual con las funciones, las estructuras pueden ser pasadas como parámetros a las funciones (tanto por valor como por referencia) y pueden ser devueltas como retorno de una función. Este apartado trata ambos casos

### 6.10.1 Estructuras como parámetros de una función

#### 6.10.1.1 Paso por valor

Una estructura puede ser pasada como parámetro a una función. Para ilustrar este concepto se propone el siguiente ejemplo, en el que se utiliza una estructura para almacenar la parte real e imaginaria de un número complejo y se le pasa a una función encargada de imprimir dicho número complejo. Nótese que este paso se realiza por valor, es decir realmente se imprime una copia de dicho complejo.

```
#include <stdio.h>

struct complejo
{
    float real;
    float imaginaria;
};

void Imprimir(struct complejo c);

void main()
{
    struct complejo comp={1, -3}; //inicializacion
    Imprimir(comp);
}

void Imprimir(struct complejo c)
{
    //el + antes de la 2ª f sirve para que ponga el signo
    printf("%f %f i\n",c.real,c.imaginaria);
}
```

**1.000000 + 3.000000i**

En este dominio, empieza a ser evidente la utilidad de usar tipos predefinidos por el usuario (`typedef`), de tal forma que el código anterior puede ser reescrito de forma más compacta como:

```
#include <stdio.h>

typedef struct complejo
{
    float real;
    float imaginaria;
} complejo;

void Imprimir(complejo c);

void main()
{
    complejo comp={1, 3};
    Imprimir(comp);
}

void Imprimir(complejo c)
{
    //el + antes del 2º %f sirve para que ponga el signo
    printf("%f %f i\n",c.real,c.imaginaria);
}
```

También es posible que una función que acepte un parámetro tipo estructura, devuelva un valor de retorno de un tipo básico. En el siguiente ejemplo se presenta una función que sirve para calcular el modulo de un número imaginario. Se han suprimido el cuerpo de la función anterior, sustituyéndolo por comentarios. Obviamente estas partes de código deben de ser incluidas en el programa real.

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

typedef struct complejo
{
    float real;
    float imaginaria;
} complejo;
```

```

void Imprimir(complejo c);
float Modulo(complejo c);

void main()
{
    complejo comp={1, 3};
    float mod;
    mod=Modulo(comp);
    printf("El modulo es: %f\n",mod);
}
float Modulo(complejo c)
{
    float m;
    m=sqrt(c.real*c.real+c.imaginaria*c.imaginaria);
    return m;
}
//cuerpo de las otras funciones

```

### 6.10.1.2 Paso por referencia

Imagínese ahora que se quiere programar una función que solicite los datos de un número complejo al usuario para que este los teclee por pantalla. Si a dicha función le pasamos por valor (copia) una estructura, cuando el usuario teclee los datos, estos no se almacenaran correctamente. Lo que hacemos sin embargo, tal y como se muestra en el código siguiente es pasar el parámetro por referencia para que la función pueda “escribir” en el:

```

#include <stdio.h>
#include <math.h>

typedef struct complejo
{
    float real;
    float imaginaria;
} complejo;

void Imprimir(complejo c);
void Pedir(complejo* c);
float Modulo(complejo c);

void main()
{
    complejo comp;
    Pedir(&comp); //notese el &
    Imprimir(comp);
}
void Pedir(complejo* c)
{
    float re,im;
    printf("Introduzca parte real e imag: ");
    // tb valdria scanf("%f %f",&c->real,&c->imaginaria);
    scanf("%f %f",&re,&im);
    c->real=re;
    c->imaginaria=im;
}
//el cuerpo de las otras funciones

```

### 6.10.2 Estructura como retorno de una función

Como se vio en un capítulo anterior, las estructuras admiten realizar asignaciones (con el signo =) de forma simple, algo que no admitían ni los vectores ni las cadenas de caracteres. Esto proporciona una interesante utilidad a la hora de manejar funciones, ya que estas admiten como tipo de retorno una estructura, algo que no sucedía ni con vectores ni con cadenas que podían devolver como máximo un puntero al vector, pero nunca una copia.

Siguiendo con el ejemplo anterior, se desea programar ahora una función que permita calcular el número complejo resultante de la multiplicación de otros dos complejos, que se pasan como parámetros.

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

typedef struct complejo
{
    float real;
    float imaginaria;
} complejo;

void Imprimir(complejo c);
void Pedir(complejo* c);
float Modulo(complejo c);
complejo Multiplica(complejo a,complejo b);

void main()
{
    complejo c1,c2,c3;
    Pedir(&c1);
    Pedir(&c2);
    c3=Multiplica(c1,c2);
    Imprimir(c3);
}
complejo Multiplica(complejo a,complejo b)
{
    complejo res;
    res.real=a.real*b.real-a.imaginaria*b.imaginaria;
    res.imaginaria=a.real*b.imaginaria+a.imaginaria*b.real;
    return res;
}
//aquí las otras funciones
```

```
Introduzca parte real e imag: 1 2
Introduzca parte real e imag: 2 3
-4.000000 +7.000000 i
```

Como ultimo ejemplo de este apartado, se muestra como una función que maneja estructuras puede hacer uso de otra función. Se completa el ejemplo anterior con una función que devuelva un complejo unitario (de modulo unidad) paralelo a un complejo que se le pasa como parámetro:

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

typedef struct complejo
{
    float real;
    float imaginaria;
} complejo;
```

```

void Imprimir(complejo c);
void Pedir(complejo* c);
float Modulo(complejo c);
complejo Multiplica(complejo a,complejo b);
complejo Unitario(complejo c);

void main()
{
    complejo c1,c2;
    Pedir(&c1);
    c2=Unitario(c1);
    Imprimir(c2);
}
complejo Unitario(complejo c)
{
    complejo res;
    float m=Modulo(c);
    res.real=c.real/m;
    res.imaginaria=c.imaginaria/m;
    return res;
}
//cuerpo de las otras funciones

```

```

Introduzca parte real e imag: 1 2
0.447214 +0.894427 i

```

### 6.11 Estructuración de funciones en ficheros

En el ejemplo anterior se ha presentado el desarrollo de varias funciones relacionadas con los números complejos. Como parece evidente, el potencial para reutilizar estas funciones en diferentes programas es muy alto (para programas relacionados con matemáticas, obviamente). Para conseguir esto de forma fácil, es conveniente estructurar y separar el código en distintos archivos de código. Aunque el resto del libro sigue manejando los programas en un único archivo, se considera relevante al menos apuntar este modo de uso, que es el que realmente se utiliza en el desarrollo de programas reales.

Se introducen 2 nuevos ficheros, un fichero de cabecera que denominaremos “complejos.h” y que contiene únicamente los prototipos y otro fichero de código fuente llamado “complejos.c” que contendrá el cuerpo de las funciones. Estos dos ficheros son los que constituyen nuestra pseudo-librería de manejo de complejos y que nos podemos llevar (copiar, por ejemplo) a otra aplicación. El fichero que contiene el `main()` simplemente tiene que hacer un `#include` al fichero de cabecera “complejos.h”, y al compilar se debe incluir en el proyecto o makefile el fichero “complejos.c”.

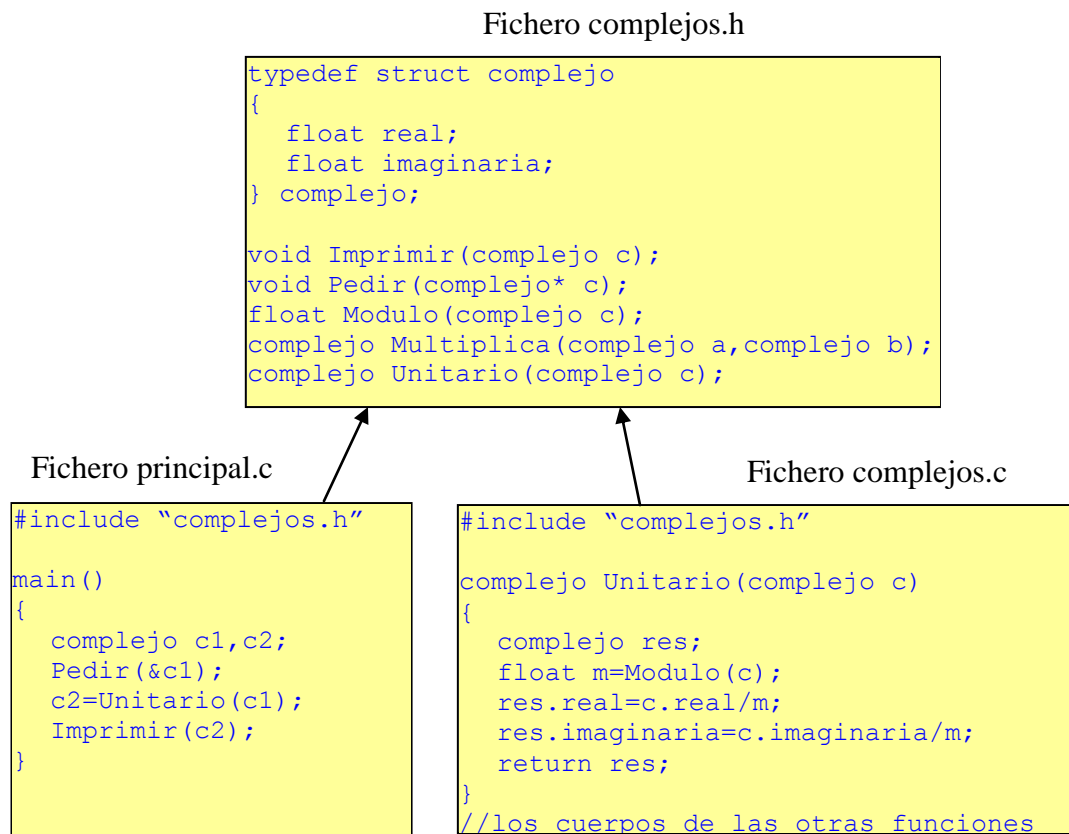


Figura 6-6. Estructuración de código en distintos ficheros

## 6.12 Recursividad

Se llama recursividad al proceso por el cual una función se llama a sí misma repetidamente hasta que se satisface una cierta condición. Normalmente este proceso se emplea para cálculos repetitivos en los que el resultado de cada iteración se determina a partir del resultado de alguna iteración anterior.

Sin embargo, de existir una solución no recursiva debe preferirse a una recursiva. No obstante, la solución no recursiva puede derivar en un programa extraordinariamente extenso por lo cual la solución apropiada una solución recursiva.

Uno de los ejemplos más comunes de recursividad es el cálculo del factorial. Si se analiza, el calculo de  $n!$  resulta igual al calculo de  $n * (n-1)!$ . Aplicando este razonamiento hasta llegar por ejemplo al factorial conocido de  $1!=1$ , se puede implementar recursivamente la función factorial:

```
#include <stdio.h>

int factorial(int num);

void main()
{
    int i;
    for (i = 0; i <= 10; i++)
        printf("%d = %d\n", i, factorial(i));
}
```

```
int factorial(int num)
{
    if (num <= 1)
        return 1;
    else
        return num*factorial(num-1); //recursividad
}
```

```
0 = 1
1 = 1
2 = 2
3 = 6
4 = 24
5 = 120
6 = 720
7 = 5040
8 = 40320
9 = 362880
10 = 3628800
```

En el siguiente ejemplo se implementa una función que permite calcular el término n-esimo de la sucesión de Fibonnaci, en la que cada término es la suma de los dos anteriores. Nótese que esta solución es adecuada para calcular uno de los términos, pero no es la más eficiente para calcular la serie completa de términos:

```
#include <stdio.h>

int fib(int n)
{
    if ((n == 0) || (n == 1))
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

main()
{
    int n,f;
    while(1)
    {
        printf("Que termino quiere: ");
        scanf("%d",&n);
        f=fib(n);
        printf("termino %d es %d\n",n,f);
    }
}
```

```
Que termino quiere: 1
El termino 1 es 1
Que termino quiere: 2
El termino 2 es 2
Que termino quiere: 3
El termino 3 es 3
Que termino quiere: 4
El termino 4 es 5
Que termino quiere: 5
El termino 5 es 8
Que termino quiere: 6
El termino 6 es 13
```

## 6.13 Parámetros de la función main()

En todos los ejemplos vistos hasta el momento, la función `main()` no recibe ningún parámetro. No obstante, es posible pasarle parámetros que el usuario introduce desde la línea de comando del sistema operativo. Los parámetros de la función `main()` son dos y se les conocen con el nombre de `argc` y `argv`.

El parámetro `argc` es de tipo entero (`int argc`) y contiene el número de parámetros que el usuario introducirá desde la línea de comandos. El nombre del programa se considera como el primer parámetro, por lo que al menos `argc` vale uno.

El parámetro `argv` es un vector de punteros a cadenas de caracteres (`char *argv[]`). Cada elemento de la cadena contiene los parámetros que el programa necesita para su ejecución y deben ser introducidos uno a uno separados mediante espacio o tabulador. Por tanto, el formato de la función `main()` con parámetros será:

```
#include <stdio.h>
void main( int argc, char *argv[] )
{
    int i;
    for(i=0;i<argc;i++)
    {
        printf("Argumento %d es: %s\n",i,argv[i]);
    }
}
```

Si se compila en un ejecutable denominado por ejemplo `parametros.exe`, la ejecución en línea de comandos sería:

```
C:\...\>parametros.exe Hola 1 12.3 Adios
Existen 5 argumentos
Argumento 0 es: parametros.exe
Argumento 1 es: Hola
Argumento 2 es: 1
Argumento 3 es: 12.3
Argumento 4 es: Adios
```

Nótese bien que los parámetros introducidos son tratados siempre como cadenas de texto, incluso cuando son números. Si es el usuario el que quiere tratar uno de los parámetros como número, debe convertirlo adecuadamente utilizando funciones del tipo `sscanf()`, `atoi()` o `atof()`.

En el siguiente ejemplo, se realiza un programa para que el usuario introduzca su nombre y edad desde la línea de comandos para mostrar un mensaje de saludo.

```
#include <stdio.h>
void main( int argc, char *argv[] )
{
    int edad;
    char nombre[50];
    if(argc!=3)
    {
        printf("Error parametros\n");
        return;
    }
    strcpy(nombre,argv[1]); //copia del 1 parametro
    sscanf(argv[2],"%d",&edad); //traduccion del 2º par a entero
    printf("Hola %s, tienes %d años\n",nombre,edad);
}
```



Si suponemos que el programa ejecutable se llama saludo.exe, en la línea de comando se tendrá lo siguiente:

```
c:\.....\ >saludo Pepe 23
Hola Pepe, tienes 23 años
```

Si alguno de los parámetros que se pasan al programa contiene espacios en blanco o tabuladores, deben usarse comillas (“”) en la ejecución del programa.

```
c:\.....\ >saludo "Pepe Perez" 23
Hola Pepe Perez, tienes 23 años
```

## 6.14 Ejercicios resueltos

**Ej. 6.1) Escribir la salida por pantalla de los siguientes programas:**

<pre>#include&lt;stdio.h&gt;  void f1(void); void f2(void);  void main() {     f1();     f2(); }  void f1(void) {     printf("1 ");     f2(); }  void f2(void) {     printf("2 "); }</pre>	<pre>#include&lt;stdio.h&gt;  void f1(int dato);  void main() {     int dato=3;     f1(dato);     printf("Dato= %d\n",dato); }  void f1(int dato) {     dato=dato*2; }</pre>
<p><b>1 2 2</b></p>	<p><b>Dato=3</b></p>
<pre>#include&lt;stdio.h&gt;  int f(int* p);  void main() {     int v[3]={1,12,123};     int a;     a=f(v);     printf("%d\n",a+1); }  int f(int* p) {     return *(p+1); }</pre>	<pre>#include&lt;stdio.h&gt;  typedef struct persona {     int edad;     float peso; }persona;  void f(persona* p);  void main() {     persona per={18,55.5f};     f(&amp;per);     printf("%d %f\n",per.edad,per.peso); }  void f(persona* p) {     p-&gt;edad=p-&gt;edad+1;     p-&gt;peso=p-&gt;peso+3; }</pre>
<p><b>13</b></p>	<p><b>19 58.50000</b></p>

**Ej. 6.2) Realice un programa que calcule la función exponencial de un número mediante la serie de Taylor:**

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Para ello el programa preguntará al usuario el número “x” con el que desea realizar el cálculo de la función exponencial y luego se realizará mediante la suma de los 10 primeros elementos de la serie, codificando las funciones `exponencial()`, `factorial()` y `potencia()`. El esqueleto del programa deberá ser el siguiente:

```
#include<stdio.h>
#define PRECISION 10

double factorial(int valor);
double potencia(double base,int expo);
void main()
{
    double x,e_x;
    printf("Numero: ");
    scanf("%lf",&x);
    e_x=exponencial(x);
    printf("la exp.de %lf es %lf\n",x,e_x);
}
```

**SOLUCION:**

```
#include <stdio.h>
#define PRECISION 10

double exponencial(double num);
double factorial(int num);
double potencia(double base,int expo);

void main(void)
{
    double x,e_x;
    int i;
    printf("Numero: ");
    scanf("%lf",&x);
    e_x=exponencial(x);
    printf("la exp.de %lf es %lf\n",x,e_x);
}

double exponencial(double num)
{
    double ex=1;
    int i;
    for(i=1;i<PRECISION;i++)
    {
        ex+=potencia(num,i)/factorial(i);
    }
    return ex;
}

double factorial(int num)
{
    double fact=1;
    int i;
    for(i=1;i<=num;i++)
        fact*=i;
    return fact;
}
```

```
double potencia(double base,int expo)
{
    double res=1;
    int i;
    for(i=0;i<expo;i++)
        res*=base;
    return res;
}
```

**Ej. 6.3)** Dado el siguiente fragmento de código, se pide programar una función que sirva para calcular la distancia entre dos puntos de un espacio Ndimensional, donde N se fijara en tiempo de compilación mediante la directiva DIM.

```
#include <stdio.h>
#define DIM 3

void main()
{
    float v1[DIM], v2[DIM];
    double dist;
    printf("Introduce las coordenadasdel punto 1:\n");
    scanf("%f %f %f",&v1[0],&v1[1],&v1[2]);
    printf("Introduce las coordenadasdel punto 2:\n");
    scanf("%f %f %f",&v2[0],&v2[1],&v2[2]);

    //llamada a la funcion

    printf("DISTANCIA = %f\n",dist);
}
```

**SOLUCION:**

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

#define DIM 3

double distancia(float*, float*, int);

void main()
{
    float v1[DIM], v2[DIM];
    double dist;
    printf("Introduce las coordenadasdel punto 1:\n");
    scanf("%f %f %f",&v1[0],&v1[1],&v1[2]);
    printf("Introduce las coordenadasdel punto 2:\n");
    scanf("%f %f %f",&v2[0],&v2[1],&v2[2]);
    dist=distancia(v1,v2,DIM);
    printf("DISTANCIA = %f\n",dist);
}

double distancia(float p1[],float p2[], int dim)
{
    int i=0;
    float dist=0;

    for(i=0;i<dim;i++)
        dist+=(p1[i]-p2[i])*(p1[i]-p2[i]);

    return sqrt(dist);
}
```

**Ej. 6.4) Ecuación cuadrada**

Realizar un programa en C que resuelva una ecuación de segundo grado del tipo

$$ax^2 + bx + c = 0$$

siendo los parámetros a, b, y c introducidos por el usuario. El programa debe implementar una función que admita como parámetros a, b, c y que devuelva (por referencia) las raíces de la ecuación.

Utilizar el tipo de retorno de la función para determinar el tipo de solución (2 reales, 1 doble, 2 complejas conjugadas). Realizar la comprobación de que realmente la ecuación es cuadrada y gestionar adecuadamente soluciones dobles e imaginarias. Los operandos tendrán precisión normal. Se utilizara la sentencia `switch`. La salida por pantalla debe de ser similar a la siguiente:

```
Introduzca los coeficientes a,b y c
de ax2+bx+c=0, separados por espacios
2 3 -4
Ecuacion 2.000000x2+3.000000x+-4.000000=0
Soluciones 3.403124 -9.403124
Press any key to continue
```

O

```
Introduzca los coeficientes a,b y c
de ax2+bx+c=0, separados por espacios
2 3 4
Ecuacion 2.000000x2+3.000000x+4.000000=0
Raices complejas: -3.000000 +/- i 4.795832
Press any key to continue
```

O

```
Introduzca los coeficientes a,b y c
de ax2+bx+c=0, separados por espacios
0 2 4
Ecuacion 0.000000x2+2.000000x+4.000000=0
Ecuacion lineal, raiz: -2.000000
Press any key to continue
```

SOLUCION:

```
#include <stdio.h>
#include <math.h> //para la funcion raiz cuadrada sqrt

int raices(float a,float b,float c,float* r1,float* r2);

void main(void)
{
    float a,b,c;//coeficientes
    float r1,r2;//raices, soluciones
    int ret;

    printf("Introduzca los coeficientes a,b y c\n");
    printf("de ax2+bx+c=0, separados por espacios\n");
    scanf("%f %f %f",&a,&b,&c);
    printf("Ecuacion %fx2+%fx+%f=0\n",a,b,c);

    ret=raices(a,b,c,&r1,&r2);
```

```
switch(ret)
{
case 0:printf("Soluciones reales %f %f\n",r1,r2);
break;
case 1:printf("Raiz doble %f\n",r1);
break;
case 2:printf("Raices complejas conjugadas %f+-i%f\n",r1,r2);
break;
case 3:printf("Ecuacion lineal, raiz: %f\n",r1);
break;
case 4: printf("Error parametros\n");
break;
}
}

int raices(float a,float b,float c,float* r1,float* r2)
{
if(a!=0.0f) //ecuacion cuadrada
{
float disc;//discriminante
disc=b*b-4*a*c;
if(disc>0)//dos soluciones reales
{
*r1=(-b+(float)sqrt(disc))/2*a;
*r2=(-b-(float)sqrt(disc))/2*a;
return 0;
}
else if(disc==0)//raiz doble
{
*r1=*r2=-b/2*a;
return 1;
}
else//soluciones complejas
{
*r1=-b/2*a;
*r2=(float)sqrt(-disc)/2*a;
return 2;
}
}
else //ecuacion no cuadrada
{
if(b!=0.0f)
{
*r1=-c/b;
return 3;
}
else
return 4;
}
}
```



# 7. Entrada y salida

## 7.1 Introducción

En C, la transferencia de datos entre dispositivos se realiza con funciones de entrada y salida (E/S) que pertenecen a la librería `<stdio.h>`. Por tanto, cualquier programa que quiera utilizar una de estas funciones tendrá que incluir la ya conocida directiva:

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

La biblioteca `<stdio.h>` proporciona los prototipos de todas las funciones de E/S y define, entre otros, dos tipos de datos nuevos: `size_t` y `FILE`. El primero es un sinónimo de `unsigned int` y el segundo es una estructura de datos que se utiliza para la lectura y escritura de ficheros. Esta biblioteca también contiene dos constantes simbólicas de uso común al operar con ficheros: `NULL` (puntero nulo), cuyo valor es 0, y `EOF` (End Of File o Fin de Fichero), cuyo valor es -1.

En este capítulo se presentan, en primer lugar, las funciones básicas que permiten el intercambio de información entre el usuario y el programa a través de la E/S estándar (teclado y pantalla) y en segundo lugar, las funciones que permiten al programa leer el contenido de un fichero o escribir datos en él.

## 7.2 Entrada y salida estándar

La entrada y salida estándar son, respectivamente, el teclado y la pantalla del ordenador. De las funciones que contiene la librería `<stdio.h>` para imprimir datos en la pantalla o leerlos desde el teclado aquí sólo vamos a ver las siguientes:

Tabla 7-1. Funciones de E/S por consola

Salida (pantalla)	Entrada (teclado)
<code>printf()</code> <code>putchar()</code> <code>puts()</code>	<code>scanf()</code> <code>getchar()</code> <code>gets()</code> <code>fflush()</code>

Las funciones `printf()` y `scanf()` son funciones de propósito general que permiten leer y escribir caracteres individuales, cadenas de caracteres, números, etc., con un determinado formato. Hasta ahora son las dos únicas funciones de E/S que se han utilizado en los programas de este libro por su potencialidad y facilidad de uso.

Las funciones más simples de E/S son `putchar()`, que escribe un caracter en la pantalla, y `getchar()`, que lee un caracter desde el teclado. Las funciones `puts()` y `gets()` escriben y leen cadenas de caracteres, respectivamente. La función

`fflush()` tiene el cometido especial de limpiar un buffer de datos, por ejemplo, el de entrada de datos desde el teclado. Más adelante justificaremos su utilidad.

Las funciones anteriores se han elegido porque son las más comunes, pero no son las únicas. Existen otras, como `getc()` o `getche()`, que se han omitido para simplificar la presentación porque, aunque tienen su utilidad para determinadas aplicaciones, hacen prácticamente lo mismo que las anteriores.

Es más, las funciones que se van a presentar son redundantes y ni siquiera es necesario usar las funciones `putchar()`, `getchar()`, `puts()` y `gets()` porque `printf()` y `scanf()` pueden hacer lo exactamente mismo. No obstante, se ha decidido incluirlas porque suelen aparecer en los programas y es útil conocerlas, al menos, para leer código fuente y porque para determinadas aplicaciones pueden tener ventajas frente a `printf()` y `scanf()`.

Finalmente, hay que decir que la descripción de las funciones que se realiza en este capítulo cubre los usos más comunes pero no es completa. El lector que desee conocer todos los detalles puede recurrir a la bibliografía especializada o a la extensa ayuda que suelen proporcionar los propios entornos de programación, como el del Microsoft Visual C++.

## 7.2.1 Salida estándar

### 7.2.1.1 La función `printf()`

La función `printf()` convierte valores del programa en caracteres con un formato especificado que son mostrados en la pantalla. Formalmente, el prototipo de la función `printf()` es el siguiente:

```
int printf( const char *formato [, argumento1, argumento2...] );
```

El primer parámetro de la función (*`const char *formato`*) es obligatorio y consiste simplemente en una cadena de control, escrita entre comillas dobles, formada por caracteres ordinarios, secuencias de escape y especificadores de formato. El resto de parámetros de la función (*`argumento1`*, *`argumento2...`*) suelen ser variables, números, caracteres, etc., y pueden no existir.

La función devuelve un número entero de tipo `int` que es igual al número de caracteres escritos por la función o un número negativo si ha ocurrido un error. Normalmente, cuando se escribe en la pantalla es habitual asumir que no va a haber errores por lo que el valor devuelto por la función no suele comprobarse.

Para ilustrar el uso de esta función vamos a ver algunos ejemplos. En el primero de ellos vamos a retomar el primer programa de este libro que muestra en la pantalla el mensaje “Hola mundo”:

```
#include <stdio.h>

void main()
{
    printf("Hola mundo\n");
}
```

La primera línea del programa es la directiva que permite usar la función `printf()` en el programa. En este ejemplo, la función sólo tiene la cadena de control formada por la secuencia de caracteres ordinarios `Hola mundo` y la secuencia de escape



\n con la que indicamos al programa que salte una línea, ya que la función no lo hace automáticamente. Por ejemplo, el siguiente programa:

```
#include <stdio.h>

void main()
{
    printf("Hola");
    printf("soy el ordenador");
}
```

Imprime en la pantalla:

**Holasoy el ordenador**

En general, una secuencia de escape está formada por la barra invertida \ seguida de una serie de caracteres. En C, las secuencias de escape más comunes son:

**Tabla 7-2. Secuencias de escape comunes**

Secuencia de escape	Definición
\n	Nueva línea (retorno de carro y salto de línea)
\t	Tabulador horizontal
\b	Retroceso ("backspace")
\r	Retorno de carro sin salto de línea
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida ("backslash")
\ooo	Carácter ASCII (o es un dígito octal)
\xhh	Carácter ASCII (h es un dígito hexadecimal)

Una particularidad de las secuencias de escape es que pueden ser utilizadas directamente como caracteres individuales.

Por ejemplo:

```
#include <stdio.h>

void main()
{
    char car;

    car='\x41';
    printf("El valor de car es %c\n", car);
}
```

Cuya salida por pantalla sería:

**El valor de car es A**

El programa anterior declara una variable de tipo `char` cuyo nombre es `car` y a continuación le asigna un valor mediante la instrucción:

```
car='\x41';
```

El valor 41 hexadecimal se corresponde con 65 en decimal, que a su vez es el valor ASCII de la letra A. De hecho, la instrucción anterior también podía haberse escrito directamente como:

```
car='A';
```

La última línea del programa se encarga de imprimir en la pantalla el caracter asignado a la variable `car`. Para ello, en la cadena de control ha aparecido un elemento nuevo: `%c`. Se trata de un especificador de formato que se utiliza para imprimir caracteres individuales.

Cuando la función `printf()` encuentra un especificador de formato en la cadena de control, busca el valor que tiene que imprimir en la lista de argumentos que viene a continuación y lo imprime con el formato especificado. Si en la cadena de control aparece más de un especificador de formato la búsqueda se hará de forma correlativa, es decir, el primer especificador de formato será sustituido por el primer elemento de la lista, el segundo por el segundo, etc.

Por ejemplo, el programa siguiente:

```
#include <stdio.h>

void main()
{
    char car='A';

    printf("El valor ASCII de %c es %d\n", car, car);
}
```

Mostrará en la pantalla:

```
El valor ASCII de A es 65
```

Cuando la función `printf()` encuentra el especificador de formato `%c` lo sustituye por el primer valor de la lista, el de la variable `car`, y lo imprime como caracter. A continuación, cuando encuentra el segundo especificador de formato, `%d`, que se utiliza para imprimir números enteros, busca el segundo argumento de la lista, también la variable `car`, pero esta vez imprime su valor como un número.

En general, los tipos de datos de C tienen que ser impresos con formatos compatibles, ya que en caso contrario pueden obtenerse resultados impredecibles o incorrectos.

Por ejemplo, el siguiente programa:

```
#include <stdio.h>

void main ()
{
    float x=2, y=5, z;

    z=x/y;

    printf("La division de %d y %d vale %d\n", x, y, z);
}
```

Darí­a como resultado:

```
La division de 0 y 1073741824 vale 0
```

El resultado es erróneo porque se imprimen números reales de tipo `float` con formato de números enteros. Para obtener un resultado correcto podríamos sustituir `%d` por especificadores como `%f` o `%g` obteniendo, respectivamente, las siguientes salidas por pantalla:

**La division de 2.000000 y 5.000000 vale 0.400000**  
**La division de 2 y 5 vale 0.4**

Como puede verse, `%f` imprime los números reales con seis cifras decimales y `%g` elimina las cifras decimales no significativas y el punto decimal si no son necesarios.

La sintaxis mínima para expresar un especificador de formato consiste en escribir el símbolo de porcentaje, %, seguido de una letra que indica el tipo de datos que se van a imprimir, es decir:

***%tipo***

Donde tipo es una letra que indica si el argumento asociado tiene que interpretarse como un carácter individual, una cadena de caracteres o un número.

La tabla siguiente muestra el significado de las letras que utilizan para especificar el tipo y su significado:

**Tabla 7-3. Especificación de tipo en E/S**

tipo	Tipo de datos	Salida
d	int	Enteros con signo en decimal
i	int	Enteros con signo en decimal
u	int	Enteros sin signo en decimal
o	int	Enteros sin signo en octal
x	int	Enteros sin signo en hexadecimal (abc..)
X	int	Enteros sin signo en hexadecimal (ABC..)
f	double	Valor con signo de la forma [-]m.dddddd
e	double	Valor con signo de la forma [-]m.dddddde[±]ddd
E	double	Valor con signo de la forma [-]m.ddddddeE[±]ddd
g	double	Valor con signo compacto en formato f o e
G	double	Valor con signo compacto en formato f o E
c	int	Carácter individual de un byte
s	cadena de caracteres	Escribe la cadena hasta el primer '\0'

El especificador de formato también puede incluir, entre el símbolo % y el tipo, los siguientes campos opcionales que permiten, por ejemplo, controlar la alineación del texto de salida, la impresión de signos o la precisión (los corchetes indican que el campo escrito entre ellos es opcional):

***% [flags] [ancho] [.precisión]tipo***

El campo *[flags]* controla la alineación de la salida y la impresión de signos, blancos, puntos decimales, y prefijos octales y hexadecimales. La tabla siguiente indica el significado de los flags más comunes:

Tabla 7-4. Banderas de formato de E/S

<i>flags</i>	Significado	Por defecto
-	Alineación a la izquierda dentro del campo <i>ancho</i> especificado.	Se alinea a la derecha.
+	Antepone al número su signo (+ o -).	Se antepone el signo -
0	Rellena la salida con ceros no significativos hasta llenar el <i>ancho</i> mínimo.	Sin ceros de relleno.
blanco ( ' ' )	Antepone un espacio en blanco si el número es positivo con signo.	Sin espacio en blanco.
#	Depende del campo obligatorio <i>tipo</i> : <i>tipo: o, x o X.</i> Antepone <b>0</b> , <b>x0</b> o <b>X0</b> . <i>tipo: e, E, f, g, y G.</i> La salida tiene siempre punto decimal. <i>tipo: g y G.</i> No se truncan los ceros arrastrados. <i>tipo: c, d, i, u, o s.</i> El campo # se ignora	Sin prefijo  Sólo se pone punto decimal si hay dígitos después. Se truncan los ceros arrastrados.

El campo *[ancho]* indica el número mínimo de caracteres de salida. Si el número de caracteres de salida es menor que ancho se añaden espacios en blanco. Si es mayor que ancho, no truncan sino que se imprimen todos los caracteres.

El significado del campo *[.precisión]* depende del campo obligatorio tipo según se indica en la tabla siguiente:

Tabla 7-5. Precisión de E/S

<i>tipo</i>	Significado del campo precisión	Por defecto
d, i, u, o, x, X	Número mínimo de dígitos que se escriben. Si el número de dígitos de salida es menor que <i>precisión</i> se añaden ceros a la izquierda.	1 dígito
e, E, f	Número de dígitos después del punto decimal. El valor se redondea.	6 dígitos
g, G	Máximo número de dígitos significativos.	6 dígitos
s	Número máximo de caracteres que se escriben.	Se imprimen todos los caracteres hasta el primer carácter nulo
c	Ninguno	

Los programas siguientes ilustran el uso de algunos de los campos anteriores:

```
#include <stdio.h>

void main()
{
    int x=5422;
```

```

printf("Alineado a la derecha\n%10d\n", x);
printf("Alineado a la derecha con signo\n%+10d\n", x);
printf("Alineado a la izquierda\n%-10d\n", x);
printf("Relleno con ceros\n%010d\n", x);
printf("En octal\n\t%#o\n", x);
printf("En hexadecimal\n\t\t%#x\n", x);
}

```

Salida por pantalla:

```

Alineado a la derecha
      5422
Alineado a la derecha con signo
    +5422
Alineado a la izquierda
    5422
Relleno con ceros
0000005422
En octal
      012456
En hexadecimal
      0x152e

```

El siguiente código ilustra el uso del punto (.) como especificador de formato de longitud de numeros enteros, reales y cadenas.

```

#include <stdio.h>

void main()
{
    int x=5422;
    double y=3.1416;

    printf("Entero con 6 digitos\n%.6d\n", x);
    printf("Real por defecto\n%f\n", y);
    printf("Real con tres decimales\n%.3f\n", y);
    printf("Cadena truncada\n%.11s\n", "Frase incompleta");
}

```

Salida por pantalla:

```

Entero con 6 digitos
005422
Real por defecto
3.141600
Real con tres decimales
3.142
Cadena truncada
Frase incom

```

### 7.2.1.2 La función putchar()

La función `putchar()` escribe un caracter en la pantalla. Su prototipo es el siguiente:

```
int putchar (int caracter);
```

A la función hay que pasarle como argumento el caracter que se quiere escribir y devuelve el caracter escrito o `EOF` si ocurre un error.

Por ejemplo, el siguiente programa imprime la letra A y luego salta una línea:

```
#include <stdio.h>
void main ()
{
    char car='A';
    putchar(car);
    putchar('\n');
}
```

Si se desea utilizar la función `printf()` en lugar de `putchar()`, el programa anterior se escribiría como:

```
#include <stdio.h>
void main ()
{
    char car='A';
    printf("%c\n", car);
}
```

### 7.2.1.3 La función puts()

La función `puts()` escribe una cadena de caracteres en la pantalla. El prototipo de la función es el siguiente.

```
int puts(const char *cadena);
```

A la función hay que pasarle como argumento la cadena de caracteres que se quiere imprimir y devuelve un valor no negativo si la operación es correcta o `EOF` si ha habido un error.

La función `puts()` sustituye el caracter nulo (`'\0'`), con el que termina cualquier cadena de caracteres, por el caracter de nueva línea (`'\n'`), de manera que automáticamente salta una línea después de imprimir la cadena.

Por ejemplo:

```
#include <stdio.h>
void main ()
{
    char cadena[]="Hola";

    puts(cadena);
    puts("mundo");
}
```

Imprimiría en pantalla:

```
Hola
mundo
```

La función `puts()` también puede sustituirse por `printf()` con el mismo resultado teniendo la precaución de añadir saltos de línea en los lugares adecuados. Por ejemplo, el programa siguiente hace lo mismo que el anterior:

```
#include <stdio.h>
void main ()
{
    char cadena[]="Hola";
    printf("%s\nmundo\n", cadena);
}
```

## 7.2.2 Entrada estándar

### 7.2.2.1 La función scanf()

La función `scanf()` lee datos desde el teclado, con el formato especificado en la cadena control, y los almacena en su lista de argumentos. Formalmente, el prototipo de esta función es el siguiente:

```
int scanf( const char *formato [, argumento1, argumento2...] );
```

La función devuelve un número entero de tipo `int` que es igual al número de argumentos correctamente leídos y asignados o `EOF` si ha habido un error. Por ejemplo, en el programa siguiente, el único en el que lo vamos a hacer, se comprueba el valor devuelto por la función `scanf()`:

```
#include <stdio.h>

void main()
{
    int f=0;
    float dato=0.0f;

    printf("Introduzca el dato: ");
    f=scanf("%f", &dato);
    if(f==EOF)
        printf("Error de lectura\n")
    else
        printf("El dato es %f\n", dato);
}
```

El primer parámetro de la función (*const char \*formato*) es una cadena de control escrita entre comillas dobles que está formada normalmente por especificadores de formato separados por un espacio en blanco, aunque pueden utilizar caracteres adicionales si se desea que los datos se introduzcan con un formato concreto. Los especificadores de formato más comunes están recogidos en la tabla siguiente:

**Tabla 7-6. Especificador de formato scanf()**

Especificador de formato	Dato de entrada
%d	Entero decimal
%f	Número real
%c	Carácter individual
%s	Cadena de caracteres
%[caracteres]	Lee caracteres hasta encontrar uno que no está en caracteres
%[^caracteres]	Lee caracteres hasta encontrar uno que está en caracteres

Un error muy frecuente entre los programadores noveles es escribir mensajes dentro de la cadena de control, lo cuál no está permitido. Si es necesario escribir un mensaje en la pantalla del estilo “Introduzca los datos”, hay que utilizar las funciones `printf()` o `puts()` antes de la usar `scanf()`.

El resto de parámetros de la función (*argumento1*, *argumento2...*) son las direcciones de memoria de los objetos en los cuáles se van a almacenar los datos leídos

desde el teclado, es decir, los argumentos se pasan por referencia. Cuando lo que se va a leer es una cadena de caracteres, el nombre de la cadena no va precedido del símbolo & porque, como ya sabemos, es un puntero que contiene la dirección de memoria que apunta a su primer elemento. Sin embargo, para el resto de variables (número enteros o reales, caracteres, etc.) el argumento está formado por el nombre de la variable precedido obligatoriamente del operador & (“dirección de”).

Si se olvida el operador & al llamar a `scanf()` el compilador no lo detecta y puede tener consecuencias graves, ya que no es lo mismo el valor de una variable que la posición que ocupa dicha variable en memoria. Por ejemplo, si escribimos el siguiente programa:

```
#include <stdio.h>

void main ()
{
    int x=5000;

    printf("La direccion de memoria de x es %d\n", &x);
    printf("Introduzca otro valor para x\n");
    scanf("%d", &x);
    printf ("x vale ahora %d\n", x);
}
```

Y tecleamos 27, obtendríamos la siguiente salida por pantalla:

```
La direccion de memoria de x es 1245052
Introduzca otro valor para x
27
x vale ahora 27
```

Sin embargo, si olvidamos el símbolo & en la instrucción de `scanf()` el dato leído (27), en lugar de guardarlo en la posición 1245052 de la memoria, la función intentaría guardarlo en la posición 5000 lo que daría lugar, en el caso menos grave, a un error de ejecución del programa.

Cuando en la cadena de control aparecen caracteres ordinarios distintos de espacios en blanco para separar los especificadores de formato, los datos introducidos desde el teclado tienen que contener esos caracteres en el mismo orden en el que aparecen.

Por ejemplo, en el siguiente programa la fecha tiene que ser introducida con el formato “%d-%d-%d”, es decir, `scanf()` tiene que leer un entero, a continuación un signo menos que es descartado, luego otro entero y así sucesivamente. Si `scanf()` encuentra un caracter distinto del signo menos después de introducir un entero, incluido el retorno de carro, la función `scanf()` termina en ese momento. La única manera de introducir los datos correctamente es escribir, por ejemplo, 14-12-2007, y luego pulsar retorno de carro.

```
#include <stdio.h>

void main ()
{
    int d, m, a;

    printf("Introduzca la fecha (dd-mm-aaaa): ");
    scanf("%d-%d-%d", &d, &m ,&a);
}
```



Cuando en la cadena de control se deja un espacio en blanco después de cada especificador de formato, `scanf()` lee, pero ignora, cualquier espacio en blanco que se introduzca desde el teclado hasta que encuentra un caracter que no sea un espacio en blanco, normalmente retorno de carro.

Cuando se leen caracteres individuales, los espacios en blanco son leídos como cualquier otro caracter, lo cuál puede dar lugar a errores de asignación. Por ejemplo, si al ejecutar el siguiente programa:

```
#include <stdio.h>

void main ()
{
    char a, b, c;

    scanf("%c%c%c", &a, &b, &c);
    printf("a vale %c\n",a);
    printf("b vale %c\n",b);
    printf("c vale %c\n",c);
}
```

Tecleamos x y z, aparecería en pantalla:

```
x y z
a vale x
b vale
c vale y
```

Es decir, a la variable b se le ha asignado el espacio en blanco que hemos dejado entre x e y. Una forma de evitar este error de asignación es dejar siempre un espacio en blanco entre los especificadores de formato (por ejemplo, `"%c %c %c"`).

La función `scanf()` tiene una particularidad cuando lee cadenas de caracteres. Por ejemplo, si ejecutamos el siguiente programa:

```
#include <stdio.h>

void main ()
{
    char asignatura[50];

    scanf("%s", asignatura);
    printf("%s\n", asignatura);
}
```

Y tecleamos **Fundamentos de Informatica**, la salida del programa será la siguiente:

```
Fundamentos de Informatica
Fundamentos
```

Es decir, en la cadena de caracteres asignatura sólo se ha guardado la palabra Fundamentos porque `scanf()` sólo lee caracteres hasta que encuentra el primer espacio en blanco (`'\0'`).

Para leer cadenas de caracteres que contengan espacios en blanco utilizando la función `scanf()` puede utilizarse la especificación de formato personalizada `%[^\n]`, es decir, se leen caracteres hasta que se encuentra retorno de carro. Utilizando esta especificación el programa anterior quedaría como:

```
#include <stdio.h>

void main ()
{
    char asignatura[50];

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

Cuya salida por pantalla sería la esperada:

<b>Fundamentos de Informatica</b> <b>Fundamentos de Informatica</b>
--

En este caso particular, puede ser más cómodo utilizar la función `gets()` que se describe más adelante, ya que lee la cadena hasta que se pulsa retorno de carro. En el ejemplo anterior simplemente escribiríamos, anticipándonos a lo que viene:

```
gets(asignatura);
```

### 7.2.2.2 La función `getchar()`

La función `getchar()` lee un caracter individual desde el teclado. Su prototipo es el siguiente.

```
int getchar (void);
```

La función no tiene argumentos y devuelve el caracter leído o `EOF` si ocurre un error o encuentra la marca `EOF`.

Por ejemplo, el programa siguiente lee un caracter desde el teclado y lo muestra en pantalla:

```
#include <stdio.h>

void main ()
{
    char car;

    printf("Introduzca un caracter\n");
    car=getchar();
    printf("El caracter leído es %c\n", car);
}
```

Puede verse que para guardar el caracter leído desde el teclado se asigna a una variable de tipo `char` el valor devuelto por la función `getchar()`. También podíamos haber utilizado la función `scanf()` en lugar de `getchar()` sustituyendo la instrucción:

```
car=getchar();
```

Por la siguiente:

```
scanf("%c", &car);
```

### 7.2.2.3 La función `gets()`

La función `gets()` lee una cadena de caracteres desde el teclado hasta que se pulsa retorno de carro. El prototipo de la función es el siguiente.

```
char *gets(char *cadena);
```

La función tiene como argumento el nombre de la cadena donde se almacenará la cadena leída y devuelve un puntero a la misma cadena o un puntero nulo (`NULL`) si se ha producido un error.

Por ejemplo, el programa siguiente es equivalente a uno anterior pero utiliza `gets()` en lugar de `scanf()` para leer la cadena de caracteres:

```
#include <stdio.h>

main ()
{
    char asignatura[50];

    gets(asignatura);
    printf("%s\n", asignatura);
}
```

A diferencia de `scanf()`, `gets()` puede leer directamente cadenas de caracteres que contengan espacios en blanco. Si se quiere hacer lo mismo con `scanf()` hay que utilizar la especificación personalizada de formato `%[^\n]`, tal y como se ha indicado anteriormente.

#### 7.2.2.4 La función `fflush()`

La función `fflush()` es una función que se encarga de limpiar un buffer de datos. Su prototipo es el siguiente:

```
int fflush(FILE *flujo);
```

La función tiene como argumento un puntero de tipo `FILE` que apunta al buffer de datos que se quiere limpiar y devuelve un 0 si el buffer se ha limpiado correctamente o `EOF` si se ha producido un error).

En el caso habitual de que queramos limpiar el buffer de entrada del teclado, simplemente tenemos que escribir:

```
fflush(stdin);
```

La necesidad de utilizar esta función se ilustra con el siguiente programa que lee dos caracteres individuales desde el teclado utilizando la función `scanf()`:

```
#include <stdio.h>

void main()
{
    char car1, car2;
    printf("Introduzca el primer caracter\n");
    scanf("%c", &car1);
    printf("Introduzca el segundo caracter\n");
    scanf("%c", &car2);
    printf("Los caracteres introducidos son %c y %c\n", car1, car2);
}
```

Suponiendo que queremos introducir A y B, ejecutamos el programa y después de teclear A y retorno de carro el programa termina mostrando lo siguiente:

```
Introduzca el primer caracter
A
Introduzca el segundo caracter
Los caracteres introducidos son A y
```

¿Qué ha ocurrido? Después de teclear **A** y pulsar retorno de carro el contenido del buffer del teclado es el siguiente:

'A'	'\n'			
-----	------	--	--	--

Pero después de que la función `scanf()` lea el carácter **A** y lo almacene en la variable `car1`, el buffer no se queda vacío, sino que en él permanece el carácter `'\n'`:

'\n'				
------	--	--	--	--

Por tanto, al ejecutar la instrucción:

```
scanf("%c", &car2);
```

Automáticamente se asigna a la variable `car2` el carácter `'\n'`. La última línea del programa muestra los valores de `car1` y `car2`, es decir, imprime una **A** y salta una línea.

Para evitar este problema, hay que utilizar la función `fflush()` antes volver a leer el buffer de entrada. Para el programa anterior, bastaría con limpiar el buffer antes de ejecutar la segunda función `scanf()`, por ejemplo:

```
#include <stdio.h>

void main()
{
    char car1, car2;

    printf("Introduzca el primer caracter\n");
    scanf("%c", &car1);
    printf("Introduzca el segundo caracter\n");
    fflush(stdin);
    scanf("%c", &car2);
    printf("Los caracteres introducidos son %c y %c\n", car1, car2);
}
```

## 7.3 Gestión de ficheros

En cualquier programa de los que hemos visto hasta ahora no se guarda ninguna información y todos los datos se pierden cuando termina la ejecución del mismo, lo cuál es un serio inconveniente. En este apartado aprenderemos como guardar los datos del programa en un fichero y como leer un fichero para recuperar datos previamente guardados.

Muchas de las funciones que se utilizan para leer o escribir ficheros son prácticamente idénticas a las funciones que se han estudiado en el apartado anterior para leer desde el teclado o escribir en la pantalla. Por ejemplo, `fprintf()` y `fscanf()` se utilizan para escribir y leer datos con formato de un fichero.

Cuando se trabaja con un fichero siempre se realiza la misma secuencia de operaciones: se abre el fichero, se realizan operaciones de lectura o escritura (L/E), y se cierra el fichero. En cada uno de los tres pasos anteriores pueden ocurrir errores que es necesario detectar y controlar para que el programa se ejecute correctamente.

A continuación se describen las operaciones básicas de apertura y cierre de un fichero. Después se presentan algunas funciones que permiten gestionar errores. Y por último, se describen las funciones más comunes que se utilizan para L/E.

### 7.3.1 Abrir un fichero: la función `fopen()`

La función `fopen()` abre un fichero para leer o escribir en él. Formalmente, su prototipo es el siguiente:

```
FILE *fopen (const char *nombre, const char *modo);
```

El primer parámetro de la función es el nombre del fichero, el cuál tiene que respetar las normas del sistema operativo en el que se ejecute el programa. En los ejemplos se ha utilizado el formato de DOS (Disk Operating System), que es válido para la mayoría de los sistemas operativos, como Windows o UNIX. En DOS, el nombre del archivo puede tener de 1 a 8 caracteres y la extensión de 0 a 3. Sin embargo, en los ejemplos sólo se utiliza la extensión `.txt`, de manera que los archivos generados por los programas pueden abrirse con cualquier editor de texto.

El segundo parámetro de la función `fopen()` indica el modo en el que se va a abrir el fichero, que puede ser para leer, escribir o añadir información ("`r`", "`w`" o "`a`", respectivamente) tal como se describe en la tabla siguiente.

**Tabla 7-7. Modos de apertura de un fichero**

Modo	Descripción
" <code>r</code> "	Abre el fichero para leer ( <u>r</u> ead). El fichero tiene que existir (si no existe, se produce un error)
" <code>w</code> "	Abre el fichero para escribir ( <u>w</u> rite). Si el fichero no existe, se crea. Si el fichero existe, se borra su contenido antes de escribir.
" <code>a</code> "	Abre un fichero para añadirle información ( <u>a</u> dd). Los datos nuevos se añaden al final del fichero. Si el fichero no existe, se crea.

Existen otros modos de abrir un fichero, que sólo vamos a mencionar aquí de pasada, ya que no los vamos a utilizar. Por ejemplo, añadiendo un signo más a los anteriores se obtienen tres nuevos modos ("`r+`", "`w+`" y "`a+`") que permiten abrir el fichero para leer y escribir.

Los ficheros también pueden abrirse en modo texto (`t`) o en modo binario (`b`) añadiendo la letra indicada (`t` o `b`) al modo. Por ejemplo, "`rb`", "`wb`" o "`ab`". Por omisión, el fichero siempre se abre en modo texto.

Cuando se crea un fichero el sistema añade automáticamente al final del mismo la marca fin de fichero (`EOF`).

La función `fopen()` devuelve un puntero a una estructura de tipo `FILE` o un puntero nulo (`NULL`) si se ha producido un error, por ejemplo, si el fichero que se quiere abrir no existe o si está protegido contra escritura.

Normalmente se comprueba si se ha producido este error (es decir, si el puntero devuelto por `fopen()` tiene valor `NULL`) y, si es el caso, se avisa al usuario de esta circunstancia y suele terminarse la ejecución del programa (en la mayoría de los programas que vienen a continuación no se realiza esta comprobación para simplificar la presentación del código).

Si no se produce un error al abrir el fichero, el puntero devuelto por `fopen()` se asigna a un puntero de tipo `FILE` que debemos haber declarado previamente en el programa. Por ejemplo:

```
FILE *pf;
pf=fopen("datos.txt", "r");
```

La primera instrucción declara un puntero de tipo `FILE` llamado “`pf`” y la segunda le asigna el puntero devuelto por la función `fopen()`, que en este ejemplo intenta abrir un fichero llamado `datos.txt` para leer su contenido. Una vez realizada esta asignación, para referirnos al fichero (`datos.txt`) utilizaremos el puntero asociado al mismo (`pf`).

Por ejemplo, el siguiente programa abre para leer un fichero llamado “`datos.txt`”:

```
#include <stdio.h>
void main ()
{
    FILE *pf;
    pf=fopen("datos.txt", "r");
    if (pf==NULL)
    {
        printf("Error al abrir el fichero\n");
        return; // Se termina el programa en este punto
    }
    else
        printf("Fichero abierto\n");
}
```

Cuya salida en pantalla sería, si no existe el fichero en el directorio de trabajo:

#### Error al abrir el fichero

Al finalizar el programa, cualquier fichero abierto se cierra automáticamente. Sin embargo, es una buena práctica cerrar un fichero cuando se ha acabado de trabajar con él, por ejemplo, para permitir que otros programas puedan acceder al mismo.

### 7.3.2 Cerrar un fichero: la función `fclose()`

La función `fclose()` cierra un fichero abierto previamente con `fopen()`. Su prototipo es el siguiente:

```
int fclose (FILE *pf);
```

El argumento que hay que pasarle a la función es el puntero de tipo `FILE` que apunta al fichero. La función devuelve 0 si el fichero se cierra correctamente o `EOF` si se produce un error.

Ejemplo:

```
#include <stdio.h>

void main ()
{
    FILE *pf;
    pf=fopen("datos.txt", "w"); //Abrimos el fichero para escribir

    //No escribimos nada (aun) en el fichero

    fclose(pf); //Cerramos el fichero
}
```

Tras ejecutar este programa, puede comprobarse que en el directorio de trabajo ha aparecido un fichero con el nombre “datos.txt” que podemos editar para comprobar que está vacío, ya que no hemos realizado ninguna operación de escritura:

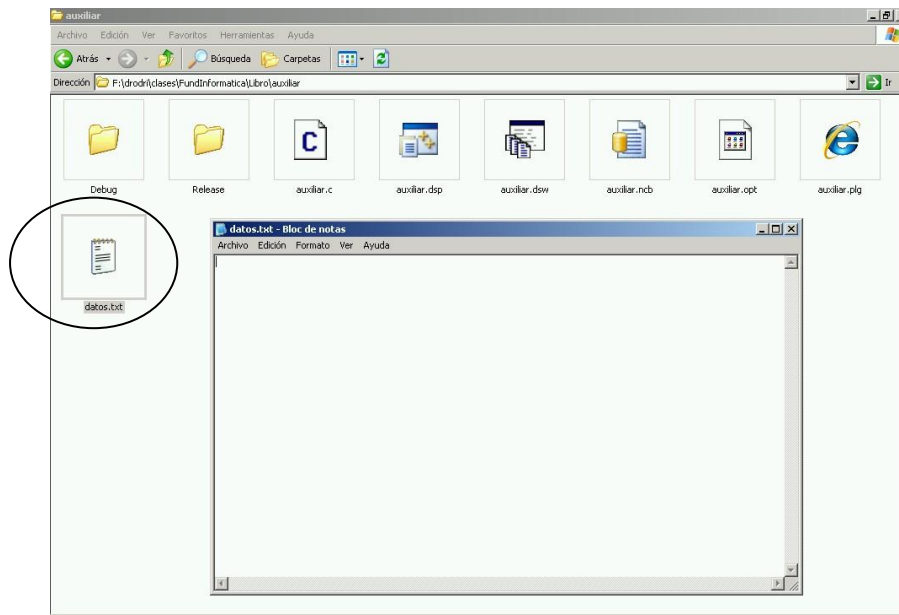


Figura 7-1. Creación de un fichero de texto

En resumen, la estructura de un programa para escribir o leer un fichero es la siguiente:

```
#include <stdio.h>
...
void main()
{
    ...
    FILE *pf;
    ...
    pf=fopen("nombre", "modo");
    ...
    // Operaciones de lectura o escritura
    ...
    fclose(pf);
    ...
}
```

### 7.3.3 Gestión de errores

Cuando se realizan operaciones de lectura o escritura con ficheros es necesario comprobar si se ha producido un error durante una operación o si se ha alcanzado la marca final de fichero (EOF). Para estos propósitos, C proporciona una serie de funciones que pertenecen a la librería `<stdio.h>`, de las cuáles se presentan aquí las siguientes: `ferror()`, `perror()`, `clearerr()` y `feof()`.

La función `ferror()` detecta si se ha producido un error después de operar con el fichero. El prototipo de la función es el siguiente:

```
int ferror (FILE *pf);
```

El argumento es el puntero de tipo `FILE` asociado al fichero. Devuelve un valor distinto de cero cuando se ha producido un error y 0 en caso contrario.

La función `perror()` escribe un mensaje de error en la pantalla. El prototipo de la función es el siguiente:

```
void perror (const char *mensaje);
```

El argumento es un mensaje que se mostrará en pantalla. La función escribe dicho mensaje terminado en dos puntos y añade el mensaje de error dado por el sistema.

La función `clearerr()` borra los errores que se hayan producido. El prototipo de la función es el siguiente:

```
void clearerr (FILE *pf);
```

El argumento es el puntero de tipo `FILE` que apunta al fichero. El programa siguiente muestra un ejemplo de uso de las tres funciones anteriores:

```
#include <stdio.h>

void main ()
{
    FILE *pf;

    pf=fopen("datos.txt", "r");

    if (pf==NULL)
    {
        perror("datos.txt");
        return;
    }

    // Aquí vendrían operaciones de lectura

    if (ferror(pf))          // Equivalente a if(ferror(pf)!=0)
    {
        perror("Error de lectura");
        clearerr(pf);
    }
    else
        printf("Lectura correcta");

    fclose(pf);
}
```

Cuya salida por pantalla sería, si no existe el fichero `datos.txt`, la siguiente:

```
datos.txt: No such file or directory
```

Finalmente, la función `feof()` detecta el final del fichero. Su prototipo es:

```
int feof (FILE *pf);
```

El argumento es el puntero asociado al fichero. La función devuelve un valor distinto de cero después de la primera operación que intente leer después de la marca final del fichero. Por ejemplo, para leer un fichero que no sabemos donde acaba puede utilizarse el siguiente bucle:

```
while (feof(pf)==0)
{
    // Operaciones de L/E
}
```



A veces, la expresión condicional del bucle anterior se escribe de una forma más compacta, pero también más críptica, como:

```
while(!feof(pf))
{
    // Operaciones de L/E
}
```

Las funciones de gestión de errores se van a omitir en los apartados siguientes para simplificar la presentación de los programas. No obstante, su uso es siempre recomendable para detectar errores y hacer los programas más robustos.

### 7.3.4 Operaciones de lectura/escritura

En este apartado se describen algunas funciones de la biblioteca `<stdio.h>` que se utilizan para leer y escribir ficheros. Son las siguientes:

**Tabla 7-8. Funciones de E/S a fichero**

Escritura	Lectura	Operaciones de L/E
<code>fprintf()</code>	<code>fscanf()</code>	Con formato
<code>fputc()</code>	<code>fgetc()</code>	Caracteres individuales
<code>fputs()</code>	<code>fgets()</code>	Cadenas de caracteres
<code>fwrite()</code>	<code>fread()</code>	Bloques de bytes

Muchas de estas funciones son redundantes. Las más utilizadas son `fprintf()` y `fscanf()` por su potencialidad y facilidad de uso. Las cuatro siguientes: `fputc()`, `fgetc()`, `fputs()` y `fgets()`, se presentan para que el lector las conozca y porque pueden serle de utilidad en alguna aplicación. No obstante, todo lo que pueden hacer también pueden hacerlo `fprintf()` y `fscanf()`. Incluso `fprintf()` y `fscanf()` también puede sustituir a sus homólogas `printf()` y `scanf()`, tal como se cuenta más adelante. Sin embargo, se ha seguido el criterio de usar las primeras cuando se opere ficheros y las segundas cuando se trate del teclado o la pantalla.

Las funciones `fwrite()` y `fread()` son más específicas para trabajar con ficheros. Permiten leer o escribir en bloques de bytes, lo cuál es útil, por ejemplo, cuando hay que trabajar con tipos de datos avanzados como, por ejemplo, estructuras.

#### 7.3.4.1 Las funciones `fprintf()` y `fscanf()`

Estas dos funciones permiten escribir y leer datos con formato y son idénticas a las funciones `printf()` y `scanf()` salvo porque tienen un parámetro adicional, situado antes de la cadena de control, que recibe el puntero asociado al fichero en el que se pretende leer o escribir.

Por ejemplo, si queremos mostrar el mensaje “Hola” en la pantalla podemos escribir:

```
printf("%s\n", "Hola");
```

Y para hacer lo mismo en un fichero:

```
fprintf(pf, "%s\n", "Hola");
```

Donde `pf` es el puntero asociado al fichero donde queremos escribir.

Con la función `fprintf()` también podemos imprimir en la pantalla, exactamente igual que lo hace `printf()`, poniendo en el lugar del puntero el nombre del flujo de datos para la salida estándar (`stdout`), es decir:

```
fprintf(stdout, "%s\n", "Hola");
```

Análogamente, si queremos leer desde el teclado una cadena de caracteres (que no contenga espacios en blanco) podemos escribir:

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

Donde `cadena` es una cadena de caracteres declarada previamente (hay que recordar, de nuevo, que `cadena` no necesita ir precedida del operador `&` porque el nombre de una cadena de caracteres es un puntero que contiene la dirección de memoria de su primer elemento).

Por tanto, si queremos realizar la misma operación pero leyendo los datos de un fichero escribiremos:

```
fscanf(pf, "%s", cadena);
```

Donde `pf` es el puntero asociado al fichero abierto previamente.

También podemos utilizar la función `fscanf()` para leer desde el teclado sustituyendo el puntero por el nombre del flujo de datos asociado con la entrada estándar (`stdin`), es decir:

```
fscanf(stdin, "%s", cadena);
```

Por ejemplo, el programa siguiente abre un archivo llamado “datos.txt”, escribe el mensaje “Hola” (más un salto de línea) y lo cierra. A continuación lo vuelve abrir, esta vez para leerlo con `fscanf()`, y muestra su contenido en la pantalla:

```
#include <stdio.h>

void main ()
{
    FILE *pf;
    char cadena[11];

    pf=fopen("mensaje.txt", "w");
    fprintf(pf, "%s\n", "Hola");
    fclose(pf);

    pf=fopen("mensaje.txt", "r");
    fscanf(pf, "%s", cadena);
    fclose(pf);

    printf("%s\n", cadena);
}
```

En el ejemplo siguiente se guardan dos vectores de la función `main()` en un fichero, cada uno en una columna.

```
#include <stdio.h>

void main ()
{
    int v1[5]={-1, 3, 5, 0, 4};
    int v2[5]={4, 9, -8, 2, 3};
    int i;
    FILE *pf;
```

```

pf=fopen("datos.txt", "w");

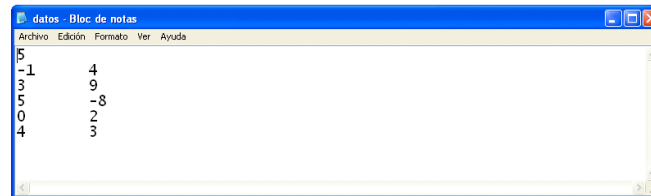
fprintf(pf, "%d\n", 5); //Guardamos primero la dimensión

for(i=0; i<5; i++)
    fprintf(pf, "%d\t%d\n", v1[i], v2[i]);

fclose(pf);
}

```

Si después de ejecutar el programa editamos el archivo “datos.txt” que se creó en el directorio de trabajo, podemos ver que su contenido coincide con lo esperado:



**Figura 7-2. Fichero de texto con los datos grabados**

La dimensión que hemos escrito en la primera línea del fichero es un recurso muy utilizado por los programadores que facilita la lectura posterior del fichero, ya que se conoce el número de operaciones de lectura que hay que hacer. Por ejemplo:

```

#include <stdio.h>

void main ()
{
    int v1[20];
    int v2[20];
    int i, dimension;
    FILE *pf;

    pf=fopen("datos.txt", "r");

    fscanf(pf, "%d", &dimension);          //Leemos primero la dimensión

    //Leemos los datos y los almacenamos en dos vectores

    for(i=0; i<dimension; i++)
        fscanf(pf, "%d %d", &v1[i], &v2[i]);

    fclose(pf);

    //Imprimimos v1 y v2 en la pantalla

    for(i=0; i<dimension; i++)
        printf("%d\t%d\n", v1[i], v2[i]);
}

```

Si no conocemos el número de datos que contiene el fichero tenemos que leer los datos sucesivamente hasta encontrar la marca EOF. Una alternativa para realizar esta operación lectura es utilizar la función `fEOF()` tal y como se ha contado más arriba. Sin embargo, en su lugar vamos a utilizar el valor devuelto por la propia función `fscanf()`, que es el número de argumentos correctamente leídos y asignados o EOF si ha habido un error.

Por tanto, para recorrer secuencialmente el fichero podemos escribir:

```
while( fscanf( ...) != EOF )
{
    // Sentencias
}
```

Cuando se escribe la condición booleana `fscanf(...) != EOF` dentro del bucle no sólo comprueba si el valor devuelto por la función es distinto de `EOF`, sino que también se ejecuta la operación de lectura correspondiente.

El siguiente programa lee números enteros de un fichero y los imprime en la pantalla:

```
#include <stdio.h>

void main ()
{
    FILE *pf;
    int x;

    pf=fopen("datos1.txt", "r");

    while(fscanf(pf, "%d", &x) != EOF)
        printf("%d\n", x);

    fclose(pf);
}
```

Por ejemplo, si creamos a mano un archivo llamado “datos1.txt” con un editor de texto (como el NotePad o bloc de notas) con el siguiente contenido:

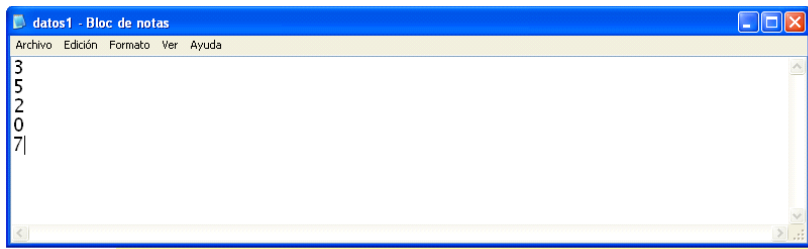


Figura 7-3. Fichero de texto con datos

Y lo guardamos en el directorio de trabajo, el programa anterior lo imprimirá en pantalla:

```
3
5
2
0
7
```

### 7.3.4.2 Las funciones `fputc()` y `fgetc()`

Estas funciones permiten escribir o leer ficheros carácter a carácter. La función `fputc()` escribe un carácter en el fichero. Su prototipo es el siguiente:

```
int fputc (int car, FILE *pf);
```

El primer parámetro, `car`, es el caracter que se quiere escribir y el segundo, `"pf"`, el puntero asociado al fichero. La función devuelve el caracter escrito o `EOF` si se produce un error.

Por ejemplo, el siguiente programa escribe el mensaje "Hola Mundo" en el archivo "saludo.txt":

```
#include <stdio.h>

void main()
{
    FILE *pf;
    char mensaje[]="Hola Mundo";
    int i=0;

    pf=fopen("saludo.txt", "w");

    while (mensaje[i]!='\0')
    {
        fputc(mensaje[i], pf);
        i++;
    }

    fclose(pf);
}
```

La función `fgetc()` lee un caracter del fichero. Su prototipo es el siguiente:

```
int fgetc (FILE *pf);
```

El argumento que hay que pasar a la función es el puntero asociado al fichero. La función devuelve el caracter leído (que habrá que asignar, por ejemplo, a una variable de tipo `char` declarada previamente) o `EOF` si se produce un error o llegamos al final del fichero. Es decir, puede que la función devuelva `EOF` sin haber llegado al final del fichero (para distinguir un caso del otro habría que utilizar además `feof()` o `ferror()`).

Por ejemplo, el siguiente programa lee un fichero y lo imprime en la pantalla:

```
#include <stdio.h>

void main()
{
    FILE *pf;
    char character;

    pf=fopen("mensaje.txt", "r");

    while((character=fgetc(pf))!=EOF)
        printf("%c", character);

    printf("\n");

    fclose(pf);
}
```

Cuya salida podría ser:

**Este es el contenido del archivo "mensaje.txt", creado con un editor de texto y guardado en el directorio de trabajo antes de ejecutar el programa.**

### 7.3.4.3 Las funciones `fputs()` y `fgets()`

Estas funciones permiten escribir y leer cadenas de caracteres. La función `fputs()` escribe una cadena de caracteres en un fichero sin el caracter nulo (`'\0'`). Su prototipo es el siguiente:

```
int fputs (const char *cadena, FILE *pf);
```

A la función hay que pasarle como primer argumento la cadena que se quiere escribir y como segundo, el puntero asociado al fichero. Devuelve un valor negativo si funciona correctamente o `EOF` si se produce un error.

Para recuperar la cadena de caracteres escrita en el fichero es aconsejable añadir el caracter de nueva línea (`'\n'`) al final de la cadena. Por ejemplo:

```
#include <stdio.h>

void main()
{
    FILE *pf;

    pf=fopen("datos.txt", "w");

    fputs("Hola Mundo", pf);
    fputc('\n', pf);

    fclose(pf);
}
```

La función `fgets()` lee una cadena de caracteres de un fichero. Su prototipo es el siguiente:

```
char *fgets (char *cadena, int n, FILE *pf);
```

El primer parámetro es el nombre de la cadena de caracteres donde se almacena la cadena leída, la cuál es terminada automáticamente con `'\0'`.

El segundo parámetro, `n`, está relacionado con el tamaño de la cadena que se quiere leer y tiene el significado que se describe más abajo.

Y el tercer argumento, es el puntero asociado al fichero. La función devuelve un puntero al principio de la cadena leída o un puntero nulo (`NULL`) si se produce un error o se detecta la marca `EOF` (para distinguir un caso del otro habría que utilizar `feof()` o `ferror()`).

La cadena de caracteres leída por la función `fgets()` es la que va hasta:

- El primer caracter `'\n'` inclusive, o
- Hasta el final del fichero, o
- Hasta que se han leído `n-1` caracteres.

Por ejemplo, el siguiente programa abre un archivo de datos para leer, llama a la función `fgets()` y comprueba si ha habido un error o se ha llegado al final del fichero. A continuación, si el puntero no es nulo, se lee la cadena hasta un máximo de 100 caracteres:

```
#include <stdio.h>

void main()
{
```

```

FILE *pf;
char cadena[101];

pf=fopen("datos.txt", "r");

if( fgets(cadena, 101, pf) == NULL)
    printf("Error o EOF\n");
else
    printf( "%s", cadena);

fclose(pf);
}

```

#### 7.3.4.4 Las funciones fwrite() y fread()

Hasta ahora, todos los archivos que se han utilizado en los programas son archivos de texto en los que los datos son almacenados como caracteres ASCII, cada uno de los cuáles ocupa un byte. Una forma de almacenar los datos que requiere menos espacio en disco es utilizar el modo binario en lugar del modo texto. En modo binario, los datos se almacén igual que en la memoria del ordenador. Por ejemplo, en modo binario un número `int` o `float` ocuparía cuatro bytes.

Las funciones `fwrite()` y `fread()` permiten escribir y leer los datos en bloques de bytes, de longitud fija y en binario.

La función `fwrite()` escribe un bloque de bytes en un fichero. Su prototipo es el siguiente:

```
size_t fwrite (const void *p, size_t n, size_t e, FILE *pf);
```

La función escribe, en el fichero asociado con `"pf"`, `e` elementos de longitud `"n"` bytes que están almacenados a partir de la posición de memoria indicada por el puntero `"p"`. La función devuelve el número de elementos escritos (si el número devuelto es menor que `"e"`, es que se ha producido un error).

Por ejemplo, el programa siguiente guarda dos números enteros en un archivo:

```

#include <stdio.h>

void main()
{
    int a=1, b=2;

    pf=fopen("datos.txt", "wb");

    fwrite(&a, sizeof(int), 1, pf);
    fwrite(&b, sizeof(int), 1, pf);

    fclose(pf);
}

```

El programa siguiente guarda el mensaje "Hola Mundo" en un archivo:

```

#include <stdio.h>

void main ()
{
    FILE *pf;
    char mensaje[]="Hola Mundo";

    pf=fopen("datos.txt", "wb");
}

```

```
fwrite(mensaje, sizeof(mensaje), 1, pf);  
fclose(pf);  
}
```

La función `fread()` lee un bloque de bytes. Su prototipo es el siguiente:

```
size_t fread (void *p, size_t n, size_t e, FILE *pf);
```

La función lee, desde el fichero asociado con `"pf"`, `"e"` elementos de longitud `n` bytes y los almacena a partir de la dirección de memoria indicada por el puntero `"p"`.

La función devuelve el número de elementos leídos. Si el número devuelto es menor que `e`, es que se ha producido un error o se ha llegado al final del fichero (para distinguir un caso de otro habría que usar `feof()` o `ferror()`).

Para leer un fichero con `fread()` hasta el final el mismo podemos utilizar el siguiente bucle:

```
while(fread( ... , ... , e , ...) == e)  
{  
    ...  
}
```

Es decir, se realiza la operación de lectura con `fread()` y se compara el valor devuelto con el número de elementos que se quieren leer (`e`).

Una de las aplicaciones más útiles de estas funciones emerge cuando se quieren leer o escribir tipos de datos avanzados, por ejemplo, estructuras.

El ejemplo siguiente muestra un ejemplo de un programa para rellenar fichas de alumnos que son almacenadas en un fichero:

```
#include <stdio.h>  
  
typedef struct  
{  
    char nombre[51];  
    char apellidos[51];  
    int matricula;  
}  
ficha;  
  
void main ()  
{  
    ficha alumno;  
    char opcion;  
    FILE *pf;  
    pf=fopen("alumnos.txt", "wb");  
  
    do  
    {  
        fflush(stdin);  
  
        printf("\nNombre: ");  
        gets(alumno.nombre);  
  
        printf("Apellidos: ");  
        gets(alumno.apellidos);  
  
        printf("Numero de matricula: ");  
        scanf("%d", &alumno.matricula);
```



```

    fwrite(&alumno, sizeof(ficha), 1, pf); //Guardamos la ficha

    fflush(stdin);
    printf("\nMas alumnos (s/n)?\n");
    scanf("%c", &opcion);
}
while (opcion=='s');

fclose(pf);
}

```

Ejemplo de salida por pantalla:

```

Nombre: Pepe
Apellidos: Perez Perez
Numero de matricula: 123

Mas alumnos (s/n)?
s

Nombre: Sancho
Apellidos: Sanchez Sanchez
Numero de matricula: 456

Mas alumnos (s/n)?
n

```

El programa siguiente abre el archivo anterior y muestra las fichas en la pantalla:

```

#include <stdio.h>

typedef struct
{
    char nombre[51];
    char apellidos[51];
    int matricula;
}
ficha;

void main ()
{
    ficha alumno;
    FILE *pf;
    pf=fopen("alumnos.txt", "rb");
    while(fread(&alumno, sizeof(ficha), 1, pf)==1)
    {
        printf("Nombre: %s\n", alumno.nombre);
        printf("Apellidos: %s\n", alumno.apellidos);
        printf("Numero de matricula: %d\n\n",alumno.matricula);
    }

    fclose(pf);
}

```

Salida por pantalla:

```

Nombre: Pepe
Apellidos: Perez Perez
Numero de matricula: 123
Nombre: Sancho
Apellidos: Sanchez Sanchez
Numero de matricula: 456

```

## 7.4 Ejercicios resueltos

### Ej. 7.1) Codificador y decodificador de ficheros de texto

Realizar un programa que lea un fichero de texto del disco y codifique o descodifique su información, guardándola en un nuevo fichero de texto:

- El fichero de texto de entrada se generara con NotePad y tendrá la extensión .txt.
- Los nombres de ambos ficheros, el de entrada y de salida se le pedirán al usuario.
- Si el programa no encuentra el archivo de entrada, informara del error y saldrá del programa.
- Se le pedirá al usuario si quiere codificar (1) o descodificar (0) el archivo en cuestión.
- La codificación consistirá en cambiar cada caracter por el siguiente a->b, b->c y sucesivamente, incluidos los caracteres de control, espacios, retornos de carro, etc.
- La descodificación consistirá en cambiar cada caracter por el anterior b->a, c->b, etc., al igual que el punto anterior.

Ejemplo de ejecución:

```
Nombre fichero entrada: Carta.txt
Codificar (1) o descodificar (0): 1
Nombre fichero salida: CartaCod.txt
Fin de la codificacion-descodificacion
Press any key to continue
```

FICHERO ORIGINAL:

```
Estimado Sr. Consejero Delegado,

Esta carta es absolutamente confidencial. No debe ser vista por nadie.
Datos restringidos:
- Capital de la empresa: 111243444 euros
- Cuenta bancaria: 123456789
- Banco: MultiMillion Zurich Bank
- Clave: elefante

Atentamente,
Su contable.
```

FICHERO CODIFICADO:

```
Ftujnbep!Ts!/Dpotfkfsp!Efmfhbep-

Ftub!dbsub!ft!bctpmvubnfouf!dpogjefodjbm!/Op!efcf!tfs!wjtub!qps!obejf/
Ebupt!sftusjohjept;
.IDbqjubm!ef!mb!fnqsftb;!222354555!fvspt
.IDvfoub!cbodbsjb;!23456789:
.ICbodp;!NvmujNjmmjpo![vsjdi!Cbol
.IDmbwf;!fmfgbouf

Bufoubnfouf-
Tv!dpoubcmf/
```

**SOLUCION:**

```

#include <stdio.h>
void main(void)
{
    FILE* f_ent;
    FILE* f_sal;
    int codifica;
    char nombre_ent[200], nombre_sal[200];
    printf("Nombre fichero entrada: ");
    scanf("%s", nombre_ent);
    printf("Codificar (1) o decodificar (0): ");
    scanf("%d", &codifica);
    printf("Nombre fichero salida: ");
    scanf("%s", nombre_sal);

    f_ent=fopen(nombre_ent, "r"); //el fichero de entrada
    if(f_ent==NULL)
    {
        printf("No se encuentra el fichero %s\n", nombre_ent);
        return; //salimos del programa
    }
    f_sal=fopen(nombre_sal, "w");

    while(1)
    {
        char car;
        int n;
        n=fscanf(f_ent, "%c", &car);
        if(n!=1)
            break;
        if(codifica) //la clave es muy obvia
            car+=1; //cambia a->b, b->c, etc
        else
            car-=1; //para decodificar b->a, c->b, etc
        fprintf(f_sal, "%c", car);
    }
    printf("Fin de la codificacion-descodificacion\n");
    fclose(f_ent);
    fclose(f_sal);
}

```

**Ej. 7.2) Gestión de pedidos**

En un fichero de texto denominado “Inventario.txt” se almacena informacion del inventario de un almacen de la siguiente forma:

<i>Elemento</i>	<i>capacidad</i>	<i>stock</i>	<i>precio_unitario</i>
-----------------	------------------	--------------	------------------------

Donde “Elemento” es una cadena de texto sin espacios identificando el tipo de objeto, “capacidad” es un número entero que representa el número maximo de esos elementos que se pueden almacenar en el almacen, “stock” es el número de elementos de ese tipo actualmente en el almacen y “precio\_unitario” es el precio de cada uno de los elementos.

Se desea hacer un programa que lea ese fichero de texto y genere otro fichero de texto denominado “Pedidos.txt” en el que se escribira una lista de aquellos objetos de los que no quedan existencias (stock=0), y la cantidad que se deben de pedir (igual a la capacidad maxima de ese elemento). El programa sacara por pantalla ademas el valor total del almacen actual, asi como cuanto costara el pedido que se va a hacer:

### Ejemplo de fichero "Inventario.txt"

<b>Tornillos</b>	<b>120</b>	<b>0</b>	<b>1</b>	<b>2.5</b>
<b>Tuercas</b>	<b>30</b>	<b>101</b>	<b>23.0</b>	
<b>Llaves</b>	<b>100</b>	<b>0</b>	<b>32.7</b>	
<b>Arandelas</b>	<b>200</b>	<b>152</b>	<b>0.7</b>	

### Fichero "Pedidos.txt" resultante

<b>Tornillos 120</b>
<b>Llaves 100</b>

### Salida por pantalla:

<b>El importe total del pedido sera: 4770.000000</b>
<b>El valor del almacen es: 2429.399902</b>

### SOLUCION:

```
#include <stdio.h>

void main(void)
{
    FILE* f1;
    FILE* f2;

    float coste_total=0.0f;
    float valor_actual=0.0f;

    f1=fopen("Inventario.txt","r");
    f2=fopen("Pedidos.txt","w");

    while(1)
    {
        char item[30];
        int capacidad;
        int cantidad;
        float precio;
        int leidos;

        leidos=fscanf(f1,"%s %d %d %f",item,&capacidad,&cantidad,&precio);

        if(leidos!=4)
            break;

        if(cantidad==0)
        {
            fprintf(f2,"%s %d\n",item,capacidad);
            coste_total+=capacidad*precio;
        }
        valor_actual+=cantidad*precio;
    }

    printf("El importe total del pedido sera: %f\n",coste_total);
    printf("El valor del almacen es: %f\n",valor_actual);

    fclose(f1);
    fclose(f2);
}
```

**Ej. 7.3) Ordenar datos de un fichero**

Ordenar un fichero de datos llamado “Vector.txt”, que contiene un máximo de 100 datos de tipo real. El primer dato contenido en el fichero es un número entero que define el número de datos que viene a continuación en el fichero.

SOLUCION:

```
#include <stdio.h>

void main(void)
{
    FILE* f;
    int num_datos;
    int i,j;
    float vector[100];

    f=fopen("Vector.txt","r");

    fscanf(f,"%d",&num_datos);
    for(i=0;i<num_datos;i++)
    {
        fscanf(f,"%f",&vector[i]);
    }
    fclose(f);

    for(i=0;i<num_datos-1;i++)
    {
        for(j=i+1;j<num_datos;j++)
        {
            if(vector[i]>vector[j])
            {
                float aux=vector[i];
                vector[i]=vector[j];
                vector[j]=aux;
            }
        }
    }

    f=fopen("Vector.txt","w");
    fprintf(f,"%d\n",num_datos);
    for(i=0;i<num_datos;i++)
    {
        fprintf(f,"%f\n",vector[i]);
    }
    fclose(f);
}
```



# 8. Problemas resueltos

## 8.1 *Máximo común divisor (MCD)*

Hacer un programa que calcule el máximo común divisor (MCD) de dos números enteros positivos, m y n, utilizando el siguiente algoritmo propuesto por Euclides en el año 300 a.C.:

1. Se comprueba que m es mayor o igual que n. Si m es menor, hay que intercambiar los valores de m y n.
2. Se divide m entre n y se calcula el resto.
3. Si el resto es cero, n es el MCD (fin del algoritmo).
4. Si el resto es distinto de cero, se reemplaza m por n y n por resto.
5. Se vuelve al punto 2.

Nota: Los números m y n se introducirán desde el teclado. Si el usuario introduce un número menor que 1 se le pide que los introduzca de nuevo.

SOLUCION:

```
#include <stdio.h>

void main ()
{
    int m, n, temporal;

    // Petición de m y n positivos

    do
    {
        printf("Introduzca m y n\n");
        scanf("%d %d", &m, &n);
    }
    while (m<1 || n<1);

    // Intercambio de m y n si m<n

    if (m<n)
    {
        temporal=m;
        m=n;
        n=temporal;
    }
```

```
// Algoritmo de Euclides

while ((m%n)!=0)
{
    temporal=m;
    m=n;
    n=temporal%n;
}

// Imprime el MCD

printf("El MCD vale %d\n", n);
}
```

## 8.2 Adivinar un número

Hacer un programa que adivine un número entre 0 y 1000 pensado por el usuario. El programa debe ir proponiendo números al usuario y éste debe contestar si son mayores o menores que el número pensado o si el ordenador ha acertado, en cuyo caso se imprimirá el número de intentos.

SOLUCION:

```
#include <stdio.h>

void main ()
{
    // Intervalo del juego
    int minimo=0, maximo=1000;
    // Otras variables
    int prediccion, intentos=0, respuesta;

    printf("Piense un numero del %d al %d\n\n", minimo, maximo);

    do
    {
        prediccion=(maximo+minimo)/2;
        printf("\nEs su numero %d?\n", prediccion);

        printf("1. Si \n");
        printf("2. No, es menor \n");
        printf("3. No, es mayor \n\n");

        scanf("%d", &respuesta);
        intentos++;

        switch(respuesta)
        {
            case 2:
                maximo=prediccion;
                break;
            case 3:
                minimo=prediccion;
            }
        }
        while(respuesta!=1);

        printf("\nHe necesitado %d intentos\n", intentos);
    }
}
```



### 8.3 Lotería

Realizar un programa que obtenga una combinación de lotería compuesta por 6 números DISTINTOS en el rango 1-49, ambos incluidos. Para ello se utilizara la función `NumeroAleatorio()` que devuelve un número entero aleatoriamente en el rango que se le define [min,max] ambos incluidos.

El esqueleto del programa deberá ser el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
int NumeroAleatorio(int min,int max);
int main()
{
}
int NumeroAleatorio(int min,int max)
{
    int n=min+rand()%(max-min+1);
    return n;
}
```

NOTA: Nótese que la función `NumeroAleatorio` no realiza ninguna comprobación de que los números sean distintos. Es decir la función, invocada varias veces seguidas podría devolver los valores 1, 13, 1, 5, 5, por ejemplo, lo que NO es una combinación valida de lotería. El programa del usuario debe solucionar este problema. Se recomienda utilizar vectores para almacenar la información.

**42 44 14 41 11 45**

SOLUCION 1:

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

int NumeroAleatorio(int min,int max);
int main()
{
    int i;
    int extraido[49]={0}; //al ppio todaslas bolas estan en el bombo
    int numeros[6];

    for(i=0;i<6;i++) //6 numeros distintos
    {
        int numero,repetido;
        do
        {
            int j;
            numero=NumeroAleatorio(1,49); //sacar un n° aleatorio
            repetido=0; //en ppio suponemos que no esta repetido
            if(extraido[numero-1]==1)
                repetido=1;
            else
            {
                extraido[numero-1]=1;
                repetido=0;
            }
        }
        while(repetido);
        numeros[i]=numero; //si estamos aqui sabemos que es distinto
        //lo guardamos y vamos a por el siguiente
    }
}
```

```
//sacar los numeros por pantalla
for(i=0;i<6;i++)
{
    printf("%d ",numeros[i]);
}
printf("\n");
}
int NumeroAleatorio(int min,int max)
{
    int n=min+rand()%(max-min+1);
    return n;
}
```

## SOLUCION 2:

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

int NumeroAleatorio(int min,int max);
int main()
{
    int i;
    int numeros[6];

    for(i=0;i<6;i++) //6 numeros distintos
    {
        int numero,repetido;
        do
        {
            int j;
            numero=NumeroAleatorio(1,49); //sacar un n°aleatorio
            repetido=0; //en ppio suponemos que no esta repetido
            for(j=0;j<i;j++) //lo comprobamos con los anteriores
            {
                if(numeros[j]==numero)
                {
                    repetido=1; //si que esta repetido
                    break; //no seguir comprob., sale del for
                }
            }
        } while(repetido);
        numeros[i]=numero; // aqui ya sabemos que es distinto
        //lo guardamos y vamos a por el siguiente
    }

    //sacar los numeros por pantalla
    for(i=0;i<6;i++)
    {
        printf("%d ",numeros[i]);
    }
}
int NumeroAleatorio(int min,int max)
{
    int n=min+rand()%(max-min+1);
    return n;
}
```

## 8.4 Integral

Realizar un programa que calcule la integral numérica de una función (utilizar como ejemplo la función  $y=x^2$ , aunque se puede utilizar cualquier otra) en el intervalo  $[a,b]$ , siendo  $a$  y  $b$  tecleados por el usuario. Esta integral corresponde al área encerrada por la función con el eje  $X$  en es intervalo y se puede aproximar cuando  $\Delta x$  es muy pequeño por:

$$\int_a^b f(x)dx \approx \sum_{x=a, a+\Delta x, a+2\Delta x \dots}^b f(x)\Delta x$$

Utilizar un valor de  $\Delta x = 1e-6$ .

SOLUCION:

```
#include <stdio.h>

void main()
{
    double a,b,x;
    double incremento=1e-7,suma=0,integral;

    printf("Introduzca a y b: ");
    scanf("%lf %lf",&a,&b);

    for(x=a;x<=b;x+=incremento)
    {
        suma+=exp(x);
    }
    integral=suma*incremento;

    printf("la integral es: %lf\n",integral);
}
```

## 8.5 Números primos

Realizar un programa que muestre por pantalla todos los números primos menores que uno dado por el usuario. Para ello se programará una función denominada EsPrimo, que tiene como único parámetro el número entero que se quiere comprobar si es primo o no. La función devuelve un 1 si el número es primo y un 0 en caso contrario.

Una vez realizado lo anterior, se solicita una función que genere un número primo aleatorio en el rango  $[min,max]$  siendo estos valores definidos como parámetros. El número primo aleatorio será devuelto mediante el valor de retorno de la función. Utilizar esta función para imprimir 20 números aleatorios por pantalla en el rango 0-100. Utilícese la función `rand()` de `<stdlib.h>` que devuelve un número entero aleatorio en el rango 0-65535

SOLUCION:

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

int EsPrimo(int n);
int PrimoAleatorio(int min, int max);
```

```

void main()
{
    int i,num;

    printf("Numero:");
    scanf("%d",&num);

    for(i=1;i<=num;i++)
    {
        if(EsPrimo(i))
            printf("%d ",i);
    }

    printf("\n20 Numeros primos aleatorios:\n");
    for(i=0;i<20;i++)
    {
        printf("%d ",PrimoAleatorio(0,100));
    }
}

int EsPrimo(int n)
{
    int divisor;
    for(divisor=2;divisor<=sqrt(n);divisor++)
    {
        if(n%divisor==0)
        {
            return 0; //No primo
        }
    }
    return 1; //numero primo
}

int PrimoAleatorio(int min, int max)
{
    int num;
    do
    {
        num=min+rand()%(max-min+1);
    }
    while(!EsPrimo(num));
    return num;
}

```

**Numero:100**

**1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97**

**20 Numeros primos aleatorios:**

**41 67 61 83 71 31 23 11 13 67 97 19 0 71 37 71 5 71 29 13**

## 8.6 Fibonacci

Escriba un programa para generar los  $n$  primeros términos de la serie de Fibonacci. El valor de  $n$  (entero y positivo) deberá ser leído por teclado. En esta serie los dos primeros números son 1, y el resto se obtiene sumando los dos anteriores:

1,1,2,3,5,8,13,21.

Garantizar mediante un bucle que el número introducido por el usuario es mayor o igual que 2, para que al menos la sucesión tenga siempre 2 términos. Se le pide al usuario un número y caso de que no sea mayor de dos, se le vuelve a pedir, y así sucesivamente.

**SOLUCION:**

```
#include <stdio.h>

void main ()
{
    int n, i, penultimo=0, ultimo=1, siguiente;

    // Lectura de n (mayor que 2)
    do
    {
        printf("Introduzca el valor de n\n");
        scanf("%d", &n);
    }
    while (n<=2);

    // Imprime los dos primeros términos
    printf("\n%d\n%d\n", penultimo, ultimo);

    // Imprime los siguientes términos
    for(i=1 ; i<=(n-2) ; i++)
    {
        // Calcula el término siguiente y lo imprime
        siguiente=penultimo+ultimo;
        printf("%d\n", siguiente);

        // Actualiza los dos últimos términos
        penultimo=ultimo;
        ultimo=siguiente;
    }
}
```

**8.7 Media y rango de un vector**

Completar el siguiente programa, que pretende sacar por pantalla la media y el rango de un vector de números aleatorios. El rango de un vector es la diferencia entre el valor máximo y el mínimo de dicho vector. La estructura del programa a completar es la siguiente:

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

#define NUM_ELEM 55

void test ();
void main()
{
    test ();
}

void test ()
{
    float m, r;
    float vector[NUM_ELEM];

    rellenar_vector(vector, NUM_ELEM);
    m=calcular_media(vector, NUM_ELEM);
    r=calcular_rango(vector, NUM_ELEM);
    printf("Media= %f  Rango= %f\n", m, r);
}
```

NOTA: La función `rellenar_vector` inicializará los contenidos del vector. Para generar los números aleatorios supóngase que se dispone de una función `float NumeroAleatorio()` que devuelve un número aleatorio distinto cada vez que es llamada.

**SOLUCION:**

```
//prototipos
void rellenar_vector(float v[],int num);
float calcular_media(float v[], int num);
float calcular_rango(float v[], int num);

//cuerpo de las funciones
void rellenar_vector(float v[],int num)
{
    int i;

    for(i=0; i<num; i++)
        v[i] = NumeroAleatorio();
}

float calcular_media(float v[], int num)
{
    int i;
    float media,suma=0;

    for(i=0; i<num; i++)
        suma += v[i];
    media = suma/num;
    return (media);
}

float calcular_rango(float v[], int num)
{
    int i;
    float rango, min, max;
    min = max= v[0];
    for(i=1; i<num; i++)
    {
        if(v[i]>max)
            max=v[i];
        if(v[i]<min)
            min=v[i];
    }
    rango = max - min;
    return (rango);
}
```

# 9. Aplicación práctica: Agenda

En este capítulo se presenta una pequeña aplicación completa que permite gestionar una agenda telefónica de contactos. No se pretende que sea una aplicación real, sino un compendio de todo lo desarrollado en este libro, ya que se incluyen en esta aplicación:

- Tipos de datos avanzados: estructuras, vectores, cadenas de caracteres
- Estructuración del programa en funciones
- Paso de parámetros por valor y por referencia, punteros.
- Funciones de E/S por consola y fichero
- Sentencias de control

El programa descrito aquí tiene muchas simplificaciones respecto a lo que podría ser un programa real. Por ejemplo, no realiza comprobaciones de que lo que teclea el usuario sea correcto. Así, si el programa espera un número (teléfono) y el usuario teclea texto, el comportamiento del programa puede ser indeterminado. También la memoria se gestiona estáticamente, admitiendo un número máximo de contactos. El programa fallaría si el usuario edita a mano el fichero de texto e introduce más entradas de las permisibles. Una implementación más correcta utilizaría memoria dinámica. Tampoco está pensado para gestionar nombres o apellidos compuestos (separados por espacios) o apellidos extraordinariamente largos.

## 9.1 Características de la agenda

Antes de programar nuestra agenda, establecemos lo que queramos que haga, esto es los requisitos de la misma, y los resumimos en los siguientes:

- El programa debe cargar automáticamente los datos de la agenda al comenzar desde un fichero de texto denominado “Agenda.txt”, que contendrá tantas líneas como contactos, y en cada línea el nombre, apellido y número de teléfono de cada contacto.
- El programa debe permitir mostrar la agenda, añadir un nuevo contacto, eliminar un contacto.
- Cuando el programa termina, guarda automáticamente los datos actualizados en el mismo fichero del que los leyó.

- El programa debe permitir buscar contactos que tengan un determinado nombre, o apellido, o teléfono, mostrando todos los contactos que coincidan.
- El programa debe permitir ordenar la agenda en orden creciente (alfabético) según el nombre, el apellido o el teléfono.

## 9.2 Prototipos de funciones y función *main()*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>

#define MAX_NUM_CONTACTOS 200

//defines para identificar el campo por el que buscamos u ordenamos
#define NOMBRE 1
#define APELLIDO 2
#define TELEFONO 3

//estructura para almacenar los datos de 1 contacto
typedef struct
{
    char nombre[51];
    char apellido[51];
    int telefono;
}contacto;

//Funciones que manejan 1 unico contacto
//para E/S por consola
contacto PedirContacto();
void MostrarContacto(contacto c);
//para E/S por fichero de texto, pasando un fichero abierto
//como parametro
int LeerContacto(FILE* f,contacto* c);
int GuardarContacto(FILE* f,contacto c);

//funciones con 2 contactos
//la funcion Compara devuelve -1 si c1<c2, 0 si c1=c2 o 1 si c1>c2
//analizando el campo especificado en el parametro
int Compara(contacto c1,contacto c2,int campo);
//la funcion Intercambia es necesaria para ordenamiento de la agenda
void Intercambia(contacto* c1,contacto* c2);

//funciones que manejan toda la agenda
//leer y guardar todos los datos en un fichero de texto
//cuyo nombre se pasa por parametro
int LeerAgenda(char fichero[],contacto agenda[],int* num);
int GuardarAgenda(char fichero[],contacto agenda[],int num);
//Muestra la agenda por pantalla
void MostrarAgenda(contacto agenda[],int num);
//ordena la agenda en funcion del "campo"
void OrdenarAgenda(contacto agenda[],int num,int campo);
//Añade el contacto "c" a la agenda
void NuevoContacto(contacto agenda[],int* num,contacto c);
//Elimina el contacto numero "i" de la agenda
void EliminarContacto(contacto agenda[],int* num,int i);
//Busca en la agenda los contactos cuyo "campo" coincidan con el
//contacto "c" que se pasa como parametro
void BuscarContacto(contacto agenda[],int num,contacto c,int campo);
```



```

//funciones de E/S basica y menus del programa
int Menu();//menu principal de opciones
int MenuCampo();//menu para seleccionar un campo
void ImprimeCabecera();//Imprime el significado de cada campo
void ImprimePie();//una linea de guiones, para formato
//funcion auxiliar que pide datos de un contacto para buscar
void Buscar(contacto agenda[],int num);

void main()
{
    contacto agenda[MAX_NUM_CONTACTOS];
    int num_contactos=0;
    int opcion=0;
    contacto c;
    int n;

    //se leen los datos del fichero
    LeerAgenda("Agenda.txt",agenda,&num_contactos);

    do
    {
        opcion=Menu();
        switch(opcion)
        {
            case 0: printf("Saliendo del programa\n");
                    break;
            case 1: MostrarAgenda(agenda,num_contactos);
                    break;
            case 2: c=PedirContacto();
                    NuevoContacto(agenda,&num_contactos,c);
                    break;
            case 3: Buscar(agenda,num_contactos);
                    break;
            case 4: n=MenuCampo();
                    OrdenarAgenda(agenda,num_contactos,n);
                    MostrarAgenda(agenda,num_contactos);
                    break;
            case 5: MostrarAgenda(agenda,num_contactos);
                    printf("Numero de contacto a eliminar: ");
                    scanf("%d",&n);
                    EliminarContacto(agenda,&num_contactos,n);
                    MostrarAgenda(agenda,num_contactos);
                    break;
        }
        //algunos detalles para una salida mas limpia
        printf("Pulse una tecla para continuar...\n");
        getch(); //espera a que se pulse una tecla
        system("cls");//esto borra la consola
    }
    while(opcion!=0);

    //cuando sale el programa guarda automaticamente el fichero
    GuardarAgenda("Agenda.txt",agenda,num_contactos);
}

```

### 9.3 Funciones de gestión de un contacto

Estas funciones permiten imprimir por pantalla o solicitar al usuario que teclee los datos de un contacto, así como guardarlos o leerlos de un fichero previamente abierto. Las funciones de E/S de fichero, necesitan que se les pase el puntero a [FILE](#),

que ha debido ser abierto previamente con `fopen()`. Si el fichero no ha sido abierto, o la lectura o escritura falla (por ejemplo, porque no queden más datos que leer), entonces, la función correspondiente `fprintf()` o `fscanf()` devolverá algo distinto al número de campos correctamente gestionados, que en este caso debe de ser igual a 3.

```
//se pide un contacto que se devuelve en return
contacto PedirContacto()
{
    contacto c;

    fflush(stdin);
    printf("Nombre: ");
    gets(c.nombre);
    printf("Apellido: ");
    gets(c.apellido);
    printf("Telefono: ");
    scanf("%d", &c.telefono);

    return c;
}

//funcion que imprime todos los campos del contacto "c"
void MostrarContacto(contacto c)
{
    printf("%-20s %-20s %-10d\n",c.nombre,c.apellido,c.telefono);
}

//funciones de E/S por fichero de texto
int LeerContacto(FILE* f,contacto* c)
{
    if(f==NULL) //si el fichero no esta abierto
        return 0; //devuelvo "falso"

    if(3==fscanf(f,"%s %s %d",c->nombre,c->apellido,&c->telefono))
        return 1;//se han leído bien los 3 campos, devuelve "cierto"

    return 0;
}

int GuardarContacto(FILE* f,contacto c)
{
    if(f==NULL)
        return 0;

    if(3==fprintf(f,"%s %s %d\n",c.nombre,c.apellido,c.telefono))
        return 1;//se han guardado bien los 3 campos

    return 0;
}
```

## 9.4 Funciones que manejan 2 contactos

Estas funciones permiten hacer operaciones sobre 2 contactos que se pasan como parámetros. La primera de ellas, `Intercambia()` pasa los parámetros por referencia para poder modificar sus valores. Esta función es necesaria para realizar el ordenamiento de la agenda.

La siguiente función `Compara()`, trabaja de forma similar a la función `strcmp()` de `<string.h>`, devolviendo -1,0, o 1 en función de la ordenación lógica de los contactos pasados como parámetros. En función del parámetro “campo” se analiza uno u otro miembro de la estructura contacto. El hecho de comparar contactos es mejor que comparar teléfonos o nombres sueltos, ya que permitiría con algunas modificaciones realizar búsquedas con condiciones compuestas del tipo “Nombre=XXXXXX y Apellido=XXXXXX” o “Nombre=XXXXXX o Telefono=0000000”

```
void Intercambia(contacto* c1, contacto* c2)
{
    contacto aux=*c1;
    *c1=*c2;
    *c2=aux;
}
int Compara(contacto c1, contacto c2, int campo)
{
    if(campo==NOMBRE)
        return strcmp(c1.nombre, c2.nombre);
    if(campo==APELLIDO)
        return strcmp(c1.apellido, c2.apellido);
    if(campo==TELEFONO)
    {
        if(c1.telefono < c2.telefono)
            return -1;
        else if(c1.telefono > c2.telefono)
            return 1;
        else
            return 0;
    }
    return -1;
}
```

## 9.5 Funciones de gestión de la agenda

Las siguientes funciones gestionan la agenda completa, que es pasada como parámetro (realmente los dos primeros parámetros) a cada una de las funciones. Estas funciones se encargan de realizar operaciones que afectan a la agenda completa: mostrar la agenda por pantalla, buscar un contacto (por distintos criterios), ordenar la agenda (por distintos criterios), añadir un contacto nuevo a la agenda, y suprimir un contacto ya existente.

```
//impresión de la agenda completa por pantalla, recorriendo el vector
//que se pasa como parametro

void MostrarAgenda(contacto agenda[], int num)
{
    int i;
    ImprimeCabecera(); //Esta funcion se vera despues, es solo formato
    for(i=0; i<num; i++)
    {
        printf("%d: ", i); //mostramos el numero de contacto
        MostrarContacto(agenda[i]); //y luego el contacto
    }
    ImprimePie(); //Esta funcion se vera despues, es solo formato
}
```

```
//funcion similar a la anterior, solo que unicamente muestra
//los contactos que coinciden con el "campo" adecuado de "c"
void BuscarContacto(contacto agenda[],int num,contacto c, int campo)
{
    int i;
    ImprimeCabecera();
    for(i=0;i<num;i++)
    {
        if(0==Compara(agenda[i],c,campo)) //se ha encontrado 1 coincidenc.
        {
            printf("%d: ",i); //mostramos el numero de contacto
            MostrarContacto(agenda[i]); //y luego el contacto
        }
    }
    ImprimePie();
}

//ordena la agenda en funcion del "campo"
void OrdenarAgenda(contacto agenda[],int num,int campo)
{
    int i,j;
    for(i=0;i<num-1;i++)
    {
        for(j=i+1;j<num;j++)
        {
            if(0<Compara(agenda[i],agenda[j],campo)) //si desordenados
                Intercambia(&agenda[i],&agenda[j]); //los intercambiamos
        }
    }
}

//añade un nuevo contacto al final de la agenda
void NuevoContacto(contacto agenda[],int* num,contacto c)
{
    int n=*num;
    if(n<MAX_NUM_CONTACTOS) //comprobamos que no hemos superado el max
    {
        agenda[n]=c; //lo añadimos
        n++; //el numero de contactos incrementa en uno
    }
    *num=n; //actualizamos el parametro pasado por referencia
}

//elimina el contacto numero "n"
void EliminarContacto(contacto agenda[],int* num,int n)
{
    int i;
    int numero=*num;
    if(n<0 || n>= numero) //el numero de contacto debe existir
        return; //si no, no hacemos nada

    for(i=n;i<numero-1;i++) //movemos todos los contactos una posicion
    {
        agenda[i]=agenda[i+1];
    }
    *num=numero-1; //el numero de contactos disminuye en 1
}
```

```

//esta funcion abre un fichero cuyo nombre se le pasa como parametro
//y lee toda la agenda del fichero
//notese que el parametro "num" que es el numero de contactos
//se pasa por referencia, para que la funcion lo actualice
int LeerAgenda(char fichero[],contacto agenda[],int* num)
{
    contacto contact;//variable temporal para leer del fichero
    int n=0;//el numero de contactos leidos

    //se abre el fichero
    FILE* f=fopen(fichero,"r");
    //si no se abre correctamente (a lo mejor no existe)
    if(f==NULL)
        return 0; //se termina de leer

    //mientras se lea correctamente un contacto mas
    while(LeerContacto(f,&contact))
    {
        //Si leemos correctamente, el vector crece
        agenda[n]=contact;
        n++;
    }
    //se actualiza el numero de contactos
    *num=n;

    //se cierra el fichero
    fclose(f);
    return 1;
}

//funcion para guardar los datos en el fichero especificado
int GuardarAgenda(char fichero[],contacto agenda[],int num)
{
    int i;
    FILE* f=fopen(fichero,"w");

    if(f==NULL)
        return 0;

    for(i=0;i<num;i++) //se recorre el vector
        GuardarContacto(f,agenda[i]); //guardando cada contacto

    fclose(f);
    return 1;
}

```

## 9.6 Funciones auxiliares de gestión de E/S

En este apartado se describen algunas funciones auxiliares de gestión de entrada y salida del programa: la muestra de menús de opciones, la selección del usuario y la comprobación de la misma, un par de funciones auxiliares que ponen cabecera y pie a la agenda cuando se saca por pantalla.

También se incluye una función auxiliar para la búsqueda de contactos en la agenda, que se encarga de pedir al usuario que es lo que quiere buscar, para luego llamar a la función `BuscarContacto()` descrita anteriormente, que es la que efectivamente realiza la búsqueda.

```
//esta funcion se encarga de imprimir las opciones
//y pedir al usuario que elija una de ellas
int Menu()
{
    int opcion=-1;

    printf(".....Menu Principal.....\n");
    printf("0-Salir\n");
    printf("1-Mostrar agenda\n");
    printf("2-Nuevo contacto\n");
    printf("3-Buscar contacto\n");
    printf("4-Ordenar agenda\n");
    printf("5-Eliminar contacto\n");
    do
    {
        printf("Elija una opcion: ");
        fflush(stdin);
        scanf("%d",&opcion);
    }
    while(opcion<0 || opcion>6);
    return opcion;
}

int MenuCampo() //para seleccionar un campo
{
    int opcion=-1;

    printf(".....Seleccione campo.....\n");
    printf("%d-Nombre\n",NOMBRE);
    printf("%d-Apellido\n",APELLIDO);
    printf("%d-Telefono\n",TELEFONO);
    do
    {
        printf("Elija una opcion: ");
        fflush(stdin);
        scanf("%d",&opcion);
    }
    while(opcion<NOMBRE || opcion>TELEFONO);
    return opcion;
}

void Buscar(contacto agenda[],int num)
{
    contacto c;
    int campo=MenuCampo();
    switch(campo)
    {
        case NOMBRE: printf("Nombre a buscar: ");
                     scanf("%s",c.nombre);
                     break;
        case APELLIDO:printf("Apellido a buscar: ");
                     scanf("%s",c.apellido);
                     break;
        case TELEFONO:printf("Telefono a buscar: ");
                     scanf("%d",&c.telefono);
                     break;
    }
    BuscarContacto(agenda,num,c,campo);
}
```

```

void ImprimeCabecera()
{
    printf("-----\n");
    printf("#: %-20s %-20s %-10s\n", "Nombre", "Apellido", "Telefono");
    printf("-----\n");
}
void ImprimePie()
{
    printf("-----\n");
}

```

## 9.7 Resultado

Nótese como en el programa no existen variables globales. Aunque pudiera parecer que en algunos casos el programa sería más sencillo, se reitera que su uso no es recomendable. En nuestro caso, las variables que contienen los datos de la agenda, son locales a la función `main()`, y son pasadas como parámetro a las funciones correspondientes. Esto permite por ejemplo, con algunas pequeñas modificaciones, gestionar desde el mismo programa 2 agendas totalmente diferentes (de dos personas diferentes, o una agenda profesional y otra personal), cosa que con variables globales sería terriblemente difícil.

A continuación se muestra una salida por pantalla ejemplo del programa

.....Menu Principal.....

0-Salir

1-Mostrar agenda

2-Nuevo contacto

3-Buscar contacto

4-Ordenar agenda

5-Eliminar contacto

Elija una opcion: 1

#: Nombre	Apellido	Telefono
0: Diego	Alonso	92123098
1: Jaime	Gomez	92476494
2: Manolo	Benitez	933453453
3: Arturo	Fernandez	913456789
4: Maria	Gonzalez	91234567
5: Lucia	Fernandez	911234567
6: Isabel	Rodriguez	91234567

Pulse una tecla para continuar...

.....Menu Principal.....

0-Salir

1-Mostrar agenda

2-Nuevo contacto

3-Buscar contacto

4-Ordenar agenda

5-Eliminar contacto

Elija una opcion: 3

.....Seleccione campo.....

1-Nombre

2-Apellido

3-Telefono

Elija una opcion: 2

Apellido a buscar: Fernandez

```
-----
#: Nombre      Apellido      Telefono
-----
3: Arturo      Fernandez      913456789
5: Lucia       Fernandez      911234567
-----
```

Pulse una tecla para continuar...

.....Menu Principal.....

0-Salir

1-Mostrar agenda

2-Nuevo contacto

3-Buscar contacto

4-Ordenar agenda

5-Eliminar contacto

Elija una opcion: 4

.....Seleccione campo.....

1-Nombre

2-Apellido

3-Telefono

Elija una opcion: 1

```
-----
#: Nombre      Apellido      Telefono
-----
0: Arturo      Fernandez      913456789
1: Diego       Alonso        92123098
2: Isabel      Rodriguez      91234567
3: Jaime       Gomez          92476494
4: Lucia       Fernandez      911234567
5: Manolo      Benitez        933453453
6: Maria       Gonzalez       91234567
-----
```

Pulse una tecla para continuar...



# Bibliografía

A continuación se dan algunas referencias bibliográficas que pueden servir de ayuda para el aprendizaje y perfeccionamiento de las técnicas de programación en C.

- Curso de programación C/C++, Ceballos, Francisco Javier, Rama, 1995
- El lenguaje de programación C, Kernighan, Brian W. y Ritchie, Dennis M., Prentice Hall Hispanoamericana, 1991
- A Book on C. Programming in C, Fourth Edition. Al Kelley, Ira Pohl. Addison-Wesley. 1997
- Programación estructurada en C Antonakos J.L. , Mansfield Jr., KC. Prentice Hall, 1997
- Programación en Microsoft C, Robert Lafore – Grupo Waite, Anaya, 1990
- Programación en C/C++, Pappas, Ch.H y Murray, V.H. Anaya, 1966
- Visual C++ 6.0 Manual del programador. Beck Zaratian. McGraw Hill. 1999
- Aprendiendo Borland C++ 5, Arnush, C. Prentice Hall, 1997
- <http://www.tayuda.com/ayudainf/index.htm>. Serie de apuntes “Aprenda como si estuviera en primero”
- <http://www.elai.upm.es/> Departamento de Electrónica, Automática, e Informática Industrial de la UPM. En Asignaturas->Fundamentos de Informática, se puede encontrar material auxiliar.

