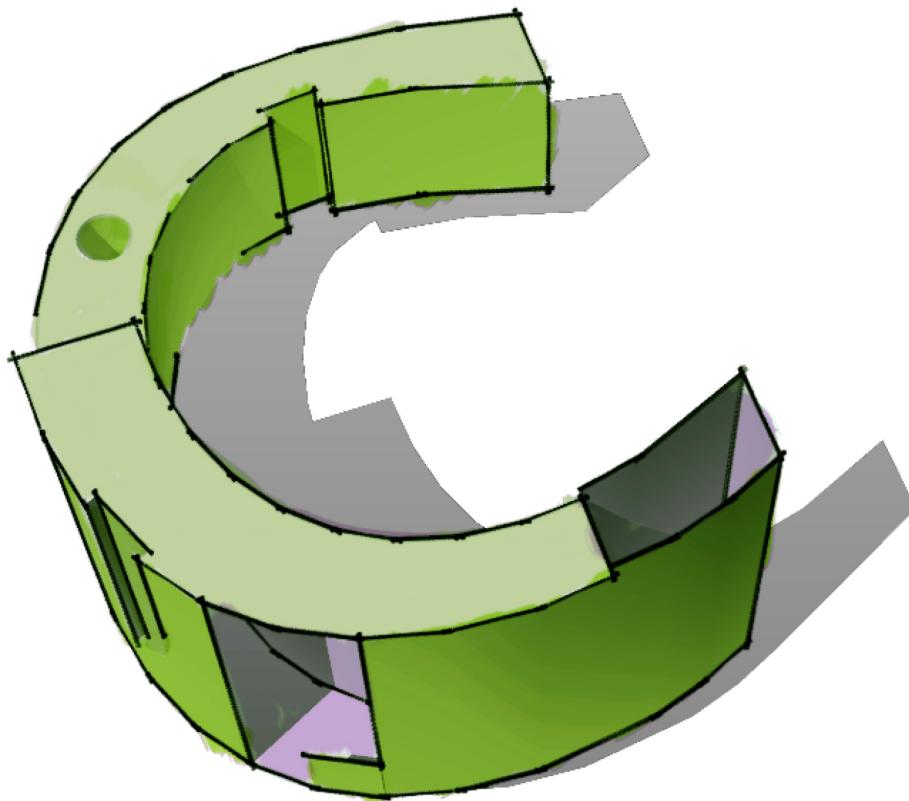


Escuela Técnica Superior de Ingenieros Industriales  
Universidad Politécnica de Madrid

# Fundamentos de programación en



Santiago Tapia Fernández  
Ángel García y Beltrán  
Raquel Martínez Fernández  
José Alberto Jaén  
Fco Javier del Álamo Lobo

3<sup>a</sup> edición



# Fundamentos de Programación en C

Santiago Tapia Fernández,  
Ángel García Beltrán,  
Raquel Martínez Fernández,  
José Alberto Jaén Gallego y  
Francisco Javier del Álamo Lobo

Departamento de Automática, Ingeniería Electrónica e Informática Industrial.  
ETSI Industriales - Universidad Politécnica de Madrid.

5 de febrero de 2015

Firmado digitalmente por TAPIA FERNANDEZ, SANTIAGO (FIRMA)  
Fecha: 2015.02.18 23:33:50 +01'00'

*Copyright 2015 por los autores.*

*Todos los derechos reservados.*

*ISBN y Depósito legal*

*Prohibida la copia o distribución de una parte o de la totalidad por cualquier medio sin la autorización expresa de los autores.*

*Copyrigth 2015 de la portada por Julio Martín Erro.*

*Se puede obtener una **edición en papel** en:*

*Sección de Publicaciones de la Escuela Técnica Superior de Ingenieros Industriales  
c/ José Gutierrez Abascal, 2  
28006 Madrid  
e-mail: [publicaciones@etsii.upm.es](mailto:publicaciones@etsii.upm.es)  
Tlf: 91 336 30 68 - Fax: 913363069*

*Se puede obtener una **copia electrónica** gratuita (no libre) en:*

*<http://aulaweb.etsii.upm.es/>  
Alumno Invitado de la asignatura PR55000007 (Fundamentos de Programación)*

A nuestras familias



## **Prologo a la primera edición**

Nuestro objetivo al escribir este libro es proporcionar una base de material de estudio y consulta a los alumnos de primero de los grados en ingeniería en tecnologías industriales y en ingeniería química de la ETSII-UPM que les permita aprender y adquirir las competencias asociadas a la programación de ordenadores en lenguaje C. Por su nivel y contenido, este material también podría usarse en otros grados para las primeras asignaturas de informática o fundamentos de programación en otras titulaciones.

El contenido del libro ha sido preparado por los profesores de la asignatura de “Fundamentos de Programación” y se ajusta a esa asignatura cuatrimestral de 6 créditos ETCS. Consta de tres partes de contenidos fundamentales (mínimo de la asignatura) y una parte extra de contenido adicional que puede servir al alumno para profundizar en algunos elementos del lenguaje y descubrir algunos nuevos.

Es aconsejable leer el libro en el orden natural en que se ha escrito. Se ha tratado, en la medida de lo posible, de que los capítulos solo usen conceptos tratados previamente. Para leer capítulos aislados sería aconsejable dominar los conceptos anteriores, bien a través del libro o bien por otras fuentes.

Hemos buscado un equilibrio entre teoría, ejemplos y problemas para dar al contenido un enfoque práctico y focalizado a la aplicación de los conceptos que se describen. En este sentido, recomendamos especialmente al lector o al alumno que utilice su ordenador para trabajar con los ejemplos y resolver los problemas planteados. Se ha evitado deliberadamente temas teóricos como el uso de pseudocódigo, el estudio de la complejidad de los algoritmos, la descripción pormenorizada de la sintaxis, etc. Nuestro enfoque se basa en una manera de entender el lenguaje C como un instrumento para la realización de programas de una forma sencilla y libre de errores, aunque no se usen todos los detalles y sutilezas del lenguaje C, esperamos haberlo conseguido.

Los autores.

Madrid, 21 de diciembre de 2011.



## **Prologo a la segunda edición**

Un libro para el estudio universitario debe ser algo vivo que evoluciona sin cambiar, que mejora sin llegar nunca a ser perfecto, que trata de resaltar cada vez más lo que es importante, pero no olvida los detalles. Fruto de este espíritu de revisión y mejora presentamos ahora esta segunda edición de un libro cuyo objetivo sigue siendo facilitar a los alumnos universitarios en general y a los alumnos de los grados impartidos en la Escuela Técnica Superior de Ingenieros Industriales en particular, el aprendizaje de esa habilidad que se llama de manera tan sencilla como “programar en C” y que, sin embargo, año a año, curso a curso, vemos que tiene conceptos, dificultades y sutilezas que hacen de ella todo un desafío.

De forma más concreta, hemos tratado de ajustar, en algunos casos reducir, en otros cambiar o ampliar el contenido de algunos capítulos para hacer el conjunto más coherente y completo sin sobrepasar la materia que se puede impartir en un semestre. Se ha retocado especialmente los capítulos de: funciones (para quitar toda referencia a la recursividad), cadenas alfanuméricas (para reducir el número de funciones trabajadas), archivos (para reducir el número de funciones trabajadas, pero estudiar con más detalle scanf y printf, también se incluye en este capítulo nociones de canales de datos) y variables dinámicas (se ha optado por incluir la pila y la cola como ejemplos representativos de las estructuras de datos dinámicas).

Por último queremos trasmitir nuestro compromiso de facilitar este libro, en su versión electrónica, de forma gratuita, pero no libre. Esto significa que haremos todo lo posible para que los estudiantes puedan obtener gratuitamente este libro, pero en modo alguno estamos autorizando su distribución o modificación libre. Esta restricción la imponemos para proteger la integridad de la obra y para garantizar que todas las personas que obtengan una copia de este libro puedan atribuir los méritos o fallos solo a los autores. Aspecto este último muy importante dado que el libro se utiliza como base para el estudio de una asignatura y su correspondiente evaluación. En este sentido, Santiago Tapia Fernández ha firmado la versión electrónica para garantizar la integridad del documento electrónico. No acepte una copia electrónica sin esa firma.

Los autores.

Madrid, 7 de febrero de 2014.



## Prologo a la tercera edición

Popularmente, no hay dos sin tres. Pero nos vamos acercando a una versión que sea estable. Nuestro principal objetivo al escribir este libro de Fundamentos de Programación en C es proporcionar un libro de referencia a los alumnos universitarios que cursan un grado de ingeniería en su primera asignatura sobre programación. Por supuesto, también y especialmente, a nuestros alumnos en la ETSII-UPM. En este sentido en esta tercera edición tratamos de plantear un índice de contenidos que esperamos que sea definitivo y que se adapte a un programa de horas de estudios y de docencia de una asignatura de programación de 6 Créditos y cuya docencia son 4 horas a la semana durante 14 semanas.

Para lograr este ajuste hemos introducido algunos cambios en la organización del contenido. Todo aquello que entendemos que no es fundamental y no cabe en el programa de la asignatura, pero nos parece interesante, lo hemos pasado a la última parte del libro: “Material Adicional”. Así pues, entendemos que las tres primeras partes del libro caben en el curso académico correspondiente y para nosotros mismos y para nuestros alumnos de la ETSII debe ser una referencia del contenido oficial de la asignatura. Naturalmente una referencia flexible porque en esta vida hay pocas cosas seguras y podemos decir que una cosa segura en ingeniería son los imprevistos.

En cuanto a los cambios en concreto: respecto de la última edición hemos reducido y reorganizado las secciones que tienen relación con la entrada-salida estándar, hemos recuperado la función fgets, pero hemos reducido los especificadores de formato obligatorios y el detalle con que se explican. Nos hemos fijado más en los casos de uso que en la sintaxis. Por ejemplo, para el especificador de cadenas ponemos el límite de caracteres leídos porque resulta crucial para utilizar las funciones de entrada correctamente. Sin embargo no ponemos la sintaxis completa del especificador de formato (que pasa al material adicional). De la misma manera entendemos que la función fgets es crucial en la lectura de archivos de texto por líneas, al ser éste un caso de uso que se presenta con bastante frecuencia en el desarrollo de software lo hemos añadido de nuevo.

El resto de cambios de importancia se encuentran en el capítulo de variables dinámicas. Hemos reducido las matrices dinámicas a una sola configuración y hemos dejado una única estructura dinámica basada en struct: las colas. Aunque, eso sí, hemos ampliado sus operaciones para introducir una nueva operación que la aleja del concepto teórico puro de cola y la acerca al de una lista. De nuevo este cambio se justifica porque entendemos que es un caso de uso interesante desde el punto de vista académico y de aplicación.

En el resto de capítulos no hay cambios significativos, solo la corrección de alguna errata.

Por último, no queremos dejar de resaltar nuestro compromiso de ofrecer a nuestros lectores la mejor calidad de contenido. Por eso, mantenemos nuestra estrategia respecto de los ejemplos que aparecen en este libro, todos ellos están compilados y el resultado de su ejecución se incluye de forma automática en las páginas de este libro. Y, de nuevo, la versión electrónica debe estar firmada digitalmente por el profesor Santiago Tapia para garantizar la integridad de este documento.

Los autores.

Madrid, 12 de febrero de 2015.



# Índice general

<b>I Introducción</b>	<b>1</b>
<b>1. Fundamentos de informática</b>	<b>3</b>
1.1. Conceptos fundamentales . . . . .	3
1.2. Breve historia de la informática . . . . .	4
1.3. El ordenador . . . . .	5
1.4. Lenguajes de programación . . . . .	7
1.4.1. Clasificación de los lenguajes . . . . .	8
1.4.2. Programas traductores . . . . .	9
1.5. Ciclo de vida del software . . . . .	9
1.6. Programas . . . . .	11
1.7. Codificación de la información y representación de datos . . . . .	11
1.7.1. Datos numéricos . . . . .	11
1.7.1.1. Formato BCD ( <i>Binary Code Decimal</i> ) . . . . .	12
1.7.1.2. Formato hexadecimal o de base 16 . . . . .	12
1.7.1.3. Formato de punto fijo (implícito) para números enteros (sin signo) . . . . .	12
1.7.1.4. Formato de punto fijo (implícito) para números enteros con signo . . . . .	13
1.7.1.5. Formato de punto fijo para fracciones . . . . .	13
1.7.1.6. Formato de punto o coma flotante . . . . .	15
1.7.2. Operaciones aritméticas . . . . .	15
1.7.3. Datos alfanuméricicos (caracteres) . . . . .	16
1.7.8. Lógica booleana . . . . .	17
<b>2. Introducción a la programación</b>	<b>21</b>
2.1. ¿Qué es el C? . . . . .	21
2.1.1. Origen del lenguaje C . . . . .	21
2.1.2. Versiones del lenguaje C . . . . .	21
2.1.3. El lenguaje estándar ANSI-C . . . . .	22
2.2. Edición, compilación y ejecución de un programa . . . . .	23
2.3. Herramientas de programación en C . . . . .	23
2.3.1. El editor de texto . . . . .	24
2.3.2. El compilador y herramientas asociadas . . . . .	24
2.4. Preparación del entorno de trabajo en Windows . . . . .	24
2.4.1. Instalación de herramientas . . . . .	24
2.4.2. Pasos para escribir programas sencillos . . . . .	25
2.5. Preparación del entorno de trabajo en GNU-Linux . . . . .	26
2.5.1. Instalación de herramientas . . . . .	26
2.5.2. Pasos para escribir programas sencillos . . . . .	27
2.6. El primer programa . . . . .	28
2.7. Un programa que hace algo . . . . .	29
2.8. Salida estándar básica en C . . . . .	30
2.9. El buen estilo de escribir en C . . . . .	31

<b>II Elementos básicos de programación</b>	<b>33</b>
<b>3. Primeros conceptos de programación</b>	<b>35</b>
3.1. Elementos básicos de un programa fuente . . . . .	35
3.2. Uso y significado de los elementos básicos del lenguaje . . . . .	37
3.3. Comandos del preprocesador . . . . .	39
3.4. Estructura de un programa . . . . .	39
<b>4. Datos y tipos básicos</b>	<b>43</b>
4.1. Datos . . . . .	43
4.2. Tipos de datos . . . . .	43
4.3. Los datos clasificados por posibilidad de modificación . . . . .	44
4.3.1. Constantes . . . . .	44
4.3.2. Variables . . . . .	44
4.4. Declaración de constantes y variables . . . . .	45
4.5. Tipos básicos de datos . . . . .	46
4.5.1. Tipos de datos numéricos . . . . .	46
4.5.2. Tipos de datos alfanuméricos . . . . .	47
4.5.3. Comentarios sobre tipos alfanuméricos . . . . .	47
4.5.4. El tamaño de los tipos básicos en C . . . . .	48
4.5.5. Selección del tipo de dato adecuado . . . . .	49
4.5.6. Conversión de datos . . . . .	50
4.6. Funciones de entrada y salida con formato para datos simples . . . . .	51
4.6.1. Aspectos Generales . . . . .	51
4.6.2. Especificadores de formato simple . . . . .	52
4.6.3. Uso simplificado de la función printf . . . . .	52
4.6.4. Uso simplificado de la función scanf . . . . .	53
4.7. Ejemplos . . . . .	54
<b>5. Expresiones y Asignaciones</b>	<b>57</b>
5.1. Expresiones . . . . .	57
5.2. Operadores . . . . .	58
5.3. Operador asignación . . . . .	58
5.4. Operadores aritméticos . . . . .	59
5.4.1. Operadores aritméticos combinados . . . . .	59
5.4.2. Operadores aritméticos incrementales . . . . .	59
5.5. Operadores de relación o comparación . . . . .	60
5.6. Operadores lógicos . . . . .	60
5.7. Operadores de bits . . . . .	61
5.8. Evaluación en cortocircuito en expresiones lógicas . . . . .	61
5.9. Operadores especiales . . . . .	61
5.10. Subexpresiones . . . . .	62
5.11. Prioridad de operadores . . . . .	62
5.12. Precauciones en la escritura de expresiones . . . . .	62
5.13. Ejemplos de expresiones . . . . .	63
<b>6. Sentencias selectivas o condicionales</b>	<b>69</b>
6.1. Sentencias o instrucciones . . . . .	69
6.2. La sentencia if . . . . .	70
6.3. Más ejemplos de uso de if . . . . .	70
6.4. Sentencias if anidadas . . . . .	72
6.5. La sentencia switch . . . . .	73
6.6. Ejemplo de uso de switch . . . . .	74

<b>7. Sentencias repetitivas o bucles</b>	<b>81</b>
7.1. Introducción . . . . .	81
7.2. El bucle while . . . . .	81
7.3. El bucle do-while . . . . .	82
7.4. El bucle for . . . . .	84
7.5. Combinación de distintos tipos de sentencias anidadas . . . . .	85
7.6. Un ejemplo de programación pobre . . . . .	86
7.7. Diferentes buenos estilos de programación . . . . .	87
<b>III Elementos avanzados de programación</b>	<b>95</b>
<b>8. Rutinas o funciones</b>	<b>97</b>
8.1. Concepto de rutina o función . . . . .	97
8.2. Definición de funciones . . . . .	97
8.3. Ejecución de funciones . . . . .	98
8.4. Parámetros de funciones . . . . .	99
8.5. Funciones de la librería estándar . . . . .	101
<b>9. Punteros</b>	<b>107</b>
9.1. Variables y direcciones de memoria . . . . .	107
9.2. ¿Qué es un puntero? . . . . .	108
9.3. Declaración de punteros . . . . .	108
9.3.1. Sintaxis . . . . .	108
9.3.2. Los punteros y el tipo al que apuntan . . . . .	109
9.3.3. Punteros genéricos . . . . .	109
9.4. Operaciones con punteros . . . . .	109
9.4.1. Operador dirección de . . . . .	110
9.4.2. Operador indirección . . . . .	110
9.4.3. Operaciones de relación e igualdad . . . . .	110
9.5. El valor nulo de direcciones de memoria: NULL . . . . .	111
9.6. Uso de punteros en C . . . . .	111
9.6.1. Referencias de variables estáticas . . . . .	111
9.6.2. Parámetros formales por referencia . . . . .	114
<b>10. Vectores y matrices</b>	<b>119</b>
10.1. Definición de vector . . . . .	119
10.2. Declaración de vectores . . . . .	119
10.3. Limitaciones de los vectores . . . . .	120
10.4. Estructura de los vectores . . . . .	120
10.5. El operador indexación . . . . .	121
10.6. Vectores y bucles . . . . .	121
10.7. Matrices . . . . .	122
10.8. Relación entre punteros y vectores . . . . .	123
10.8.1. Operador [ ] . . . . .	123
10.8.2. Aritmética de punteros . . . . .	123
10.8.3. Conversión implícita . . . . .	124
10.8.4. Declaración de parámetros formales . . . . .	125
10.8.5. Vectores de punteros y punteros a vector . . . . .	125
10.9. Paso de vectores a funciones . . . . .	125
10.10. Paso de matrices a funciones . . . . .	126
10.11. Ejemplos con vectores y matrices . . . . .	127
<b>11. Cadenas de caracteres</b>	<b>137</b>
11.1. Cadenas de caracteres . . . . .	137
11.2. Funciones con cadenas alfanuméricas . . . . .	138
11.3. Uso de funciones básicas con cadenas alfanuméricas . . . . .	139

<b>12. Estructuras</b>	<b>145</b>
12.1. Introducción . . . . .	145
12.2. El tipo <b>struct</b> o estructura . . . . .	145
12.3. Funciones y struct . . . . .	146
12.4. Arrays y struct . . . . .	147
12.5. Punteros a estructuras . . . . .	148
12.6. Uso del tipo estructura . . . . .	148
12.6.1. Agrupación de datos que definen un objeto del problema . . . . .	148
12.6.2. Programación orientada a objetos en C . . . . .	149
<b>13. Archivos y Canales de Datos Estándar</b>	<b>155</b>
13.1. Introducción . . . . .	155
13.2. Estructura y tipos de archivos . . . . .	156
13.3. Archivos y canales de datos . . . . .	156
13.4. Variables de tipo puntero a archivo en C . . . . .	157
13.5. Declaración de variables puntero a archivo . . . . .	157
13.6. Asociación de variables y archivos . . . . .	158
13.6.1. Apertura de archivos . . . . .	158
13.6.2. Cierre de archivos . . . . .	160
13.7. Uso de los canales estándares . . . . .	160
13.7.1. Direcciónamiento a un archivo . . . . .	161
13.7.2. Tuberías . . . . .	161
13.8. Escritura y lectura de datos con formato en archivos de texto . . . . .	161
13.8.1. El carácter <b>eoln</b> y la librería estándar de C . . . . .	161
13.8.2. Uso de la función <b>fprintf</b> . . . . .	162
13.8.3. Uso de la función <b>fscanf</b> . . . . .	163
13.8.4. Lectura de datos de un archivo de texto con <b>fscanf</b> . . . . .	164
13.8.5. Uso de la función <b>fgets</b> . . . . .	166
13.8.6. Datos con formato en archivos . . . . .	168
13.9. Otras operaciones . . . . .	169
<b>14. Estructuras de datos dinámicas</b>	<b>173</b>
14.1. Variables estáticas y dinámicas . . . . .	173
14.2. Punteros y variables dinámicas . . . . .	174
14.3. La función <b>malloc</b> . . . . .	174
14.4. La función <b>free</b> . . . . .	175
14.5. Puntero a puntero e indirección múltiple . . . . .	175
14.6. Punteros y variables dinámicas simples . . . . .	175
14.6.1. Creación y liberación de una variable dinámica . . . . .	175
14.6.2. Manipulación de la variable dinámica . . . . .	177
14.6.3. Múltiples punteros y variables dinámicas . . . . .	177
14.6.4. Asignación entre punteros . . . . .	178
14.7. Variables dinámicas de tipo vector . . . . .	180
14.8. Matrices dinámicas . . . . .	181
14.8.1. Matriz dinámica como vector de punteros . . . . .	182
14.9. Variables dinámicas de tipo <b>struct</b> . . . . .	183
14.9.1. Uso del tipo estructura para variables dinámicas . . . . .	183
14.9.2. Estructuras de datos dinámicas . . . . .	184
14.9.3. Implementación de una cola . . . . .	184
14.9.4. Ampliación de la implementación de una cola . . . . .	187
<b>IV Material adicional</b>	<b>195</b>
<b>15. Introducción</b>	<b>199</b>
15.1. El precompilador de C . . . . .	199
15.1.1. Descripción . . . . .	199
15.1.2. Comandos del preprocesador . . . . .	199

15.2. Programa de utilidad para compilación: make . . . . .	202
15.2.1. Instalación de GNU Make en Windows . . . . .	202
15.2.2. Make . . . . .	202
15.2.3. Uso de make para compilar y ejecutar . . . . .	202
15.2.4. Alternativas a make . . . . .	203
<b>16. Elementos Básicos de programación</b>	<b>205</b>
16.1. Otros tipos de datos . . . . .	205
16.1.1. El tipo enumerado . . . . .	205
16.1.2. Tipo size_t . . . . .	205
16.2. Tipos alfanuméricos y codificación de texto . . . . .	206
16.3. Funciones de entrada y salida estándar con formato . . . . .	207
16.3.1. Aspectos Generales . . . . .	207
16.3.2. Especificadores de formato simple . . . . .	207
16.3.3. Uso simplificado de la función printf . . . . .	208
16.3.4. Uso simplificado de la función scanf . . . . .	208
16.3.5. Uso avanzado de la función printf . . . . .	210
16.3.6. Uso avanzado de la función scanf . . . . .	211
16.4. Los conceptos left-value y right-value . . . . .	213
16.4.1. La asignación . . . . .	213
16.4.2. Left value y right value . . . . .	213
16.5. Operadores especiales . . . . .	214
16.5.1. Operadores especiales . . . . .	214
16.6. Sentencias de control especiales . . . . .	215
16.6.1. La sentencia break en bucles . . . . .	215
16.6.2. La sentencia continue . . . . .	216
16.6.3. La sentencia goto . . . . .	217
16.7. Funciones avanzadas de string.h . . . . .	219
16.7.1. Cuadros de funciones con string . . . . .	219
16.7.2. Ejemplos de uso de funciones avanzadas con cadenas alfanuméricas . . . . .	219
<b>17. Elementos Avanzados de programación</b>	<b>223</b>
17.1. Punteros y modelo de memoria . . . . .	223
17.2. Referencia: la semántica del concepto. . . . .	224
17.2.1. El tipo de los datos referidos . . . . .	224
17.3. Otras soluciones para implementar matrices dinámicas . . . . .	225
17.3.1. Matriz dinámica de filas de tamaño conocido y constante . . . . .	225
17.3.2. Matriz dinámica a partir de un vector unidimensional . . . . .	226
17.4. Recursividad . . . . .	228
17.5. Campos de Bits . . . . .	230
17.6. Tipos unión . . . . .	231
17.7. Definición e implementación de una pila . . . . .	232



# Parte I

## Introducción



# Capítulo 1

## Fundamentos de informática

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Definir conceptos básicos de la Informática (Conocimiento)
2. Describir la estructura de un ordenador (Conocimiento)
3. Definir el concepto de lenguaje de programación y describir el ciclo de vida del software (Conocimiento)
4. Describir los formatos de codificación de datos más significativos en informática (Conocimiento)
5. Convertir un valor numérico natural, entero o real a binario (y viceversa) (Aplicación)
6. Definir el álgebra booleana y describir sus operaciones básicas (Conocimiento)

### 1.1. Conceptos fundamentales

El término *informática* es un neologismo alemán (informatik, Karl Steinbuch, 1957) que viene de la contracción de *información automática* y que se ha tomado como la traducción de la expresión original inglesa *Computer Science*. La informática puede definirse como la disciplina que intenta dar un tratamiento científico a una serie de materias: el diseño de computadores, la programación de los mismos, el procesamiento automático de la información y el diseño de algoritmos para la resolución de problemas de diversa índole.

Un *computador* u *ordenador* es una máquina electrónica que bajo el control de uno o varios *programas*, de forma *automática* acepta y procesa *datos*, efectuando operaciones lógicas y aritméticas con ellos, y proporciona los resultados del proceso. De forma algo más vaga se podría decir que un ordenador es una máquina de propósito general destinada al procesamiento de información. El término *automática* indica que no hay intervención humana. Un *programa* es un conjunto ordenado de instrucciones que controlan el funcionamiento de un ordenador para llevar a cabo una tarea específica. Para Niklaus Wirth:

Programa = Algoritmos + Estructuras de Datos

Un *algoritmo* es una secuencia de reglas o pasos precisos que permiten obtener unos resultados a partir de unos datos. Por ejemplo: una receta. El algoritmo debe ser claro (no ambiguo) y finito en términos de recursos empleados. Asimismo debe ser independiente del lenguaje que se emplee para implementar cada paso. Un *programa* es la traducción de un algoritmo a un lenguaje de programación. Siguiendo el ejemplo anterior: la receta (algoritmo) puede traducirse al idioma francés o al español (lenguaje). Un programa puede estar formado por un conjunto de algoritmos, cada uno de los cuales, lleve a cabo una tarea específica. Por otro lado, existen diferentes *estructuras de datos* definidas por las relaciones o las formas en que pueden agruparse los datos que las constituyen. La estructura actual de los ordenadores no realiza distinción alguna entre datos y programas dada la naturaleza de éstos a la hora de almacenarlos, como se verá más adelante. En general y atendiendo al formato de codificación de los datos, los dispositivos que procesan datos pueden clasificarse en *analógicos* y *digitales*. Un dispositivo analógico almacena y procesa datos que están representados en términos de una variable continua, por ejemplo, un voltaje eléctrico. Son rápidos pero poco precisos y difíciles de programar. Sólo sirven para tareas sencillas. Un reloj de agujas o un termómetro de mercurio serían dispositivos analógicos. En un dispositivo digital la variable puede tomar un conjunto discreto de valores, por ejemplo, datos en formato binario: se emplean dispositivos o elementos con dos estados estables. Es mucho más flexible y preciso. La codificación viene condicionada por los elementos físicos empleados en la construcción del computador: un interruptor

abierto o cerrado, un transistor conduciendo o no, una cinta magnética magnetizada en un sentido o en otro, una señal eléctrica caracterizada por un pulso o por su ausencia... Cuando en lo sucesivo se hable de ordenador será a este tipo de dispositivo al que se haga referencia.

**Un dato es cualquier información codificada de forma que pueda ser aceptada y procesada por un computador.** Como se verá más adelante, será un valor que toma una variable para cada uno de los elementos de un conjunto. Para que pueda ser almacenada, transferida de una parte a otra y procesada por el ordenador, la codificación de la información se lleva a cabo en formato binario. Un dígito binario o **bit** (resultado de la contracción de la expresión *binary digit*) es la unidad más pequeña y fundamental de información. Un octeto (en inglés, *octet*) o **byte** es un conjunto de ocho bits y es la unidad práctica de información. Como el byte es, para la mayoría de usos, una unidad de representación o almacenamiento de información relativamente pequeña, se suelen emplear los múltiplos del byte mostrados en la Tabla 1.1 para indicar esta capacidad de representación o almacenamiento de la información.

Cuadro 1.1: Múltiplos para la representación de la información en binario

Unidad	Equivalencia
1 Kilobyte (o KB)	$= 2^{10}\text{bytes} = 1024\text{ bytes} \approx 10^3\text{bytes}$
1 Megabyte (o MB)	$= 2^{10}\text{KB} = 2^{20}\text{bytes} = 1048576\text{ bytes} \approx 10^6\text{bytes}$
1 Gigabyte (o GB)	$= 2^{10}\text{MB} = 2^{30}\text{bytes} = 1073741824\text{ bytes} \approx 10^9\text{bytes}$
1 Terabyte (o TB)	$= 2^{10}\text{GB} = 2^{40}\text{bytes} \approx 10^{12}\text{bytes}$
1 Petabyte (o PB)	$= 2^{10}\text{TB} = 2^{50}\text{bytes} \approx 10^{15}\text{bytes}$
1 Exabyte (o EB)	$= 2^{10}\text{PB} = 2^{60}\text{bytes} \approx 10^{18}\text{bytes}$
1 Zettabyte (o ZB)	$= 2^{10}\text{EB} = 2^{70}\text{bytes} \approx 10^{21}\text{bytes}$
1 Yottabyte (o YB)	$= 2^{10}\text{EB} = 2^{80}\text{bytes} \approx 10^{24}\text{bytes}$

## 1.2. Breve historia de la informática

La operación de calcular con los dedos de las manos en la Prehistoria cedió su nombre al *cálculo digital*, sinónimo también del término actual *informática*. Además este hecho provoca la adopción del sistema numérico *decimal* en la primitiva cultura egipcia. En la antigua Babilonia se emplea ya el sistema posicional mientras que el 0 es introducido en la India. El intento de automatizar y acelerar las operaciones de cálculo aritmético marca ya desde el principio de su historia la evolución de la informática. El *ábaco* es el primer instrumento mecánico de cálculo conocido y está considerado el primer prototipo de máquina de calcular. Las primeras versiones se construyen en China. El sistema decimal facilita la realización de sumas y restas pero no de productos y divisiones. La introducción de los logaritmos por *John Napier* en el siglo XVII permite la transformación de productos y divisiones en sumas y restas y *William Oughtred* al inventar la regla de cálculo también facilita todo este conjunto de operaciones aritméticas. Ya en el siglo XVII aparecen los primeros dispositivos mecánicos construidos a base de engranajes. En 1653, *Blaise Pascal* (1623-1662) diseña la primera calculadora mecánica automática que suma y resta para ayudar a su padre que era recaudador de impuestos. En 1673, *Gottfried Wilhem Leibniz* (1646-1716) construye una máquina con ruedas escalonadas para multiplicar y dividir. En 1822, *Charles Babbage* (1791-1871) construye una máquina diferencial, que calcula diferencias hasta con ocho decimales para corregir unas tablas de logaritmos llenas de errores. Como sus predecesoras todavía era una máquina de uso específico para resolver un tipo muy determinado de problema. En 1833 concibe la máquina analítica, es decir, una máquina de uso general o universal en la que, para realizar cada tarea concreta, hay que programarla. No llega a construir la máquina pero ya establece la diferencia entre la entrada de datos, el programa y la salida de resultados.

*George Boole* (1815-1864), a mediados del siglo XIX, desarrolla el álgebra booleana, base del diseño de circuitos en ordenadores digitales. En 1894, *Hollerith* (1860-1929) construye una máquina capaz de leer tarjetas perforadas que se emplea para mecanizar el censo de los Estados Unidos de América. *Alan Turing* (1912-1954), a principios del siglo XX, define un modelo matemático de computador consistente en un ordenador teórico, completamente abstracto, que pudiera llevar a cabo cualquier cálculo realizable por un ser humano.

En 1941 Konrad Zuge construye la computadora Z3 con 2300 relés y que era programable en sistema binario.

En 1944 aparece el primero de los dispositivos *electromecánicos* basado en relés: se le llamó *MARK-I*. Es una máquina universal construida en Harvard y financiada por IBM. Podía realizar todas las operaciones aritméticas básicas y calculaba logaritmos y senos. Medía 17 metros de longitud y 3 de altura. Contenía 760.000 piezas conectadas por 800 km de cables. Tardaba 10 segundos para multiplicar (comparar con los  $10^{-9}$ s que tarda el CRAY-2) debido a la gran cantidad de partes mecánicas y electromecánicas que poseía. El *MARK-II*, ya totalmente eléctrico, aparece en 1947.

Después surgen los *dispositivos eléctricos*, basados en válvulas de vacío: el *ENIAC* (contracción de la expresión *Electronic Numerical Integrator And Computer*) construido en Pennsylvania en 1946. Consumía 150.000 vatios, por lo que, disponía de un sistema de refrigeración y cuando se ponía en funcionamiento las luces de la ciudad tenían menos intensidad. Ocupaba 130 m<sup>2</sup> y pesaba 30 toneladas. Poseía 18.000 tubos de vacío, 70.000 resistencias y 10.000 condensadores, 6.000 interruptores y era 5.000 veces más rápido que el *MARK-I*. Empleaba todavía la aritmética decimal y no la binaria. Era capaz de hacer el trabajo de 100 ingenieros durante un año en dos horas. Aproximadamente cada dos días el ordenador dejaba de funcionar porque uno de sus 18.000 tubos se fundía. A modo de anécdota, el término *debugging*, empleado actualmente para hacer referencia a la depuración o detección de errores en un programa, se acuñó en el momento en que una polilla (*bug*) provocó un cortocircuito al introducirse entre los tubos de vacío de la máquina.

En 1945 *John von Neumann* (1903–1957) pone las bases de la arquitectura básica de un ordenador consistente en la definición de cuatro bloques interconectados: una unidad de control, una unidad aritmética, la memoria central y una unidad de entrada/salida. Las claves del éxito fueron la utilización de aritmética binaria y almacenamiento en memoria de las instrucciones del programa junto con los datos. Esto último aceleró las operaciones del ordenador. El uso de anillos magnéticos de ferrita resolvió la necesidad de utilizar elementos de memoria seguros, de bajo coste y rápidos. El primer ordenador de este tipo fue el *Torbellino*, utilizado por su rapidez en el control del tráfico aéreo.

Con el desarrollo de los *transistores* (contracción de la expresión *transfer resistor*) surgen los dispositivos electrónicos. Como el volumen físico y el precio se reducen considerablemente, ya se comercializa la serie 7.000 de IBM. Paralelamente, se aumenta la seguridad y la velocidad de proceso. Mas tarde la concentración de elementos electrónicos en circuitos integrados o *chips* inicia una nueva generación de dispositivos electrónicos. Aparecen los primeros miniordenadores (PDP-1). La siguiente generación se basa en la utilización de arquitecturas masivamente paralelas: Hipercubo de Intel, *Thinking Machines*,... Suelen distinguirse diversas generaciones de computadores dependiendo de los elementos físicos constituyentes:

- 1<sup>a</sup> Generación: Caracterizada por la tecnología del **relé**, el **tubo de vacío** y los **núcleos de ferrita** (1945-1958). La velocidad de proceso se mide en milésimas de segundo (ms). Disipan una gran cantidad de energía calorífica. Los trabajos se realizan uno a uno (monoprogramación) y no existe sistema operativo. Se programa en lenguaje máquina directamente y se utiliza la aritmética de punto fijo.
- 2<sup>a</sup> Generación: Caracterizada por la utilización de **transistores** y **diodos** (1957-1968). La velocidad de proceso se mide en millonésimas de segundo (μs). Se utilizan núcleos de ferrita para las memorias. Aparecen los primeros lenguajes simbólicos y los sistemas operativos y se empieza a utilizar la aritmética de coma flotante.
- 3<sup>a</sup> Generación: Caracterizada por la integración de circuitos o **chips** (1964-1971). La velocidad se mide en nanosegundos (ns). Aparecen los lenguajes de alto nivel, la multiprogramación, el multiproceso y las bases de datos.
- 4<sup>a</sup> Generación: Caracterizada por la **integración masiva de circuitos** (1972-1980s). El tiempo de proceso se mide en pico segundos. Surgen los lenguajes de cuarta generación, la inteligencia artificial y los sistemas expertos.
- 5<sup>a</sup> Generación: Caracterizada por un aumento extraordinario de las prestaciones y la velocidad de los microprocesadores para ordenadores personales (1980s-2005). En la década de los años 80 aparecen la mayoría de los procesadores *Intel* de la familia x86, desde el 8086 (de 1978) al 80486 (de 1989) para culminar con el primer microprocesador Pentium en 1993 y el Pentium IV en 2005. Todos estos años se caracterizaron por el aumento vertiginoso de la velocidad de los microprocesadores (hasta aproximadamente 3,8 GHz), por el paso de arquitecturas de 16 a 32 bits y, en general, de las prestaciones de los ordenadores personales.
- 6<sup>a</sup> Generación: Caracterizada por el abandono de la carrera de prestaciones de los microprocesadores (que incluso bajan de características) en favor del aumento del número de núcleos presentes en cada microprocesador (2005 - presente). En esta etapa se ha alcanzado el tope tecnológico de prestaciones de velocidad para los microprocesadores y, en vista de ello, se opta por aumentar el número de núcleos en cada microprocesador, lo que equivale en la práctica a tener varios microprocesadores en un solo chip. Esta generación coincide aproximadamente con la familia de los ordenadores *multicore*, desde el *Dual-Core* hasta los últimos i7 con 4 núcleos reales y 8 núcleos virtuales.

### 1.3. El ordenador

Un ordenador se compone de hardware y software.

Cuadro 1.2: Generaciones de ordenadores en función del número de circuitos integrados por mm<sup>2</sup>

Año	Integración	Acrónimo	Circuitos/mm <sup>2</sup>
1960	<i>Short Scale Integration</i>	SSI	<100
1966	<i>Medium Scale Integration</i>	MSI	100-1000
1969	<i>Large Scale Integration</i>	LSI	1000-10000
1975	<i>Very Large Scale Integration</i>	VLSI	10000 -100000
1985	<i>Ultra Large Scale Integration</i>	ULSI	100000-1000000
1990	<i>Giga Large Scale Integration</i>	GLSI	>1000000

El *hardware* es el soporte físico del ordenador, es decir, los componentes físicos que lo constituyen: conjunto de circuitos electrónicos, cables, carcasa,... Por ejemplo, el diseño de circuitos y la electrónica son disciplinas relacionadas con el hardware.

El *software* es el soporte lógico, es decir, los programas que dirigen el funcionamiento del ordenador. Los programas van a ser el principal objeto de estudio de esta obra. La construcción de algoritmos, la estructura de datos y la metodología para desarrollar programas son varios ejemplos de disciplinas relacionadas con el software.

Los ordenadores pueden clasificarse de acuerdo con sus capacidades en:

1. Microcomputadores: Ordenadores personales con dispositivos de almacenamiento interno y capacidad limitada (Mb, Gb).
2. Minicomputadores: Multiusuario (decenas), almacenamiento medio (Gb, Tb).
3. *Mainframes*: Mayores computadores, decenas o cientos de terminales, almacenamiento masivo,...
4. Supercomputadores: Desarrollo reciente, muy rápidos: optimizados para cálculo científico.
5. *Clusters*: Agrupación de ordenadores para funcionar coordinadamente en la resolución de cálculos científicos.

### Estructura física de un ordenador

Cualquiera de los tipos de ordenador anteriores se compone de distintas partes, cada una de las cuales lleva a cabo una labor específica. El *procesador* o unidad central de proceso (CPU) es el dispositivo donde se lleva a cabo el tratamiento de los datos. Los ordenadores de tamaño medio o grande pueden tener más de uno. Las funciones básicas del procesador son:

1. Manipulación de datos: operaciones elementales de tipo aritmético (sumas, desplazamientos de bits...) y lógico (operaciones del álgebra de Boole) llevadas a cabo por la *Unidad Aritmético-Lógica* (ALU)
2. Control de la ejecución de los programas: transferencias de datos con la memoria, control de la secuencia en la que se ejecuta el programa,... Se lleva a cabo en la *Unidad de Control* (UC). Contiene un reloj o generador de pulsos que sincroniza todas las operaciones elementales del ordenador.
3. Adicionalmente, dispone de una memoria propia limitada para el almacenamiento de resultados intermedios, información necesaria para el control de la ejecución del programa,...

Las principales características de los procesadores son:

1. Tamaño en bits de los datos que maneja (*registros internos*).
2. Tamaño de los datos con los que se comunica al exterior.
3. Velocidad del reloj de sincronización (tiempo empleado en un ciclo elemental de máquina) o su inversa, la frecuencia de ciclo que se mide en kiloherzios (KHz), Megaherzios (MHz) o Gigaherzios (GHz).
4. Variedad del conjunto de instrucciones que es capaz de realizar.
5. Rendimiento global del procesador: número de instrucciones capaz de realizar por segundo (*MIPS*) o número de operaciones en coma flotante por segundo (*MFLOPS*).

Existen distintos tipos de procesadores que pueden clasificarse en:

1. CISC (*Complete Instruction Set Computers*): los habitualmente empleados en ordenadores personales.

2. RISC (*Reduced Instruction Set Computers*): disponen de un limitado número de instrucciones pero son capaces de ejecutarlas mucho más rápidamente que un procesador normal.
3. *Transputers*: Son computadores en un chip ya que incluyen memoria y controladores de puertos. Interesantes para el diseño de computadores masivamente paralelos.
4. Procesadores vectoriales: Capaces de realizar la misma operación simultáneamente sobre un cierto número de datos. Muy adecuados para operaciones matriciales.
5. Procesadores *pipeline*: Capaces de procesar varias instrucciones simultáneamente, no esperando a que cada instrucción se haya ejecutado completamente, sino comenzando con la siguiente cada vez que una parte del decodificador de instrucciones quede libre.

La *memoria*, principal, central o interna, es el dispositivo formado por circuitos electrónicos integrados donde se almacena, se carga, información -instrucciones y datos- antes, durante y después de su tratamiento por su procesador. Las características principales de la memoria son su tamaño (capacidad de almacenamiento de datos) y el tiempo de acceso (tiempo necesario para recuperar un dato). Para acceder a una información determinada, se emplea su dirección o posición en la memoria. La memoria puede clasificarse en:

1. Memoria ROM o *Read Only Memory* (memoria de sólo lectura). Se emplea para almacenar datos y programas necesarios para la operación del ordenador. Es una memoria no volátil, es decir, los datos no se pierden cuando se apaga el ordenador.
2. Memoria RAM o *Random Access Memory* (literalmente, memoria de acceso aleatorio). Permite la lectura y escritura de datos y es la empleada normalmente para el almacenamiento de la información que se está tratando en un momento dado. Es volátil, es decir, los datos se pierden cuando se apaga el ordenador.
3. Memoria caché: Parte de la memoria RAM normalmente de acceso más rápido donde se almacenan datos de uso más frecuente. En las arquitecturas más modernas la gestión de la memoria caché es una de las características más importantes para mejorar la velocidad de los microprocesadores.
4. Memoria virtual: Memoria RAM residente físicamente en los sistemas de almacenamiento masivo y gestionada por el sistema operativo.

Las *memorias auxiliares* o *sistemas de almacenamiento masivo* son dispositivos donde se pueden almacenar grandes volúmenes de información (de acceso más infrecuente). Son no volátiles y de acceso más lento que la memoria interna. Su tamaño actual va desde centenares de MB (por ejemplo, un CD-ROM) a varios TB. Existen de muy diferentes tipos: cintas magnéticas, discos magnéticos, CD-ROM, DVD, discos ópticos, *pendrives* o discos USB...

Los *buses* permiten la transmisión de datos entre las distintas partes del hardware mientras que los dispositivos de entrada y salida de datos, junto con los elementos de interconexión, permiten el intercambio de información con el entorno (incluyendo otros sistemas). Existen distintos tipos: terminales (constan de pantalla y teclado), impresoras, *plotters* o trazadores, escaners, tarjetas de red, dispositivos de adquisición de datos,...

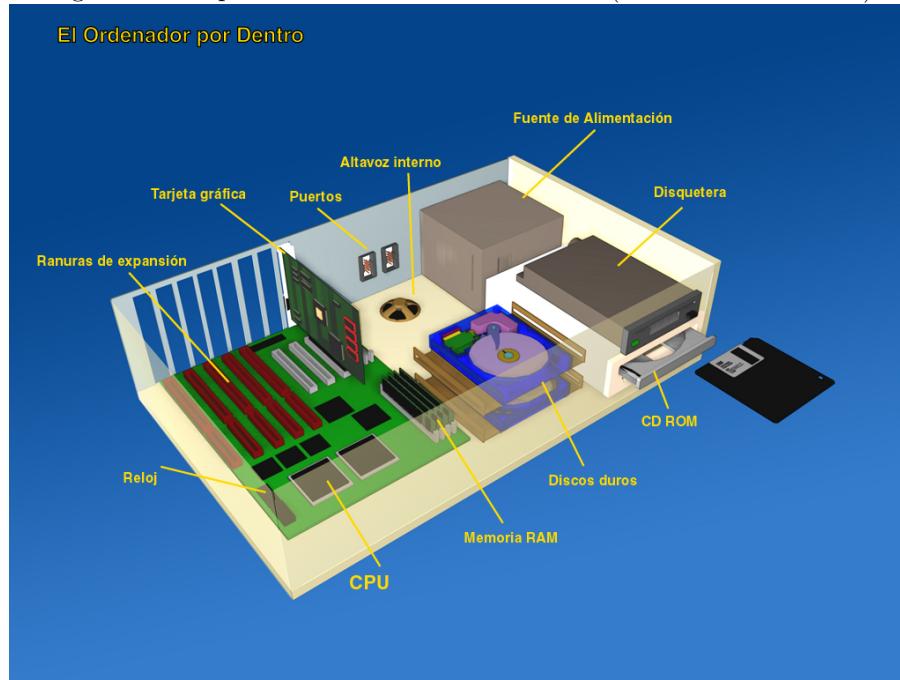
Los dispositivos de *entrada de datos* transforman las informaciones de entrada en señales binarias de naturaleza eléctrica. Un ordenador puede tener diferentes dispositivos de entrada: el teclado, un ratón, un escáner, una lectora de tarjetas de crédito.

Los dispositivos de *salida de datos* presentan los resultados de los programas ejecutados por el ordenador, transformando las señales eléctricas binarias en imágenes visualizadas o dibujadas o caracteres impresos o reproducidos acústicamente. Ejemplos de dispositivos de salida de datos son: un monitor, una impresora o un *plotter*. Otros dispositivos como las tarjetas de red y los modems son, indistintamente, dispositivos de entrada y salida de datos. Finalmente, las ranuras de expansión permiten la conexión del ordenador con otros dispositivos externos. En la figura se muestra un esquema del interior de un ordenador personal de sobremesa.

## 1.4. Lenguajes de programación

Un *lenguaje de programación* es un medio de comunicación entre el usuario y el ordenador formado por un conjunto de símbolos elementales (*alfabeto*), palabras (*vocabulario*) y reglas (*semántica y sintaxis*). Con un lenguaje se construyen cada una de las *órdenes* o *instrucciones* para el ordenador de la secuencia de instrucciones que constituyen un programa.

Figura 1.1: Esquema funcional de un ordenador (cortesía de J. Martín)



#### 1.4.1. Clasificación de los lenguajes

Existen muchos lenguajes de programación que pueden clasificarse en cuatro grandes grupos:

**Lenguajes máquina:** son códigos binarios de instrucciones, es decir, sucesiones de ceros y unos (1100 0010 1010 0101 ...) que componen un repertorio reducido de operaciones. Es el único lenguaje que entienden los circuitos electrónicos que forman la Unidad de Control del procesador y cada procesador tiene un lenguaje máquina específico. La programación con un lenguaje máquina es muy laboriosa para el programador a todos los niveles: escritura, interpretación, depuración y mantenimiento.

**Lenguajes ensambladores:** son códigos de instrucciones formados por grupos de palabras mnemotécnicas (LOAD = almacenar, ADD = sumar, CMP = comparar,...). Estos lenguajes sustituyen los códigos binarios por códigos mnemotécnicos más fáciles de identificar y recordar. Cercanos a los lenguajes máquina, también dependen de cada procesador. Los programas no son directamente interpretados por el procesador, necesitándose un programa traductor llamado *ensamblador*. A los lenguajes máquina y ensamblador también se les conoce como lenguajes de *bajo nivel*. Tienen como ventaja el que los programas ocupan poco espacio en memoria y tienen un tiempo mínimo de ejecución. Como inconvenientes principales estarían el que dependen del procesador, es necesario un conocimiento muy profundo del hardware y las operaciones que describen cualquier proceso son muy elementales. Se emplean especialmente en microprocesadores industriales.

**Lenguajes de alto nivel:** Son universales: no dependen del procesador ni de la estructura interna de cada ordenador. No se necesita conocer el hardware específico de dicha máquina. Están más próximos al lenguaje natural y son claros, simples, eficientes y legibles. Sus instrucciones equivalen a múltiples instrucciones en lenguaje máquina. Tiene el problema de que los programas son más lentos de ejecución y necesitan ser traducidos a lenguaje máquina para que el procesador los entienda. Como ejemplos de lenguajes de alto nivel pueden destacarse los siguientes: FORTRAN, ALGOL, COBOL, BASIC, PASCAL, C, LISP, PHP, SQL...

**Lenguajes orientados a objetos y librerías de sistema.** Los lenguajes orientados a objetos añaden a las características propias de los lenguajes de alto nivel un nuevo elemento de programación, la *clase*, que permite un mejor diseño y estructuración del programa. Normalmente se usan para grandes proyectos de programación. Ejemplos de lenguajes orientados a objetos son ADA, C++, Java, etc. Algunos incluyen librerías de clases que proporcionan funcionalidades propias del sistema: interfaz de usuario gráfica (ventanas), comunicaciones sobre Internet, servicios de encriptación y muchas más. Dos ejemplos significativos de estos lenguajes orientados a objetos con librería de sistema son Java y la familia de lenguajes *dot Net* (punto Net), especialmente C# (C almoradilla ó C sharp). Una extensión significativa de los lenguajes orientados a objetos es la programación genérica (basada en *templates*)

ó plantillas) y que aumenta aún más el nivel de abstracción de los programas. Ejemplos de programación genérica son los tipos genéricos de Java o las librerías STL (*Standard Template Library*) y Boost de C++.

### 1.4.2. Programas traductores

Como los procesadores sólo entienden su lenguaje máquina se necesitan programas para traducir los programas escritos a otros lenguajes a lenguaje máquina. Estos programas traductores pueden clasificarse en:

1. *Ensambladores*: traducen de lenguaje ensamblador a lenguaje máquina.
2. *Compiladores*: traducen las instrucciones de un programa escrito en un lenguaje de alto nivel a un programa escrito en lenguaje máquina. El programa fuente es el programa escrito en lenguaje de alto nivel. El programa objeto es el programa ya traducido en lenguaje máquina. En muchos casos el resultado de la compilación es sólo una parte o módulo del programa ejecutable que posteriormente construirá el enlazador o montador (*linker*) a base de dichos módulos e incorporando código de las librerías o unidades utilizadas.
3. *Intérpretes*: traducen de un lenguaje de alto nivel a lenguaje máquina pero ejecutando cada instrucción conforme se va traduciendo todo el programa hasta su última instrucción. A diferencia de los compiladores, los intérpretes no generan un programa objeto.
4. *Preprocesadores*: traducen un programa escrito en un lenguaje de alto nivel a otro escrito en otro lenguaje de alto nivel. También realizan operaciones de sustitución, inclusión de archivos, etc. En general, realizan operaciones de procesamiento de texto.

## 1.5. Ciclo de vida del software

Se entiende por *ciclo de vida* de una aplicación software al periodo que comprende desde el momento en que se decide desarrollar el programa hasta que éste deja de estar en funcionamiento. Un producto o aplicación software trata de resolver un problema o necesidad y está formado por uno o varios programas, junto con sus datos y la documentación asociada. Su desarrollo no se reduce a codificar un programa en un determinado lenguaje de programación. La *ingeniería del software* es la disciplina que estudia los aspectos metodológicos relacionados con el diseño, desarrollo y mantenimiento de programas. En la secuencia de desarrollo de un programa es necesario tener en cuenta las siguientes fases:

1. *Análisis del problema*. Se analiza de forma clara qué problema debe resolver el programa. Se determinan cuáles deben ser los datos de entrada y cuáles son las de salida. Como resultado de esta fase se obtienen dos documentos: la Definición del Problema y el Documento de Requisitos. En esta fase deben intervenir tanto el usuario final del programa como el informático o grupo de informáticos especialistas.
2. *Diseño y Arquitectura*. En esta fase se decide qué estructuras de datos van a emplearse (diseño de las estructuras de datos) y cómo se van a ejecutar las acciones (diseño de los procesos). Como resultado se obtendrá el pseudocódigo u organigrama del programa. La responsabilidad de esta fase es de los analistas informáticos junto con los usuarios finales del programa.
3. *Codificación*. Se traducen o codifican los algoritmos y estructuras diseñadas anteriormente al lenguaje de programación seleccionado. Como resultado se obtendrá el o los programas fuente que, una vez compilados y enlazados, darán lugar al programa ejecutable.
4. *Verificación y Validación*. Se comprueba que el programa funciona correctamente (verificación) y que cumple las especificaciones de requisitos definidos en la Fase 1 (validación). Para llevar a cabo estos objetivos se realizarán pruebas de cada módulo o unidad, de la integración de todos los módulos y de validación de las especificaciones funcionales.
5. *Instalación y Depuración*. El programa se instala en el sistema informático en el que debe funcionar y se realizan las correcciones necesarias. Cuando es conveniente se elimina el código redundante.
6. *Explotación y Mantenimiento*. Una vez a disposición del usuario, se mantiene un periodo de asistencia técnica o mantenimiento y se solucionan problemas, errores o dudas que aparezcan, se readapta a nuevas especificaciones o circunstancias o, simplemente, se mejora algún aspecto del programa.
7. *Retirada*. Con el paso del tiempo el programa puede quedar obsoleto, eliminándose del sistema en el que fue instalado.

Estas fases pueden desarrollarse de forma secuencial o solapada. Por otro lado, las actividades de planificación y documentación (descripción detallada de los resultados de cada fase) forman parte de todas ellas. En concreto, la documentación resulta especialmente importante para facilitar el mantenimiento del software.

Dentro del desarrollo de software resultan particularmente importantes las herramientas que dan soporte al desarrollo de aplicaciones, se pueden indicar, entre otras, las siguientes familias de herramientas:

1. Herramientas de análisis y diseño. Normalmente, cuando se habla de diseño se habla de UML (*Unified Modeling Language*), un estándar que permite representar la estructura y diseño de un programa (el modelo). Existen distintas herramientas tanto de edición de diagramas UML como de traducción a código fuente.
2. Gestión de versiones (*SCM: source control management*). Son herramientas que permiten guardar la historia de modificación de los archivos de código fuente. Son especialmente útiles en el desarrollo de grandes aplicaciones y en trabajo en grupo. Ejemplos de gestores de versiones son *Subversion* (acrónimo: SVN), *Git* o *Mercurial*.
3. Analizadores de rendimiento (*Profilers*). Son herramientas que permiten evaluar el comportamiento de un programa en cuanto a su rendimiento. En general, se busca mejorar la velocidad del programa o el uso de memoria.
4. Depuradores (*Debuggers*). Herramientas para ejecutar el programa paso a paso. Permiten introducir puntos de parada (*break points*) para parar la ejecución en un punto concreto del programa y estudiar el estado del programa a través de los valores de las variables del mismo.
5. Refactorización (*Refactoring*). Estas herramientas de análisis de código permiten detectar código ineficiente o mal diseñado, así como modificar el diseño de una aplicación, por ejemplo, sustituyendo el nombre de algún elemento en todo el código donde se use.
6. Gestor de pruebas (*Testers*). Estas herramientas permiten escribir y gestionar casos de prueba automáticos para los programas. Generalmente prueban pequeños módulos de la aplicación.
7. Documentadores. Estas herramientas permiten obtener archivos de documentación de un software a través del análisis de la estructura del mismo y de comentarios que se escriben en los archivos de código fuente. La documentación se puede obtener en distintos formatos, pero es corriente que se genere HTML. Ejemplos significativos son el *Javadoc* en Java y *Doxxygen* para C, C++.
8. Gestión de proyectos. Hay una variedad extraordinaria de formas de gestionar proyectos de software, para algunas de ellas se han construido herramientas que permiten analizar el estado de desarrollo, la orientación del trabajo, el reparto de tareas, etc. Muchos de estos sistemas se basan en el uso de *tickets* o similares.
9. Gestor de compilación. En proyectos de gran alcance y envergadura suele ser necesario suministrar la forma en que se debe compilar (construir el proyecto) hasta obtener las librerías y programas objetivo. La forma de construir (compilar y enlazar) un proyecto viene determinada por las dependencias entre los distintos archivos de código fuente que constituyen el proyecto y por la eventual ejecución de otras etapas de preproceso o post-proceso. El gestor clásico de compilación es un programa denominado *make* y que hoy en día todavía se encuentra en algunos proyectos. Hay otros gestor más modernos entre los que destaca *CMake*, *Ant* o *Gradle* (estos dos últimos usados sobre todo para proyectos de Java y/o Android). Para proyectos en C/C++ existe un desarrollo muy interesante, llamado *biicode* (<http://www.biicode.com/>), que permite la descarga directa automática de las dependencias externas que insertemos en nuestro código fuente.
10. Sitios de proyectos de código libre o abierto (*opensource*). Aunque estrictamente no son herramientas de desarrollo es conveniente conocer que existen Webs donde se da soporte al desarrollo de proyectos de software libre de una forma u otra. Por su relevancia *sourceforge*, <http://sourceforge.net/>, y *github*, <https://github.com/>, son ejemplos significativos de estos sitios.
11. IDE (*Integrated Development Environment* o Entorno de desarrollo integrado). Son aplicaciones que permiten escribir y probar un programa directamente sobre la misma aplicación. Generalmente integran otras herramientas: *profiler*, *debugger*, refactorizadores de código, gestión de versiones, y tienen funciones avanzadas de edición de código, como el autorrelleno. También son capaces de gestionar código en distintos lenguajes. Ejemplos de IDEs: *Codeblocks*, *Geany*, *Anjuta*, *Eclipse*, *Netbeans*.

## 1.6. Programas

Como ya se ha comentado anteriormente, un *programa* es un conjunto de instrucciones que controla el funcionamiento de un ordenador para llevar a cabo una tarea específica. Los programas pueden clasificarse en programas de sistema operativo y programas de aplicación.

Un *sistema operativo* está formado por un programa o un conjunto de programas que gestiona el funcionamiento del hardware, controla la ejecución de los programas y procesa las órdenes del usuario al ordenador. Ejemplos de sistemas operativos son: Windows 7, Linux, MS-DOS, UNIX, Windows XP, Mac/OS, VMS, ... Los sistemas operativos emplean un lenguaje de control específico cuyas instrucciones se denominan comandos. Por otro lado los sistemas operativos sirven de soporte a los programas de aplicación. Al poner en funcionamiento el ordenador el sistema operativo se carga en la memoria RAM y mas concretamente en las direcciones iniciales de memoria. Las principales funciones de un sistema operativo son:

1. Gestión de disposición de E/S (escritura, lectura, control)
2. Ejecución de programas (carga, ejecución, interrupciones)
3. Gestión de archivos (creación, borrado, lectura)
4. Detección de errores (en memoria, en periféricos)
5. Asignación de recursos (CPU, memoria)
6. Protección (confidencialidad, no interferencia)
7. Contabilidad (carga de los recursos)

Los programas de aplicación sirven para que el ordenador lleve a cabo una tarea específica. Pueden clasificarse en: paquetes integrados de software (Microsoft Office, Works, OpenOffice...), editores o procesadores de texto (Word, WordPerfect, LyX...), hojas de cálculo (Excel, Lotus 1-2-3,...), aplicaciones de diseño gráfico (AutoCad, Microstation,...), gestores de bases de datos (Access, Oracle,...), etc.

Una *base de datos* (BD) es un conjunto de datos relacionados entre sí y almacenados de forma sistemática para facilitar su uso posterior. Existen programas denominados sistemas gestores de bases de datos (SGBD), que permiten almacenar y posteriormente acceder a los datos, que pueden ser de diferentes tipos, de forma rápida y estructurada.

## 1.7. Codificación de la información y representación de datos

Los *datos* son abstracciones de la realidad necesarias para la resolución de una tarea. Deben estar representados de forma que sean procesables por un ordenador. Un *código* es un sistema de signos y reglas que permite representar información. Cuando se utiliza un código para guardar, representar e interpretar información se dice que está codificada. Los circuitos electrónicos que componen los ordenadores emplean el formato binario para representar y procesar la información, lo que implica la codificación correspondiente de ésta. Por lo tanto los datos aparecen representados mediante secuencias de ceros y de unos. Si bien el *bit* es la unidad más pequeña de información, el *byte* es la unidad usual de información. Un *byte* es una secuencia de ocho bits. Como cada uno de estos bits puede tomar dos valores, un byte podrá tomar  $2^8 = 256$  valores distintos. El ordenador puede trabajar con uno o varios bits a la vez, normalmente 8 (1 byte), 16 (2 bytes), 32 (4 bytes) ó 64 (8 bytes). Una *palabra* es una cadena finita de *bits* que puede ser manipulada de forma simultánea como un conjunto por un ordenador en una operación. El *ancho o longitud de palabra* es el número de *bits* de la palabra. Los *archivos* o *ficheros* son estructuras de datos que residen en la memoria secundaria. Son conjuntos de informaciones estructuradas del mismo tipo y de un tamaño indeterminado. Los archivos pueden ser de datos o de programas.

En general, pueden considerarse dos tipos de datos: los datos *numéricos* y los datos *alfanuméricicos* o *caracteres*. Una de las diferencias prácticas entre ellos es que ciertas operaciones están definidas para un tipo de dato y no para el otro. Tanto para codificar datos numéricos como para caracteres se emplean varios formatos alternativos.

### 1.7.1. Datos numéricos

Dependiendo de la naturaleza de los valores que se quieran representar, los datos numéricos pueden codificarse empleando diferentes formatos. En general, estos formatos siguen una notación posicional ponderada normalizada, es decir, un esquema de codificación que permite representar números según una expresión aritmética donde se consideran los valores numéricos preasignados a un alfabeto de dígitos y la posición que ocupan éstos en la representación.

Cuadro 1.3: Número de bits y posibles combinaciones

Número de bits	Nº combinaciones
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8 (1 byte)	$2^8 = 256$
10	$2^{10} = 1024$
15	$2^{15} = 32768$
16 (2 bytes)	$2^{16} = 65536$
20	$2^{20} = 1048576$
32 (4 bytes)	$2^{32} = 4294967296$
64 (8 bytes)	$2^{64} \approx 1.8447 \cdot 10^{19}$

#### 1.7.1.1. Formato BCD (*Binary Code Decimal*)

Con el formato Decimal Codificado a Binario (BCD) se codifican los diez dígitos del sistema decimal por sus correspondientes representaciones binarias. Se necesitan cuatro bits para cada dígito decimal. Por ejemplo, el número 135 en formato BCD es 0001 0011 0101. El principal inconveniente de este formato es que desaprovecha seis combinaciones del total de posibilidades de representación.

#### 1.7.1.2. Formato hexadecimal o de base 16

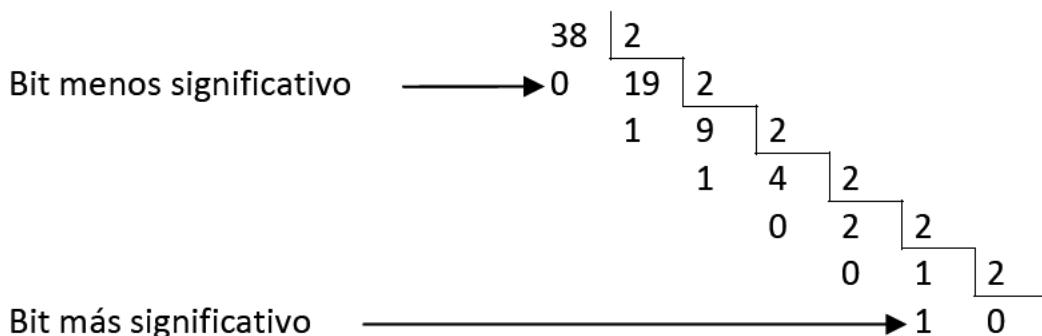
Un dígito hexadecimal, expresado mediante uno de los diez dígitos decimales (0-9) o una de las seis primeras letras del alfabeto (A-F), representa cuatro dígitos binarios. Con cuatro dígitos binarios pueden obtenerse  $2^4 = 16$  combinaciones distintas. Este formato suele emplearse como código intermedio: es un compromiso razonable entre lo cercano a la máquina y lo práctico para el usuario. Por ejemplo, la secuencia binaria 0100 1010 0010 1111 equivale en formato hexadecimal a 4A2F.

#### 1.7.1.3. Formato de punto fijo (implícito) para números enteros (sin signo)

Para los enteros sin signo se emplea la representación en base 2 del número natural. Es una representación directa, es decir, cada bit indica el coeficiente de la expresión de dicho número como suma de potencias de base 2 con exponente creciente desde 0 según su posición de derecha a izquierda. Por ejemplo, la siguiente secuencia de 6 bits:

$$100110 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 0 + 0 + 4 + 2 + 0 = 38$$

En general,  $n$  bits permiten representar  $2^n$  números naturales distintos. El intervalo de representación es de 0 a  $2^{n-1}$ . También se le denomina sistema de numeración en base 2, **binario natural o binario puro**. Si se quiere transformar un entero decimal a binario, se divide sucesivamente el valor y los correspondientes cocientes por 2 hasta obtener un cociente nulo. Los restos son los dígitos binarios buscados. El último resto corresponde al bit más significativo (el situado más a la izquierda) y el primer resto corresponde al bit menos significativo (el situado más a la derecha). Por ejemplo, para transformar el valor entero decimal 38 a binario:



#### 1.7.1.4. Formato de punto fijo (implícito) para números enteros con signo

El problema surge cuando se quieren representar enteros con signo, ya que éste requiere un bit. Existen diferentes maneras de resolver la codificación de los enteros con signo. Los formatos más importantes son: signo-magnitud, complemento a 2 y exponente desplazado.

En el formato de *signo-magnitud* (SM), el bit situado más a la izquierda (el bit más significativo) representa el signo del entero (0 para positivo, 1 para negativo). Los demás bits representan su valor absoluto. El intervalo de representación es  $-2^{n-1}+1 \leq x \leq 2^{n-1}-1$ .

Por ejemplo:

$$100110 = -(0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -(0 + 0 + 4 + 2 + 0) = -6$$

$$000110 = 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 0 + 0 + 4 + 2 + 0 = 6$$

El formato de *complemento a 2* (C2) es el habitualmente utilizado en los ordenadores para representar valores numéricos enteros con signo. El bit más a la izquierda representa el valor  $-2^{n-1}$  siendo n el número total de bits empleado. En este caso, el intervalo de representación es  $-2^{n-1} \leq x \leq 2^{n-1}-1$ . Por ejemplo:

$$000110 = 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 0 + 0 + 0 + 4 + 2 + 0 = 6$$

$$100110 = -1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -32 + 0 + 0 + 4 + 2 + 0 = -26$$

El valor opuesto de un entero (cambio de signo) representado en complemento a 2 se puede obtener cambiando ceros por unos y viceversa y sumando uno a la expresión en binario que resulta. Por ejemplo, partiendo que la secuencia 000110 representa el valor 6, si cambiamos los ceros por unos y viceversa (111001) y sumamos 1, obtenemos el 111010, que representa el valor opuesto -6. Esta propiedad permite obtener la representación de un entero negativo con facilidad encontrando la representación del valor absoluto y luego cambiando el signo. Por ejemplo:

$$111010 = -1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -32 + 16 + 8 + 0 + 2 + 0 = -6$$

La ventaja del formato C2 es que las sumas y restas con valores numéricos representados con este formato quedan reducidas a sumas, independientemente del signo de los operandos. Esto simplifica notablemente las operaciones a realizar por las unidades aritmético-lógicas de los microprocesadores y es el motivo para que C2 sea la codificación de enteros por excelencia en informática.

El formato del *exponente desplazado o sesgado* considera el signo restando un valor constante al valor entero absoluto representado por una secuencia binaria. El valor constante restado suele ser  $2^{n-1}$ , siendo n el número de bits de la secuencia.

Por ejemplo:

$$100110 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 - 2^5 = 32 + 0 + 0 + 4 + 2 + 0 - 32 = 6$$

$$000110 = 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 - 2^5 = 0 + 0 + 0 + 4 + 2 + 0 - 32 = -26$$

En la tabla 1.4 se muestra el valor correspondiente de todas las posibles combinaciones de cuatro dígitos binarios en función del formato elegido para representar un valor numérico entero.

#### 1.7.1.5. Formato de punto fijo para fracciones

El formato de punto o coma fija para fracciones consiste en la expresión del número como suma de potencias positivas y negativas de 2 y en el que el punto que separa las potencias negativas está en una posición determinada entre dos dígitos binarios. Por ejemplo:

$$1001,101 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 8 + 1 + 0.5 + 0.125 = 9.625$$

Cuadro 1.4: Conversión de los principales formatos para la representación de números enteros

Binario (4 bits)	BCD	Hexadecimal	Binario natural (sin signo)	Signo-Magnitud	Complemento a 2 (C2)	Exponente Desplazado
0000	0	0	0	0	0	-8
0001	1	1	1	1	1	-7
0010	2	2	2	2	2	-6
0011	3	3	3	3	3	-5
0100	4	4	4	4	4	-4
0101	5	5	5	5	5	-3
0110	6	6	6	6	6	-2
0111	7	7	7	7	7	-1
1000	8	8	8	0	-8	0
1001	9	9	9	-1	-7	1
1010	-	A	10	-2	-6	2
1011	-	B	11	-3	-5	3
1100	-	C	12	-4	-4	4
1101	-	D	13	-5	-3	5
1110	-	E	14	-6	-2	6
1111	-	F	15	-7	-1	7

Para números negativos también se pueden emplear los formatos de signo-magnitud y de complemento a dos. En estos casos, si el bit más significativo vale 1, se toma, respectivamente, como valor de signo negativo o como el coeficiente del sumando  $-2^{m-1}$ , siendo m el número de bits a la izquierda de la coma. Por ejemplo:

$$(SM) \quad 0001,101 = 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

$$(SM) \quad 1001,101 = -(0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}) = -(1 + 0.5 + 0.125) = -1.625$$

$$(C2) \quad 0001,101 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

$$(C2) \quad 1001,101 = -1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = -8 + 1 + 0.5 + 0.125 = -6.375$$

Para pasar un valor fraccionario decimal a binario se multiplica sucesivamente la parte fraccionaria del valor de partida y de los resultados obtenidos en los productos por 2. El valor en binario se forma con las partes enteras (cero o unos) de los productos obtenidos. Por ejemplo:

$$\begin{array}{r}
 0.15625 \\
 \times 2 \\
 \hline
 0.3125
 \end{array}
 \quad
 \begin{array}{r}
 0.3125 \\
 \times 2 \\
 \hline
 0.625
 \end{array}
 \quad
 \begin{array}{r}
 0.625 \\
 \times 2 \\
 \hline
 1.25
 \end{array}
 \quad
 \begin{array}{r}
 0.25 \\
 \times 2 \\
 \hline
 0.5
 \end{array}
 \quad
 \begin{array}{r}
 0.5 \\
 \times 2 \\
 \hline
 1.0
 \end{array}$$

$$0.15625 = ,00101$$

$$\begin{array}{r}
 0.6745 \\
 \times 2 \\
 \hline
 1.349
 \end{array}
 \quad
 \begin{array}{r}
 0.349 \\
 \times 2 \\
 \hline
 0.698
 \end{array}
 \quad
 \begin{array}{r}
 0.698 \\
 \times 2 \\
 \hline
 1.396
 \end{array}
 \quad
 \begin{array}{r}
 0.396 \\
 \times 2 \\
 \hline
 0.792
 \end{array}
 \quad
 \begin{array}{r}
 0.792 \\
 \times 2 \\
 \hline
 1.584
 \end{array}
 \quad
 \begin{array}{r}
 0.584 \\
 \times 2 \\
 \hline
 1.168
 \end{array}
 \quad
 \begin{array}{r}
 0.168 \\
 \times 2 \\
 \hline
 0.336
 \end{array}
 \quad
 \begin{array}{r}
 0.336 \\
 \times 2 \\
 \hline
 0.672
 \end{array}$$

$$0.6745 = ,10101100...$$

La **precisión** de números reales (racional o irracional) representados en codificación de coma o punto fijo está determinada por el número de dígitos empleado. En los sistemas de representación numérica en **coma fija**, la **resolución es uniforme** en todo el intervalo de representación e igual que el **peso del dígito menos significativo**. Si un número real  $x$  se aproxima por su representación más cercana,  $x'$ , entonces la precisión o error absoluto de representación que se obtiene es igual o menor que la resolución. Nótese que la representación de un número racional (fracción) puede ser

exacta en un sistema de numeración y aproximada en otro sistema con distinta base. La exactitud de la representación depende de la divisibilidad del denominador por la base del sistema de numeración.

#### 1.7.1.6. Formato de punto o coma flotante

Es la solución al problema del aumento geométrico del número de bits necesario para representar valores enteros o reales mayores. Una solución a este problema consiste en la utilización de la *notación científica* o notación de punto o coma flotante. Todo valor numérico se puede representar por una mantisa, una base (que coincide con la del sistema de numeración) y un exponente. Utilizando el sistema decimal y, por lo tanto, una base decimal:

$$54016 = 0.54016 \cdot 10^5 = 0.54016E + 05$$

$$0.00374 = 0.374 \cdot 10^{-2} = 0.374E - 02$$

En el último formato, la letra E sustituye a la base 10 de la potencia. Pero, como ya se ha dicho los ordenadores emplean sistemas binarios (con base binaria), lo que queda implícito al codificar cualquier valor numérico mediante una mantisa y un exponente expresados como potencias negativas y positivas de 2. La mantisa puede representarse empleando un formato en punto fijo, mientras que para el exponente se puede utilizar cualquier formato de representación de un entero. Por ejemplo:

$$54016 = 2^{15} + 2^{14} + 2^{12} + 2^9 + 2^8 = (2^{-1} + 2^{-2} + 2^{-4} + 2^{-7} + 2^{-8}) \cdot 2^{16}$$

se convierte en formato binario en 011010011 (mantisa) y 010000 (exponente)

$$0.0032501 = 2^{-9} + 2^{-10} + 2^{-12} + 2^{-14} + 2^{-16} = (2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8}) \cdot 2^{-8}$$

se convierte en formato binario en 011010101 (mantisa) y 111000 (exponente)

Se dice que un valor binario en coma flotante está *normalizado* si el valor absoluto de la mantisa, m, cumple la siguiente condición:  $0.5 \leq |m| < 1$ . El valor binario del ejemplo anterior está normalizado ya que el valor numérico de la mantisa es 0.83203. Por el contrario, las secuencias binarias, 8 bits de mantisa, 6 bits de exponente: 00110101 111001 ó 11001110 010010 representan valores numéricos no normalizados. Es muy habitual el empleo del formato en coma flotante con exponente desplazado en la que al valor entero del exponente tomado como entero positivo se le resta siempre  $2^{n-1}$  siendo n el número de bits utilizado para la codificación del mismo exponente. La precisión también está determinada por el número de dígitos empleados. Como se verá en la siguiente sección, los principales factores para seleccionar una codificación específica dependen del coste de la traducción, el coste del almacenamiento y el coste del procesamiento. Las operaciones realizadas con este tipo de datos son relativamente lentas y se llevan a cabo empleando código o subrutinas específicas (*software*) o coprocesadores matemáticos (*hardware*).

#### 1.7.2. Operaciones aritméticas

Para realizar operaciones con datos numéricos en formato binario hay que utilizar la aritmética binaria teniendo en cuenta, además, la limitación del número de bits para representar operandos y resultados. Este último punto determina la existencia de un intervalo de representación y una precisión para los datos numéricos a manipular. La forma en la que se realicen las operaciones dependerá del formato de representación de los valores numéricos. Cualquiera que sea la forma de representación y el número de bits empleado hay un límite en el rango o intervalo de números que pueden representarse. Cuando el resultado de una operación excede de este intervalo se produce un *desbordamiento* u *overflow*. Pero, entonces ¿por qué no se emplean, por ejemplo, 100 bits o una cantidad incluso mayor, para codificar cualquier dato numérico? Sencillamente por tres motivos:

1. el coste de la codificación (pruébese a codificar -45 en complemento a dos con 100 bits)
2. el coste de almacenamiento (mayores necesidades en la capacidad de almacenamiento y ¡la memoria en un ordenador sale muy cara!) y
3. el coste del tratamiento de la información (las operaciones aritméticas se complican).

Asimismo, esta limitación también incide en la precisión o resolución de representación. Si se están utilizando valores fraccionarios, a mayor número de bits mayor precisión. De nuevo aparece el problema de utilizar un número grande de dígitos para representar un valor. Por otro lado, algunas operaciones se realizan en términos de otras. Las multiplicaciones se llevan a cabo mediante desplazamientos<sup>1</sup> a la izquierda y sumas. Las divisiones mediante desplazamientos a la derecha y restas.

En relación con el desarrollo de programas científico-técnicos es necesario comprender y tener presente que:

1. La operación aritméticas sobre representaciones de números enteros son exactas, pero están sujetas a desbordamiento si el resultado excede el rango de representación del número según el número de bytes utilizado.
2. La representación y operaciones de números reales en coma flotante están sujetas a errores numéricos por la aproximación de las cantidades a través de un número finito de bits de mantisa y exponente. Además están sujetas a un rango de representación, aunque este valor suele ser suficientemente grande para no dar problemas.

### 1.7.3. Datos alfanuméricicos (caracteres)

Básicamente, los ordenadores sólo trabajan con números (en realidad con bits). Pueden representar letras y otros caracteres asociándoles un número a cada uno de ellos mediante tablas. El carácter es la unidad de información a nivel del alfabeto humano. Hay caracteres alfabéticos, numéricos (dígitos decimales) y especiales (signos de puntuación, etcétera). En un principio lo habitual era emplear secuencias de seis, siete u ocho caracteres para representar los distintos caracteres. Una de las primeras tablas de equivalencia entre caracteres y números y una de las más extendidas es la codificación o tabla ASCII (correspondiente a las siglas de *American Standard Code for Information Interchange*). Utiliza 7 bits para 128 caracteres. Permite representar cuatro tipos de caracteres:

1. Caracteres alfabéticos, tanto para las mayúsculas como para las minúsculas: A, B, C,... Z, a, b, c,... z
2. Caracteres numéricos, es decir, los dígitos decimales: 0, 1, 2, 3,... 9
3. Caracteres especiales y de puntuación: incluyen algunos signos de puntuación : & #, \*
4. Caracteres de control.

El principal inconveniente de la tabla ASCII original es que carece de representación para las numerosas letras especiales que se encuentran en idiomas distintos del inglés.

Para dar cierto soporte internacional el ASCII se amplió posteriormente a 8 bits (ASCII extendido, ampliado o de 8 bits) y a una serie de tablas estándares según distintas agrupaciones de idiomas. Así, por ejemplo, el estándar ISO 8859-1 se corresponde con la tabla de caracteres de Europa occidental, recientemente modificada para dar lugar al estándar ISO 8859-15, en donde se incluye el símbolo del euro (€) que hasta entonces no aparecía en las tablas.

No obstante, este sistema de tablas tiene un inconveniente importante y es la ausencia de soporte simultáneo para todas los idiomas. Esta limitación se puede concretar, por ejemplo, en la imposibilidad de representar en el mismo texto letras de distintos idiomas que no estén agrupados en la misma tabla ISO. A pesar de este inconveniente la codificación de caracteres según estándares ISO es quizás la más utilizada.

Actualmente y en el futuro se empleará cada vez con mayor frecuencia los códigos UNICODE (también especificados en normas ISO). Existen dos formas de representar Unicode que se utilizan con frecuencia: las representaciones UTF-8 y UTF-16 (UTF, *Unicode Transformation Format*). La representación UTF-8 utiliza de 1 a 4 bytes<sup>2</sup> para representar letras en la práctica totalidad de los idiomas. Es decir, la representación de las letras tiene longitud variable, hay letras que se codifican en 1 byte (por ejemplo, las letras inglesas) y otras en 2, 3 ó incluso 4 bytes. Una ventaja de UTF-8 es que la representación del alfabeto inglés en UTF-8 y ASCII coincide (es compatible, se interpreta o codifica de la misma manera). La representación mediante UTF-16 es parecida a la UTF-8 pero, en este caso, se utilizan 1 ó 2 grupos de 2 bytes cada uno para realizar la representación, es decir, una letra está representada mediante 2 ó 4 bytes.

En relación con el desarrollo de programas en general y especialmente aquellos en que sea necesario comunicación entre ordenadores y/o internacionalización de los textos incluidos es necesario conocer:

1. la importancia de conocer el estándar seguido en la codificación de los textos,

---

<sup>1</sup>Se verá más adelante, un desplazamiento consiste en mover las cifras de un número hacia un lado u otro. Es una operación equivalente a la forma en que se colocan los números en una multiplicación de varias cifras cuando se obtienen los resultados parciales de la misma, antes de sumar.

<sup>2</sup>Por esto Microsoft utiliza las siglas MBCS (multibyte character set) para referirse al UTF-8, a pesar de que esta denominación puede inducir a confusión porque el UTF-16 también es una representación de varios bytes. Microsoft denomina al UTF-16 simplemente como UNICODE. De nuevo de forma ambigua porque no es el único UNICODE.

Cuadro 1.5: Tabla ASCII

P	C	P	C	P	C	P	C
0	<i>nul</i>	32	<i>spa</i>	64	@	96	'
1	<i>soh</i>	33	!	65	A	97	a
2	<i>stx</i>	34	"	66	B	98	b
3	<i>etx</i>	35	#	67	C	99	c
4	<i>eot</i>	36	\$	68	D	100	d
5	<i>eng</i>	37	%	69	E	101	e
6	<i>ack</i>	38	&	70	F	102	f
7	<i>bel</i>	39	,	71	G	103	g
8	<i>bs</i>	40	(	72	H	104	h
9	<i>tab</i>	41	)	73	I	105	i
10	<i>lf</i>	42	*	74	J	106	j
11	<i>vt</i>	43	+	75	K	107	k
12	<i>ff</i>	44	-	76	L	108	l
13	<i>cr</i>	45	.	77	M	109	m
14	<i>so</i>	46	/	78	N	110	n
15	<i>si</i>	47	0	79	O	111	o
16	<i>dle</i>	48	1	80	P	112	p
17	<i>dc1</i>	49	2	81	Q	113	q
18	<i>dc2</i>	50	3	82	R	114	r
19	<i>dc3</i>	51	4	83	S	115	s
20	<i>dc4</i>	52	5	84	T	116	t
21	<i>nak</i>	53	6	85	U	117	u
22	<i>syn</i>	54	7	86	V	118	v
23	<i>etb</i>	55	8	87	W	119	w
24	<i>can</i>	56	9	88	X	120	x
25	<i>en</i>	57	:	89	Y	121	y
26	<i>sub</i>	58	<	90	Z	122	z
27	<i>esc</i>	59	=	91	[	123	{
28	<i>fs</i>	60	^	92	\	124	
29	<i>gs</i>	61	>	93	]	125	}
30	<i>rs</i>	62	?	94	_	126	~
31	<i>us</i>	63		95		127	del

2. que la representación de un mismo texto en distintos estándares no es necesariamente igual y que, en algunos casos, a partir de la representación no se puede deducir el estándar empleado y
3. que quizás la mejor representación es el UTF-8, aunque no sea la que se emplea por defecto en la mayoría de ordenadores y tenga el inconveniente de la longitud variable de caracteres.

## 1.8. Lógica booleana

La *lógica booleana* desarrollada en 1847 por el matemático inglés George Boole (1810–1864) es un conjunto de operaciones que manipulan datos o variables booleanas. Un dato o variable booleana sólo puede tener dos valores o estados: verdadero/falso, alto/bajo, 1/0,... El tratamiento último de los datos en la Unidad Aritmético-Lógica del procesador de un ordenador se lleva a cabo en términos de variables booleanas. Las operaciones lógicas operan sobre datos o variables booleanas produciendo otro dato o variable booleana. Cada operación lógica se caracteriza por una *tabla de la operación* o *tabla de verdad*, que proporciona el valor de la variable resultante en función de todos los valores posibles de las variables de entrada. Cada operación tiene dos símbolos: el del *circuito lógico* o puerta lógica que realiza la operación y el del álgebra booleana. La puerta lógica es el dispositivo que realiza la operación lógica. Las operaciones lógicas básicas son:

NOT (*Negación*): Tiene una variable de entrada y otra de salida. El valor de la variable de salida es el opuesto al de la variable de entrada. La operación se expresa como  $Q = \neg P = \overline{P}$  y la tabla de verdad de esta operación se muestra en la Tabla 1.6.

Cuadro 1.6: Tabla de verdad del operador NOT

P	$Q = \neg P$
0	1
1	0

AND (*Producto* o conjunción lógica): Tiene dos variables de entrada y una de salida. La de salida es 1 si todas las variables de entrada son 1. La operación se expresa como  $R = P \cdot Q$  y la tabla de verdad de esta operación se muestra en la Tabla 1.7.

Cuadro 1.7: Tabla de verdad del operador AND

P	Q	$R = P \cdot Q$
0	0	0
0	1	0
1	0	0
1	1	1

La tabla de verdad para la operación AND puede extenderse para tres o más variables siguiendo las reglas establecidas.

OR (*Suma* o disyunción lógica): Tiene dos variables de entrada y una de salida. La de salida es 1 si alguna de las variables de entrada es 1. La operación se expresa como  $R = P + Q$  y la tabla de verdad de esta operación se muestra en la Tabla 1.8.

Cuadro 1.8: Tabla de verdad del operador OR

P	Q	$R = P + Q$
0	0	0
0	1	1
1	0	1
1	1	1

La tabla de verdad para la operación OR también puede extenderse para tres o más variables siguiendo las reglas establecidas.

XOR (*Suma exclusiva*): Tiene dos variables de entrada y una de salida. La de salida es 1 si únicamente una de las variables de entrada es 1. La operación se expresa como  $R = P \oplus Q$  y la tabla de verdad de esta operación se muestra en la Tabla 1.9.

Cuadro 1.9: Tabla de verdad del operador XOR

P	Q	$R = P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

Como en los operadores anteriores la tabla de verdad para la operación XOR también puede extenderse para tres o más variables siguiendo las reglas establecidas.

Puede demostrarse que cualquier operación lógica con cualquier número de entradas puede realizarse mediante combinaciones de puertas NOT y AND.

## Ejercicios resueltos del capítulo de Fundamentos

- Un alumno dispone de un dispositivo de memoria USB (*Universal Serial Bus*, en inglés *pendrive* o *USB flash drive*) con capacidad de almacenamiento de 128 megabytes y de otro dispositivo similar de 256 megabytes. Indicar la capacidad total de almacenamiento de datos de ambos en bits. Nota: es suficiente indicar la expresión numérica decimal correspondiente.

2. Expresar en formato binario el siguiente número expresado en hexadecimal: AE9D
3. Si se tienen diez bits para la representación binaria de un valor numérico entero en complemento a dos y en formato de signo-magnitud, ¿cuál es, en cada caso, el número negativo de mayor valor absoluto que se puede representar? Indicar la respuesta en el sistema decimal.
4. Diseña los campos de bits mínimos necesarios para codificar en binario la fecha de nacimiento (día del mes, mes y año) de una persona nacida entre el año 1 y el año 10.000. Para ello debe indicarse con claridad la codificación y el significado de cada bit ó bits; y si un dato se codifica en varios bits el número de ellos. Utilícese el diseño para codificar la fecha de nacimiento del propio alumno.
5. Se quiere diseñar un formato de representación binaria en coma o punto fijo que, utilizando el MÍNIMO número de bits, cumpla las tres condiciones siguientes: (a) debe representar números reales con un valor absoluto de, al menos, hasta 1000 (incluido), (b) el valor representado por el dígito binario menos significativo es 0.125 y (c) debe emplear la representación en signo-magnitud. Indicar el número MÍNIMO total de dígitos binarios necesarios.
6. En una representación en formato de punto fijo y complemento a 2 con 8 bits (5 bits para la parte entera y 3 para la parte no entera), indicar el valor mínimo y máximo de los números representados y completar el siguiente diagrama. ¿Cuántas representaciones binarias distintas podemos realizar entre los números decimales 1,875 y 1,999 sin contar éstas dos?
7. Dada la siguiente secuencia de bits 1100 1001, indicar cuál es el valor decimal representado: si es un número entero, expresado en: (a) binario puro, (b) en signo-magnitud y (c) en complemento a dos ó si es un número real expresado en coma flotante (los 4 primeros bits corresponden a la mantisa y los cuatro últimos para el exponente, ambos en punto fijo y complemento a dos).
8. Dada las siguientes secuencias binarias que corresponden a valores reales expresados en coma flotante (los 6 primeros bits corresponden a la mantisa y los cuatro últimos para el exponente, ambos en punto fijo y C2): (a) 010000 0001, (b) 001001 0010, (c) 100001 0100, (d) 011111 1111 y (e) 110000 1000, ordenarlas de menor a mayor según el valor decimal representado. Nota: no es necesario indicar el valor representado. Es suficiente indicar la letra ordinal correspondiente al valor numérico binario.
9. Sabiendo que se utiliza 8 bits para representar un valor en coma flotante, 4 para la mantisa en punto fijo y a) signo-magnitud b) complemento a dos) y 4 para el exponente (exponente desplazado). ¿Cuál es el valor decimal almacenado en: 1111 1101?
10. Representar el valor entero decimal 208 en binario en formato de coma flotante normalizado ( $1/2 \leq |\text{mantisa}| < 1$ ) con el menor número de bits para mantisa y exponente, ambos en complemento a dos.
11. Representar exactamente el número -5,50 en formato de coma o punto fijo y en formato de coma o punto flotante siempre en complemento a 2 y utilizando 8 bits en ambas representaciones. Se pueden repartir en cada caso los 8 bits en la forma mas adecuada para representar la parte entera y decimal en el caso de punto fijo y la mantisa y el exponente en el caso de punto flotante en formato normalizado.
12. Codificar el valor numérico decimal  $-7.25$  en binario utilizando un formato de coma o punto fijo con 12 bits y complemento a 2 y otro formato de coma o punto flotante con 8 bits para la mantisa y 4 bits para el exponente (ambos en complemento a 2). Nota: Debe considerarse que la representación en coma flotante está normalizada ( $1/2 \leq |\text{mantisa}| < 1$ ).
13. Calcular las siguientes operaciones lógicas bit a bit sobre los números binarios que se indican: (a) 0101 AND 0011 y (b) 0101 OR 0011
14. Sea el operador booleano NN definido mediante la siguiente tabla de verdad (0=falso, 1=verdadero): Escribir la expresión más simple posible del operador NN en función de los operadores booleanos básicos (AND, OR y NOT).

P	Q	R = P NN Q
0	0	1
0	1	0
1	0	0
1	1	0

15. Representar su número de matrícula como alumno de la ETSII-UPM dividido entre  $10^3$ (mil) como un valor en punto fijo y complemento a dos, si no admite una representación exacta truncar por la derecha.

## Soluciones a los ejercicios del capítulo de Fundamentos

1.  $(128+256) \text{ Megabytes} \cdot 2^{20} \text{bytes/Megabyte} \cdot 8 \text{ bits/byte} = 3.221.225.472 \text{ bits en total}$
2. 1010 1110 1001 1101
3. En complemento a dos: -512. En signo magnitud: -511
4. 5 bits en base 2 para el días del mes / 4 bits en base 2 para el mes / 14 bits en base 2 para el año. Por ejemplo, si la fecha es 29/08/1972 la secuencia binaria de representación es 11101 1000 00011110110100
5. 14
6. Valor mínimo = -16. Valor máximo = +15,875. Representación de 1,875 = 00001,111. Representación de 1,999 = 00001,111. Número de representaciones binarias diferentes entre 1,875 y 1,999: 0
7. (a) 201, (b) -73, (c) -55, (d),  $-1/256 = -0,00390625$
8. c, e, d, a, b
9. -28 y -4
10. Mantisa = 0.1101, exponente = 01000.
11. En punto fijo: 1010.1000; en punto flotante: 10101 011
12. Coma o punto fijo: 1111000,11100. Coma o punto flotante: 10001100 0011
13. (a) 0001, (b) 0111
14. NOT (P OR Q)
15. Por ejemplo, si el número de matrícula es 11625 entonces el número a representar es 11,625 que, en binario en punto fijo y C2, se puede codificar como 01011,10100

## Ejercicios propuestos del capítulo de Fundamentos

1. A partir de la representación de -9.30 en punto fijo y complemento a dos, con 10 bits y el punto entre el 5º y 6º bit, se pide obtener dicho valor en binario en coma flotante con 5 bits para la mantisa (m) y los otros cinco para representar el exponente (e), ambos en complemento a dos, utilizando formato normalizado ( $1/2 \leq |m| < 1$ ).
2. En una representación en formato de punto fijo y complemento a 2 con 8 bits (5 bits para la parte entera y 3 para la parte no entera), indicar el valor mínimo y máximo de los números representados y completar el siguiente diagrama. ¿Cuántas representaciones binarias distintas podemos realizar entre los números decimales 1,875 y 1,999 sin contar éstas dos?
3. Construir un circuito lógico que implemente un semi-sumador binario.
4. Diseñar una codificación binaria que permita representar el número  $1/3$  de forma exacta.
5. Encontrar una manera de representar una cantidad de euros en forma de números enteros.
6. Encontrar ventajas e inconvenientes de utilizar una codificación de números enteros o reales para representar unidades monetarias (dinero, por ejemplo, en euros).

## Capítulo 2

# Introducción a la programación

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Definir qué es el lenguaje C (Conocimiento)
2. Definir los conceptos de edición, compilación y ejecución (Conocimiento)
3. Describir las herramientas necesarias para trabajar con C (Conocimiento)
4. Instalar las herramientas necesarias para trabajar con C (Aplicación)
5. Realizar la codificación, compilación y ejecución de un programa dado en C (Aplicación)
6. Describir la estructura del código fuente de un programa en C (Conocimiento)
7. Realizar la compilación y ejecución de un programa dado en C (Aplicación)
8. Interpretar la estructura del código fuente de un programa en C (Aplicación)
9. Escribir la estructura básica de un programa de C (Aplicación)

### 2.1. ¿Qué es el C?

#### 2.1.1. Origen del lenguaje C

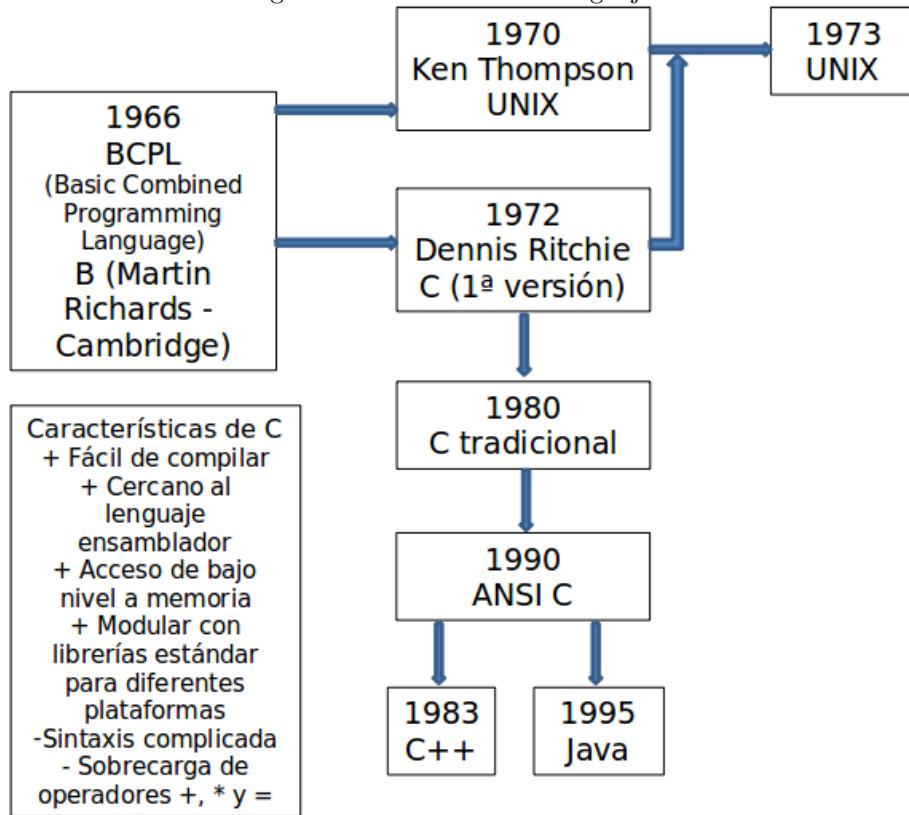
El lenguaje de programación C fue desarrollado originalmente por *Dennis Ritchie* de *Laboratorios Bell* a principios de la década de los 70 y fue diseñado para ejecutarse en un ordenador PDP-11 con el sistema operativo *UNIX*. Posteriormente hubo un gran interés en que pudiera funcionar bajo el sistema operativo MS-DOS en el IBM-PC y compatibles ya a principios de los ochenta. Es un excelente lenguaje de programación para este entorno por la sencillez de sus expresiones, lo compacto de su código y el amplio abanico de su aplicabilidad. También debido a la relativa simplicidad y fácil escritura de un compilador de C, es habitualmente el primer lenguaje de alto nivel disponible en un ordenador nuevo, incluyendo, ordenadores personales, estaciones de trabajo y *mainframes*. Aunque C no está considerado como el mejor lenguaje para un principiante en programación debido a su naturaleza criptica. Permite al programador un amplio rango de operaciones desde un alto nivel hasta un nivel muy bajo, acercándose al lenguaje ensamblador, dando la impresión de que su flexibilidad no tiene límites. Un programador experimentado en C pronunció la frase “Puedes programar cualquier cosa en C” lo que puede quedar demostrado en la práctica por cualquier otro programador experimentado. Por otro lado, tiene el inconveniente de dejar al programador una gran responsabilidad porque es muy fácil construir un programa que tenga muchos pequeños errores fatales que, por ejemplo, un buen compilador de Pascal pueda detectar con facilidad. El lenguaje C deja mucho poder en manos del programador.

En la figura 2.1 se recoge un diagrama de la historia del lenguaje C.

#### 2.1.2. Versiones del lenguaje C

El lenguaje C es relativamente antiguo, su origen se remonta a 1972. Naturalmente con el paso del tiempo el lenguaje ha ido sufriendo las lógicas evoluciones, lo que significa que existen diferentes versiones de C. Más aún, los lenguajes de programación se basan en lo que entiende y procesa el programa que llamamos compilador. En este sentido, el lenguaje

Figura 2.1: Evolución del Lenguaje C



C está extendido en multitud de plataformas, desde microcontroladores hasta *clusters* de ordenadores y desde *GNU-Linux* hasta *Windows*. En cada una de estas plataformas se pueden encontrar uno o varios compiladores, de manera que esta proliferación de compiladores y plataformas provoca pequeñas diferencias en el lenguaje. Incluso aunque el lenguaje C disfruta de una buena transportabilidad cuando los programas son migrados de una versión a otra, hay diferencias entre los compiladores que se encontrarán cuando se intente compilar el mismo código con compiladores distintos. En 1978, *Dennis Ritchie* y *Brian Kernighan* publicaron la primera edición de *El lenguaje de programación C*, para formalizar y definir la especificación completa del lenguaje C.

Por poner un ejemplo, solo en entorno Windows se puede encontrar el compilador de Microsoft, el de Borland, el de Intel, el compilador MinGW (*minimalist GNU compiler for Windows*, es decir, el compilador *gcc* para este sistema operativo), ...

En general se intentan establecer estándares que faciliten la portabilidad entre plataformas, tanto desde el punto de vista del propio lenguaje como de las librerías más básicas (que se suelen llamar de sistema). De hecho, las mayores diferencias en el lenguaje se encuentran al usar librerías no estándar tales como las llamadas a un sistema operativo concreto. Normalmente estas diferencias se pueden minimizar teniendo cuidado en la escritura de los programas.

En lo que sigue y en la medida de lo posible se va a usar una forma estandarizada del lenguaje C: el lenguaje estándar ANSI-C, conocido también como C89.

### 2.1.3. El lenguaje estándar ANSI-C

Conforme se fue haciendo evidente que el lenguaje C se estaba convirtiendo en un lenguaje muy popular y disponible en una amplia variedad de ordenadores, un grupo de personas se reunieron para proponer un conjunto de reglas estándar en el uso del lenguaje C. El grupo representaba todos los sectores de la industria del software. Despues de muchas reuniones en 1989 completaron un estándar para el lenguaje C, aceptado por la *American National Standards Institute* (ANSI) y por la *International Standards Organization* (ISO). Se ratificó como el "Lenguaje de Programación C" *ANSI X3.159-1989*. Esta versión del lenguaje se conoce a menudo como *ANSI-C*, o a veces como C89 y no obliga a ningún grupo o usuario pero, como es ampliamente aceptado la mayoría de los compiladores, asegura en la práctica la compilación de programas escritos estrictamente bajo este estándar.

El estándar ANSI-C (C89) no es el último. En los años 90 se revisó este estándar para añadir algunas características de C++ (y otros lenguajes). No obstante, el C89 sigue siendo un estándar importante, soportado por muchos compi-

ladores y además es un subconjunto de C99 (el último estándar de C), lo que en el fondo tiene como consecuencia que todo programa en C89 también respeta el estándar C99.

A efectos prácticos, cuando se compila un programa se puede pedir al compilador que siga un estándar u otro. En lo que sigue se utilizará como referencia el ANSI-C, especificando en las opciones de compilación “**-ansi -pedantic**” para asegurar que el programa está codificado conforme a este estándar.

## 2.2. Edición, compilación y ejecución de un programa

El proceso de construcción de un programa es sencillo si se tienen las herramientas necesarias para (a) escribir y depurar los errores de sintaxis del programa escrito en un lenguaje de programación, (b) traducirlo al lenguaje de la máquina en la que se quiera ejecutar y por último, (c) depurar los errores de funcionamiento o lógicos del programa.

Más detalladamente, los pasos que se deben seguir son:

1. Un programa se escribe mediante un *editor de texto* en un lenguaje de programación. Por ejemplo, C. Este proceso se denomina **edición**.
2. El programa así escrito recibe el nombre de *código/programa fuente* y se almacena en un archivo de texto que se guarda en el ordenador como cualquier otro archivo. De hecho, el formato de los archivos fuente es texto plano, uno de los formatos de archivo más sencillos que existen. Para el ordenador, este archivo es el equivalente a un folleto de instrucciones en un idioma que desconoce.
3. Existen una serie de herramientas capaces de traducir automáticamente los archivos con el *código/programa fuente* que escriben los programadores al lenguaje que puede interpretar el ordenador, *lenguaje máquina/ binario*. Entre las herramientas que traducen el *programa fuente* a *lenguaje máquina*, las más importantes son los *compiladores*. Los compiladores traducen el *código/programa fuente* a un programa equivalente escrito en *lenguaje máquina*, el resultado de esta traducción se denomina *código/programa objeto*.
4. El archivo que contiene el *código/programa objeto* es el verdadero programa que puede ser ejecutado por el ordenador.

Así pues, generar un programa de ordenador consiste en escribir algo parecido a un manual de instrucciones con una cierta codificación, un lenguaje específico distinto del lenguaje natural y luego traducirlo mediante herramientas a otro lenguaje que es el propio de los ordenadores.

Por tanto, para realizar este proceso se necesitan dos herramientas informáticas:

1. Un editor para escribir el documento llamado código/programa fuente.
2. Un traductor para traducirlo a lenguaje máquina, generándose el código/programa objeto correspondiente.

## 2.3. Herramientas de programación en C

En informática se llama *entorno de desarrollo* a las herramientas (programas) que se utilizan para escribir programas de ordenador. En ocasiones estas herramientas se agrupan en una aplicación conjunta que recibe el nombre de IDE (*Integrated Development Environment*) o SDK (*Software Development Kit*). Cuando una persona va a empezar a programar lo primero que necesita es conocer las herramientas que mejor se adapten a sus necesidades y experiencia de programación.

En este texto se van a utilizar pocas herramientas para no añadir más complejidad al aprendizaje del lenguaje de programación. No obstante, la programación en un nivel profesional debe apoyarse en el empleo de herramientas más complejas y completas.

Así pues, no se va a utilizar un entorno de desarrollo integrado sino que se van a utilizar las siguientes herramientas elementales:

Uso	Windows	XUbuntu (GNU-Linux)
Edición de código fuente	Notepad++	geany
Traducción a código máquina (compilar)	mingw (gcc para Windows)	gcc
Gestión de la compilación de un programa	gnu make (para Windows)	gnu make

Siempre que sea posible se va a realizar código en C completamente estándar y portable de una plataforma a otra. Por esto se dan las herramientas que se van a utilizar tanto en GNU-Linux como en Windows.

### 2.3.1. El editor de texto

Un editor de texto es una aplicación que sirve para crear, editar y guardar archivos en formato de texto. Los archivos de texto utilizan tablas de caracteres (ASCII, ISO-xxxx, UTF-8, ...) para guardar información textual en disco o en el sistema de almacenamiento correspondiente.

Dado que los estándares de codificación de texto son universales, todos los editores de texto en cualquier plataforma pueden abrir o guardar este tipo de archivos. De esta manera, la elección de un editor u otro es una elección del usuario según la comodidad o funcionalidad que le proporcione el correspondiente editor.

### 2.3.2. El compilador y herramientas asociadas

El compilador GNU no es un único programa o herramienta sino que incluye varias herramientas que se suelen englobar en un conjunto de herramientas que, en general, se llama compilador:

1. El preprocesador (*preprocessor*).
2. El compilador propiamente dicho (*compiler*).
3. El enlazador (*linker*).
4. El depurador (*debugger*).

Como se ha indicado con anterioridad se llama *compilador* al programa que traduce un archivo de texto, código fuente, escrito en el lenguaje de programación correspondiente a código máquina (el que se puede ejecutar en un microprocesador). En el caso de C, la traducción a código máquina se ha dividido en tres fases: *preprocesado*, *compilación* y *enlace* o *enlazado*, cada uno ejecutado por un programa distinto.

1. El *preprocesador* realiza una sustitución de cadenas de texto o contenido de archivos en el código fuente. El preprocesador no realiza ninguna comprobación sintáctica ni de estructura de los archivos con los que trabaja. Se puede pensar que el preprocesador realiza operaciones sofisticadas de copiar y pegar texto sobre los archivos de código fuente. El resultado del preprocesador es, de nuevo, un conjunto de archivos de texto (*código fuente*) que se pasa al compilador.
2. El *compilador* realiza la traducción completa de todo el contenido de todos los archivos de código fuente indicados. Se debe observar que el compilador no solo compila el código que se haya escrito sino también el que se haya “pegado” en los archivos por la acción del preprocesador. El compilador produce un archivo de código máquina por cada archivo original. Generalmente este archivo tiene extensión .o ó .obj y se suele denominar *código objeto*. Estos archivos, a pesar de estar escritos en lenguaje máquina no se pueden ejecutar directamente.
3. El *enlazador* busca entre el código objeto los fragmentos de código máquina necesarios para formar el programa definitivo. El enlazador utiliza sólo aquellas partes de código objeto que son necesarias (generalmente el compilador ha generado mucho código objeto innecesario para el programa concreto que se está realizando) y además estructura, es decir, coloca este código objeto en la forma necesaria para que el microprocesador sea capaz de ejecutarlo como un programa. En Windows el resultado final es un archivo con extensión .exe.

## 2.4. Preparación del entorno de trabajo en Windows

### 2.4.1. Instalación de herramientas

Con el sistema operativo Windows XP, Vista ó 7 se puede utilizar cualquier editor de texto como, por ejemplo, el Bloc de Notas. Existe un editor alternativo de libre uso que tiene una interfaz de edición mas atractiva y con algunas utilidades interesantes a la hora de visualizar los distintos elementos del programa escrito en el lenguaje C. Se llama *Notepad++*. Por otro lado también es necesario descargarse e instalar un compilador de C. En este texto se indicará como se descarga y se instala el compilador MinGW.

#### Instalación del compilador

1. Dirección del entorno *mingw*: [www.mingw.org](http://www.mingw.org)<sup>1</sup>
2. En el lateral izquierdo, dentro de la sección Navigation > About, pulsar el texto *Downloads*,

---

<sup>1</sup>Visto por última vez el 7 de diciembre de 2011

3. Se accede a la dirección de SourceForge, en la que hay que seleccionar *Looking for the latest version? Download mingw-get-inst-20AAMMDD.exe*, donde AAMMDD son año, mes y día, respectivamente <sup>2</sup>.
4. Esto conduce a una página de descarga donde empezará la descarga en breve. En caso de que no se inicie de forma automática, se puede pulsar el texto “*direct link*” o se puede escoger un servidor alternativo (*mirror*).
5. Se ejecuta el archivo o se guarda el archivo y luego se ejecuta.
6. Al ejecutar se abre una ventana de comandos en línea con los mensajes de instalación. Se pueden escoger las opciones por defecto (excepto Aceptar la licencia) y si se desea se pueden escoger compiladores adicionales (C++, Objective C, Fortran, ADA, ...).

Alternativamente se puede instalar un IDE que instale el compilador. Por ejemplo, la instalación del IDE llamado *Codeblocks* incluye la instalación de mingw y su instalación resulta más sencilla.

### **Instalación de Notepad++**

1. Dirección de Notepad++ : <http://notepad-plus-plus.org/><sup>3</sup>
2. Pulsar la solapa *Download*
3. Pulsar en el archivo de instalación. Por ejemplo: Notepad++ v5.9.6.2 Installer <sup>4</sup>.
4. Se ejecuta el archivo o se guarda el archivo y luego se ejecuta.
5. Aceptando todas las ventanas del instalador por defecto

### **Configurar variables de entorno**

Dado que se va a usar en línea de comandos (opción *Símbolo del sistema* en el S.O. Windows) hay que incluir la ruta de carpetas o directorios donde se encuentra tanto el compilador y como la aplicación make en la variable de entorno del sistema PATH de Windows. Puesto que Windows cambia el acceso a la ventana que modifica las variables de entorno, lo más sencillo es poner “*Configurar Variable de Entorno*” en el sistema de ayuda Windows.

- En Windows 7 las opciones que deben elegirse son: Panel de control > Sistema y seguridad > Sistema > Configuración avanzada del sistema > Opciones avanzadas > Pulsar botón Variables de entorno > En el cuadro Variables del sistema buscar la variable PATH > Seleccionarla > Pulsar el botón Editar > Añadir la secuencia: ;C:\MinGW\bin;C:\Archivos de Programa\GnuWin32\bin; al final del valor de la variable PATH y pulsar Aceptar en todas las ventanas abiertas.
- En Windows XP, las opciones que deben elegirse son: Panel de control > Sistema > Opciones Avanzadas > Variables de entorno > Variable del Sistema > Buscar la variable PATH > Seleccionarla > Modificar > Añadir la secuencia: ;C:\MinGW\bin;C:\Archivos de Programa\GnuWin32\bin; al final del valor de la variable PATH y pulsar Aceptar en todas las ventanas abiertas.

### **2.4.2. Pasos para escribir programas sencillos**

#### **Preparación de la carpeta de trabajo**

A la hora de trabajar resulta conveniente tener almacenado el trabajo en una carpeta claramente identificable. Es decir, lo mejor es crear una carpeta en el usuario correspondiente y guardar allí todos los programas que se vayan a escribir.

Por ejemplo, el *usuario1* podría crear una nueva carpeta de nombre *programasC* de la carpeta *users/usuario1*, de forma que contenga los archivos de código fuente y objeto de dicho *usuario1*.

Para crear esa nueva carpeta, el proceso sería:

1. Abrir el explorador de Windows: Equipo > Disco local C: > usuarios > usuario1 >
2. Pulsar el botón Nueva Carpeta.
3. Asignar el nombre *programasC* a dicha carpeta. <sup>5</sup>

<sup>2</sup>La última versión vista era del 16 de marzo de 2009

<sup>3</sup>Visto por última vez el 7 de diciembre de 2011

<sup>4</sup>Los números de versión se corresponden con la última versión vista

<sup>5</sup>Atención, en Windows 7, la carpeta *C:/usuarios* es la misma que *C:/users*

## Escritura de los programas

En general para escribir un programa sencillo se debe escribir el código fuente utilizando un editor de texto y, una vez escrito, se debe guardar en la carpeta de trabajo.

Si se ha creado la carpeta en *users/usuario1/programasC* y se ha instalado *Notepad++* basta con acceder al editor y elegir Archivo > Nuevo. A continuación hay que seleccionar el lenguaje con el que se va a desarrollar el programa. Para ello se tiene que elegir en la barra de menú principal la opción Lenguaje > C > C

En ese momento ya se estará en condiciones de escribir el contenido (código fuente) del nuevo programa en lenguaje C. Al finalizar hay que almacenar el nuevo programa en la carpeta *C:/usuarios/usuario1/programasC* utilizando la opción del menú principal Archivo > Guardar como > y buscando la carpeta anteriormente citada.

Si se ha seleccionado adecuadamente el lenguaje, el nuevo programa se almacenará con el nombre elegido y con la extensión .c, de forma que si se elige de nombre *primero*, la carpeta *C:/usuarios/usuario1/programasC* contendrá ahora un archivo de nombre *primero.c*, cuyo contenido es el archivo fuente recien creado.

Las modificaciones que se vayan incorporando en el programa fuente en la zona de edición de *Notepad++* tienen que almacenarse en la carpeta utilizando la opción menú principal Archivo > Guardar.

## Compilación de los programas

Los pasos que se deben seguir para utilizar el compilador son los siguientes:

1. Acceder al compilador a través de la consola del sistema. En Windows esta operación se lleva a cabo seleccionando Inicio > Programas > Accesorios > Símbolo del sistema. Automáticamente se abre una ventana de fondo negro en la que aparece el *prompt* o indicador del sistema con el nombre de la carpeta de usuario a la que se accede por omisión, en nuestro caso *C:\Users\usuario1*>
2. Cambiar a la carpeta creada por *usuario1* para almacenar el trabajo tecleando:

```
cd programasC
```

El indicador o *prompt* del sistema cambiará a *C:\Users\usuario1\programasC>*

3. Ejecutar el compilador, escribir detrás del *prompt* del sistema:

```
gcc -Wall -pedantic -ansi -o primero.exe primero.c
```

para comenzar la traducción del programa contenido en *primero.c*. Si el programa carece de errores de sintaxis el compilador genera un archivo de nombre *primero.exe*, consecuencia de la inclusión del parámetro *-o* en la línea de comando. Este archivo con el programa objeto se almacena en la misma carpeta de trabajo del usuario.

En esta ventana pueden aparecer mensajes de ayuda debido a la inclusión del parametro *-Wall* (*Warning all*) o de error si se han encontrado errores de sintaxis.

Las reglas y estándares que sigue la compilación de los programas en lenguaje C son los especificados en el lenguaje ANSI C de forma estricta. Esto es lo que produce la utilización de los dos últimos parámetros de la línea de comandos explicitada anteriormente (*-ansi* y *-pedantic*).

## Ejecución de los programas

En la misma ventana de fondo negro del sistema, basta con teclear el nombre del programa sin extensión. Si no existe ningún error lógico o de ejecución en nuestro programa el sistema interpretará correctamente las órdenes. En caso contrario se obtendrán mensajes de error o resultados *sorprendentes*.

## 2.5. Preparación del entorno de trabajo en GNU-Linux

### 2.5.1. Instalación de herramientas

Dada la profunda relación entre GNU-Linux y C (GNU-Linux está escrito en C) no debe sorprender que la instalación de las herramientas de trabajo con C en GNU-Linux sea muy sencilla. Como existen algunas diferencias entre las distribuciones de GNU-Linux, se va a elegir una sola distribución para mostrar la instalación y uso de las herramientas. En concreto se trabajará con *XUbuntu* (versión 12-04-LTS, con escritorio Xfce<sup>6</sup>).

<sup>6</sup>A partir de la versión 11-04 el escritorio de Ubuntu es Unity, en Unity la manera de acceder a las aplicaciones ha cambiado significativamente y por eso se prefiere esta otra distribución, más sencilla y más ligera.

Normalmente los ordenadores actuales se compran con Windows preinstalado. Si éste es el caso del lector y está dispuesto a probar, pero no a tener problemas con su equipo, se le recomienda instalar XUbuntu mediante un *Live CD* (se descarga de la página de XUbuntu y se siguen las instrucciones que aparecen allí para grabarlo) y escoger la opción “instalar Ubuntu al lado de los sistemas anteriores” (*Ubuntu alongside your current system*).

Una vez instalado XUbuntu para instalar las herramientas de desarrollo a través de las herramientas gráficas hay que:

1. Abrir el gestor de paquetes *Synaptic* en (Menú Aplicaciones<sup>7</sup> > Sistema).
2. Introducir la contraseña de administrador.
3. Localizar el paquete (editar > buscar) llamado “gcc”.
4. Marcarlo para instalar (hacer click derecho en el recuadro delante del nombre del paquete).
5. Pulsar Aplicar.

Para instalarlo desde la línea de comandos (en inglés, *shell*, el equivalente a la línea de comandos de Windows, *command.com*) basta con ejecutar la línea de comandos (Menú Aplicaciones -> Accesos -> Terminal) y teclear:

```
sudo apt-get install gcc8
```

Después pulsar *intro* y finalmente introducir la contraseña de administrador. En ambos casos la contraseña del administrador es la del usuario que se ha indicado en la instalación inicial de XUbuntu.

A continuación instalar el entorno de programación *geany* mediante el paquete del mismo nombre por cualquiera de las dos formas utilizadas para instalar gcc.

## 2.5.2. Pasos para escribir programas sencillos

### Preparación de la carpeta de trabajo

Aunque normalmente en los proyectos de C desarrollados bajo GNU-Linux se distribuyen los archivos por carpetas distintas con el código fuente, código objeto y demás, para simplificar el proceso de escritura de nuestros programas se usará simplemente una carpeta donde se guardarán todos los archivos. Dado que en el proceso de escritura de los programas se empleará como base la línea de comandos (*shell*, consola, línea de comandos, ...) se pondrán las instrucciones para crear esta carpeta mediante la línea de comandos.

1. Abrir la línea de comandos que se encuentra en Aplicaciones > Accesos > Terminal.
2. Para crear la carpeta teclear (y pulsar *enter* al final<sup>9</sup>):

```
mkdir programasC10
```

3. Para pasar a esta carpeta teclear:

```
cd programasC
```

En adelante cuando se desee empezar a trabajar se realizarán los pasos 1 y 3 para situarse en la carpeta de trabajo correspondiente.

---

<sup>7</sup>En la instalación por defecto un ícono que aparece en la esquina superior izquierda y que equivale al botón de inicio de Windows.

<sup>8</sup>La tecla tabulador sirve para auto-completar posibles paquetes, una pulsación o dos pulsaciones para ver más opciones.

<sup>9</sup>En adelante no se volverá a recordar, pero siempre que se indica un comando que se escribe en la línea de comandos es necesario pulsar *enter* (tecla para poner saltos de línea en un editor de texto) para que el comando se empiece a ejecutar.

<sup>10</sup>La línea de comandos muestra por pantalla un mensaje de que está lista para recibir nuevas órdenes. Este mensaje se llama *prompt* y, por ejemplo, en Ubuntu consta de usuario@equipo, seguido del nombre de la carpeta actual de trabajo cuando sea distinta de la carpeta de usuario y, al final, un símbolo \$. En muchos tutoriales que describen comandos de línea de comandos incluyen este mensaje que naturalmente no es necesario escribir cuando se ejecuten.

### Escritura de los programas

Para crear un nuevo archivo de código fuente vamos a utilizar el editor *geany*. Al instalar geany se crea un acceso en (Menú Aplicaciones -> Desarrollo). Una vez abierto se pueden utilizar las opciones del menú archivo o los iconos para crear nuevos archivos, abrir archivos existentes o guardar. Aunque geany permite compilar y ejecutar directamente se aconseja usar la línea de comandos para realizar estas tareas.

El entorno de desarrollo *geany* es una aplicación que permite (entre otras cosas) escribir archivos de texto de código fuente en C. Entre sus funcionalidades destaca que es capaz de resaltar con colores de letra diferentes distintos elementos sintácticos de muchos lenguajes de programación, entre ellos C. Se usa como cualquier otro editor de texto y tiene las acciones habituales de los editores de texto distribuidas entre sus menús.

Para mejorar la manera en que quedan presentados los programas al abrirse con otros editores es conveniente indicar a *geany* que transforme los tabuladores en espacios. Esto se lleva a cabo mediante la opción (Editar > Preferencias > Editor > Sangría) y marcando la opción correspondiente. Normalmente el tamaño del tabulador se fija en cuatro espacios.

Es importante guardar repetidamente el archivo que se esté escribiendo para evitar perder información. Además siempre que se vaya a compilar, antes hay que guardar el archivo. Para guardar se selecciona: (Archivo > Guardar) ó (Archivo > Guardar como ...).

### Compilación de los programas

Una vez escrito el código fuente se va a utilizar el compilador desde la línea de comandos para construir el programa. Suponiendo que (a) ya se está en la carpeta de trabajo, (b) que se ha creado un archivo de código fuente llamado *primero.c* y (c) que se ha grabado la última versión (ver puntos anteriores), se teclea en la línea de comandos:

```
gcc -Wall -pedantic -ansi -o primero primero.c
```

Esto ejecutará el compilador *gcc* con las opciones indicadas. Si no hay errores de sintaxis en *primero.c* el compilador construye un archivo *primero* que constituye el programa que estamos desarrollando.

Las opciones utilizadas en el comando anterior le indican al compilador que genere un archivo de nombre *primero* (-o *primero*), que muestre todo los avisos (-Wall de *Warning all*) y que sea estricto respecto del estándar ANSI (combinación de las opciones -pedantic y -ansi).

### Ejecución de los programas

Una vez compilado el programa podemos ejecutarlo, lo más sencillo es volver a usar la línea de comandos, suponiendo que se continua con el ejemplo anterior, para ejecutar el programa se teclea:

```
./primero11
```

se mostrará por pantalla el resultado del programa y cuando la ejecución acabe aparecerá de nuevo el *prompt*.

Para realizar compilaciones y ejecuciones repetidas se puede utilizar las flechas de los cursores en la línea de comandos para recuperar (y ejecutar de nuevo) comandos anteriores.

## 2.6. El primer programa

Una buena manera de empezar a aprender C es con un programa sencillo. A continuación, en el ejemplo 2.1, se muestra el programa más sencillo que puede escribirse en C, es decir, no hay forma de simplificarlo y desgraciadamente no hace nada.

Ejemplo 2.1: Programa vacío

---

```
1
2 int main()
3 {
4 }
```

---

<sup>11</sup>El punto inicial significa carpeta de trabajo, es decir, la carpeta actual de trabajo, mientras que la barra es la marca para separar la carpeta y el nombre del archivo.

La palabra `main` es muy importante, y debe aparecer una única vez en todo programa de C. Es el punto de comienzo (en inglés *entry point*) del programa cuando éste se ejecuta aunque no tiene porqué ser lo primero que aparece escrito en el código fuente del programa. Siguiendo a `main` hay un par de paréntesis<sup>12</sup> que indican que `main` es una función. Este concepto se explicará más adelante. Por ahora, sólo es necesario saber que hay que escribir esos paréntesis. Las llaves de las líneas siguientes delimitan el código del programa en sí. Las instrucciones o sentencias del programa irán entre estas dos llaves, que no existen en este caso porque el programa no hace absolutamente nada aunque sea un programa escrito en C.

Este programa se puede compilar ejecutando el comando que aparece a continuación. Las opciones que aparecen en este ejemplo serán las que utilicemos como referencia durante el resto del manual.

```
gcc vacio.c -Wall -ansi -pedantic -o vacio
```

Al compilarse se produce la siguiente salida:

```
vacio.c: En la función ‘main’:
vacio.c:3: aviso: el control alcanza el final de una función que no es void
```

Estos avisos cambian si no se compila con las opciones indicadas. No obstante, la mejor manera de solucionar estos avisos es transformar el programa según aparece en 2.2.

---

#### Ejemplo 2.2: Programa vacío sin avisos

---

```
1
2 int main()
3 {
4     return 0;
5 }
```

---

Este programa modificado debería poder ser compilado en todo compilador ANSI-C. Las diferencias entre los dos programas anteriores se explicarán posteriormente.

## 2.7. Un programa que hace algo

El siguiente paso es incluir entre las anteriores llaves alguna instrucción en lenguaje C que al ser ejecutadas realicen alguna tarea. En concreto se va a construir un programa que visualice en la consola del ordenador el mensaje: *Esta es una linea de texto en pantalla*. El contenido de este programa se muestra en el ejemplo 2.3.

---

#### Ejemplo 2.3: Muestra un mensaje

---

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Esta es una linea de texto en pantalla.");
7
8     return 0;
9 }
```

---

En el ejemplo 2.3 se puede observar que el programa tiene dos sentencias entre las llaves:

1. La primera de ellas, la sentencia `printf("Esta es una linea de texto en pantalla.");`; visualiza por pantalla el texto que aparece entre los paréntesis y entre las comillas dobles. La palabra `printf` es una llamada a una función definida en la librería estándar de C cuyo archivo de cabecera es *stdio.h* (*standard input output*). La función `printf`, así como muchas otras funciones, se encuentra declarada en un archivo cuyo nombre es precisamente *stdio.h*. Para incluir el contenido de esta librería en un programa es necesario escribir la primera línea del anterior programa, que no es propiamente una instrucción en lenguaje C sino de instrucción para el preprocesador y que se verá más adelante.

<sup>12</sup>esta declaración de `main` es una versión simplificada, la declaración completa tiene 2 parámetros entre los paréntesis.

2. la segunda instrucción, la sentencia `return 0;` finaliza la ejecución del programa y devuelve como resultado el número entero 0, que concuerda con el tipo de dato que debe devolver la función `main` definido en la declaración de la misma, el `int` que se escribe delante de `main`.
3. Cada una de estas sentencias se completa con el carácter punto y coma que marca el final de cada de ellas.

## 2.8. Salida estándar básica en C

La utilidad de un programa que no muestre o genere ningún tipo de información es muy limitada. Para generar esta información y poder disponer de ejemplos significativos del comportamiento del lenguaje es conveniente conocer cómo se puede mostrar por pantalla información en un programa en C.

En esta sección se describe como se puede obtener una visualización de datos y/o resultados mediante el dispositivo de salida estándar del ordenador que es la pantalla. En la sección anterior se ha mostrado una de las utilidades de la función `printf`: la visualización en pantalla de texto encerrado entre comillas dobles. Pero `printf` tiene muchas más utilidades, dependiendo de la sintaxis que se utilice en el programa. La sintaxis de la función `printf` es muy extensa. Por ahora se va a describir únicamente cómo mostrar números.

Si se desea visualizar datos numéricos enteros o datos numéricos reales entonces se deben utilizar unas marcas entre las comillas dobles en el lugar donde queremos que se escriba el número. Estas marcas reciben el nombre de *especificadores de formato*. Estos especificadores están formados por el símbolo `%` y, en su sintaxis más sencilla, por una letra que indica el tipo de valor que se desea visualizar por pantalla. Si es un valor entero entonces la letra es `d` (de número decimal, también se puede poner `i` de *integer*) y si es real `f` (de *float*, coma flotante).

Ejemplo 2.4: Muestra datos numéricos

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Esta es una linea de texto en pantalla.\n");
7
8     printf("Este es el número entero tres %d\n",3);
9
10    printf("Este es el número real tres %f\n",3.0);
11
12    return 0;
13
14 }
```

---

En el programa que se muestra en el ejemplo 2.4, se observan cuatro sentencias entre las llaves:

1. En la primera de ellas, el único detalle a destacar es el carácter de control *nueva línea* (en inglés *newline*) representado por `\n` y que se encuentra al final de la sentencia. Este carácter mueve el cursor de escritura<sup>13</sup> al principio de la siguiente línea de la pantalla. En cualquier lugar que aparezca se introducirá una nueva línea, incluso en mitad de una palabra.
2. En la segunda sentencia `printf`, entre paréntesis y dentro de la cadena de caracteres encerrada entre comillas dobles se encuentra el carácter `%`. Este carácter le indica a la función `printf` que sustituya esa marca en la visualización del texto por pantalla por el valor que aparece detrás de la coma, en este caso para mostrar el valor entero 3. Como el carácter que sigue al `%` es una `d`, indica que el valor que debe esperar es un número entero. Después de la comilla doble, aparece una coma seguida del valor entero correspondiente, en este caso, 3. De ahí toma la función `printf` el valor que va a sustituir en `%d` visualizándose en la pantalla. Podrían incluirse más especificadores de formato `%d` en el texto entrecomillado, así como más valores después del primero lo que produciría la salida por pantalla de más valores enteros en el texto. Los valores adicionales se separan por comas. En cualquier caso, el número de especificadores de formato y de valores debe ser el mismo o la ejecución de `printf` puede tener un resultado inesperado.

<sup>13</sup>En general, el cursor es la localización del lugar donde se va a realizar la siguiente operación de escritura o lectura. En la mayoría de los procesadores de texto aparece gráficamente como una raya horizontal que parpadea.

3. En la tercera sentencia, el especificador de formato %f indica que el valor que se visualiza es un valor real en coma flotante (la forma de escribir el número es similar a la notación científica).

Más adelante se verán más detalles acerca de la entrada y salida estándar de datos con formato.

## 2.9. El buen estilo de escribir en C

El estilo en que se escribe un programa, ya sea en C o en cualquier otro lenguaje de programación, es muy importante a la hora de depurar de errores el programa o simplemente a la hora de releerlo, entender su estructura o interpretar la utilidad del código al cabo de algún tiempo.

La utilización de comentarios dentro del programa, de elementos separadores de instrucciones, de identificadores significativos que ayuden a comprender la información contenida en los mismos es fundamental a la hora de hacer legible un programa. Especialmente importante es el uso de un buen estilo de *indentación* (del inglés *indentation*). La *indentación* consiste en sangrar el comienzo de las líneas del archivo (añadir tabuladores o espacios al principio de la línea antes del texto) para que reflejen la estructura de código.

Un ejemplo de buena escritura se muestra en el ejemplo 2.5:

Ejemplo 2.5: Buen estilo

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6 /* El programa principal empieza aquí */
7
8     printf("Un buen formato ");
9     printf ("puede ayudar ");
10    printf ("a entender un programa.\n");
11
12    printf("Un mal formato ");
13    printf ("puede hacer un programa ");
14    printf ("ilegible.\n");
15
16    return 0;
17
18 }/* main acaba aquí */

```

---

El programa que se muestra en el ejemplo 2.5 es una muestra de código de programa con un buen estilo. Es muy fácil ver rápidamente qué hace. Como el compilador ignora los espacios en blanco extra y los saltos de línea se tiene bastante libertad para darle un estilo adecuado.

Este mismo programa se podría escribir de la siguiente manera:

Ejemplo 2.6: Estilo confuso

---

```

1
2 #include <stdio.h>
3 int main(){printf("Un buen formato "); printf("puede ayudar ");
4 printf("a entender un programa.\n") ; printf("Un mal formato ");
5 printf("puede hacer ilegible "); printf("un programa.\n");return 0;}

```

---

Si se estudia el código del programa en el ejemplo 2.6, se tardará en observar que hace exactamente lo mismo que el anterior. Si bien al compilador no le importa el estilo del programa, dicho estilo si será importante cuando se quiera analizar, modificar o depurar los posibles errores del mismo. La forma de adquirir un buen estilo de escritura es la experiencia y la observación de otros códigos escritos en C por programadores más expertos. Además, existen unas normas más o menos establecidas con estilos de referencia ampliamente utilizados por los programadores. En este texto se va a emplear un estilo muy parecido al estilo conocido como *K&R*. Adicionalmente en algunos casos se añadirán comentarios en el cierre de llaves para indicar que es lo que cierra esa llave. Además se usarán 4 espacios para sangrar cada nivel de tabulación del código. Se utilizan espacios en vez de tabuladores por portabilidad debido a que el tamaño del sangrado de los tabuladores depende de cada editor de texto de manera que el mismo código fuente se puede ver con un buen estilo en un editor y no en otro.

## Ejercicios de programación

1. Escribir un programa que visualice el nombre del programador en la pantalla.
2. Modificar el programa anterior para que visualice la dirección y el teléfono añadiendo dos sentencias adicionales `printf`.
3. Verificar qué ocurre cuando en el programa del ejemplo 2.5 se modifica en la tercera instrucción `printf` el especificador de formato `%f` por `%d`.
4. Añadir la siguiente línea justo después de la última llamada a `printf` en el ejemplo 2.5 para comprobar que ocurre al ejecutar el programa:

```
printf("este es un tres%d\n y otro%f\n y otro%c", 3, 3.0, '3');
```

## **Parte II**

# **Elementos básicos de programación**



## Capítulo 3

# Primeros conceptos de programación

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Enumerar los elementos básicos del lenguaje C (Conocimiento)
2. Entender cómo se distinguen cada uno de estos elementos básicos (Comprensión)
3. Reconocer en un programa cualquiera de los elementos básicos (Aplicación)
4. Definir los conceptos de identificador, palabra reservada, constante literal, separador, signo de puntuación, espacio en blanco y comentario (Conocimiento)

### 3.1. Elementos básicos de un programa fuente

Al igual que en otros lenguajes de programación, en C se pueden distinguir elementos que constituyen el lenguaje en distintos niveles. Si se toma como referencia el lenguaje natural escrito estos niveles podrían ser las letras, las palabras y las oraciones. Naturalmente dentro de cada uno de estos niveles se pueden encontrar clasificaciones de estos elementos del lenguaje: consonantes y vocales, verbos, nombres, adjetivos u oraciones subordinadas. Los lenguajes de programación de alto nivel utilizan el lenguaje natural como referencia aunque sean mucho más formales y mucho más limitados. Así pues en el lenguaje de programación C también se tienen niveles y clasificación de elementos constitutivos del lenguaje y una serie de reglas sintácticas que permiten distinguir y clasificar dichos elementos.

Los archivos de código fuente y, por tanto, el lenguaje C es básicamente texto sin formato, es decir, es una secuencia de letras, signos y espacios en blanco que se pueden escribir o leer directamente.

En el nivel más básico del lenguaje se encuentran los siguientes elementos:

**Caracteres alfanuméricos**, Es decir, letras de la lengua inglesa, números y el carácter *guión bajo* o *subrayado* (`_` o *underscore*). La letras Ñ, ñ y las vocales con tilde no se pueden usar en el contexto de caracteres alfanuméricos<sup>1</sup>, que no sean texto libre (ver más adelante).

**Signos (de puntuación y operaciones)**, en inglés *punctuators*, es decir, caracteres que no son letras del abecedario o números del 0 al 9. No todos los signos de puntuación habituales en escritura se encuentran entre estos signos porque tienen significados especiales.

- Se excluye el guión bajo o subrayado (`_` o *underscore*) porque se asimila a una letra,
- Se excluye el carácter *barra invertida* (`\`, *backslash* o también carácter de escape<sup>2</sup>) porque es un modificador que afecta al significado del siguiente carácter y, por lo tanto, no se puede usar de forma aislada. Por ejemplo, cuando a la barra invertida le sigue un carácter con un significado especial se entiende ese carácter de manera literal, por ejemplo, `\"` corresponde al carácter *comilla doble* o `\\"` es un único carácter `\`. En general cuando a una barra invertida le sigue una letra u otro carácter hay que buscar el significado concreto por el contexto. Otro uso habitual es para representar un carácter que no se puede generar por teclado.

<sup>1</sup>Usualmente en informática, el término *alfanumérico* incluye todo tipo de caracteres que aparecen un texto. En este caso se hace una clasificación más precisa de los caracteres que aparecen en un archivo de código fuente.

<sup>2</sup>Esta denominación proviene del uso "antiguo" de este carácter en las impresoras para enviar caracteres que no debían ser interpretados como caracteres para imprimir sino como órdenes o comandos para la impresora.

- El carácter *sostenido* o *almohadilla* (# *number sign* o *pound sign*), tiene un significado especial que se explicará más adelante y, de hecho puede incluirse o eliminarse de la lista de signos de puntuación con parecidos argumentos.
- La lista completa de separadores tal y como aparece en el estándar ISO/IEC 9899 se muestra en la tabla 3.1.

Cuadro 3.1: Signos de puntuación en C

[ ]	( )	{ }	.	->						
++	--	&	*	+	-	~	!			
/	%	<<	>>							
<	>	<=	>=	==	!=	^		&&		
?	:	;	...							
=	*=	/=	%=	+ =	- =	<<=	>>=	&=	^ =	=
,	#	##								
<:	:>	<%	%>	%:	%: %:					

**Espacios blancos**, es decir, cualquier carácter textual que no tiene una representación gráfica sino que aparece como *blanco*. Por ejemplo: un tabulador, el espacio de la barra espaciadora, el carácter *retorno del carro* y/o el *salto de línea*.

Adicionalmente en ciertos contextos dentro de los archivos de código fuente puede aparecer texto libre, es decir, texto que el compilador no analiza sintácticamente y que, por tanto, se puede considerar fuera de las reglas del lenguaje de programación.

**Texto libre**, es decir, letras, números, signos, espacios en blanco o cualesquiera caracteres que aparezcan en el código fuente pero que no forman parte del lenguaje porque es información para el programa (datos literales alfanuméricos) o para el programador (comentarios). En el texto libre pueden aparecer las letras especiales del español y, en general, cualquier carácter que se pueda generar en el teclado. Cualquier carácter o combinación de caracteres y/o palabras dentro de un contexto de texto libre deja de tener su significado habitual dentro de lenguaje, excepto la barra invertida en el contexto de datos literales alfanuméricos. En todo caso, se verá más adelante.

Esta es una clasificación *morfológica*, es decir, se han dividido los caracteres de un texto escrito en lenguaje C en familias que se distinguen por su forma de manera intuitiva<sup>3</sup>. Ahora bien, esta clasificación tiene su importancia en las reglas sintácticas del lenguaje porque cada una de estas familias puede utilizarse o no en un determinado contexto. En concreto, se van a definir los conceptos correspondientes con las partes más elementales de un programa en C, especialmente el concepto de *palabra* y *separador* en relación con la clasificación anterior.

**Palabra**. Es un conjunto de caracteres alfanuméricos con un significado concreto dado por el lenguaje o por el contexto del programa. Las palabras **no** pueden empezar por un carácter numérico, pero si incluirlo en el medio. Ninguna palabra puede contener signos de puntuación, ni letras propias de la lengua española (Ñ, ñ, vocales con tilde), pero sí pueden incluir un carácter de guion bajo en cualquier posición.

**Separador**. Consiste en uno o varios signos de puntuación o espacios que separan palabras. En concreto, un signo de puntuación y un espacio en blanco separan dos palabras, pero también se separan con: un signo de puntuación con uno o varios espacios a los lados, varios espacios en blanco seguidos, una secuencia espacial de signos de puntuación, etc.

Existen algunos signos de puntuación que tienen un significado especial (los *espacios en blanco* nunca tienen un significado más allá de la utilidad de separar palabras) que introducen secuencias de caracteres que no son palabras, sino texto libre, formado por caracteres textuales de todo tipo. Se distinguen los siguientes:

**Comentarios**. Un comentario es un texto, es decir, un bloque de caracteres de todo tipo (texto libre, tal y como se han descrito más arriba) que no están sujetos a las normas sintácticas de C. El compilador ignora todos los comentarios independientemente de su posición dentro del código fuente. Los comentarios se escriben entre los separadores: /\* y \*/ de forma que /\* abre el bloque de comentario y \*/ lo cierra. Dentro de los comentarios se pueden usar las letras Ñ, ñ, vocales con tilde y, en general, cualquier carácter que se pueda generar con el teclado. Todos los caracteres dentro de un bloque de comentario pierden el significado especial que puedan tener.

<sup>3</sup>En realidad una definición precisa sólo puede darse mediante la enumeración de todos los elementos de cada familia que se no se ha incluido por brevedad.

**Cadena literal alfanumérica.** Es una secuencia de caracteres de cualquier tipo que el compilador interpreta como un texto, es decir, un dato textual, literalmente tal y como se haya escrito. Se escribe entre una pareja de comillas dobles (" quote, carácter situado sobre el 2 en el teclado en español). Por ejemplo, "hola" es la cadena literal *hola*. Una cadena literal alfanumérica se tiene que escribir en una única línea de texto del programa fuente. Si se desea utilizar el carácter *comilla doble* dentro de una cadena literal tiene que estar precedido de una barra invertida. Por ejemplo: "Juan dijo: \"hola\\"" es la cadena *Juan dijo: "hola"*. Si se quiere incluir un salto de línea en la cadena literal se deberá poner \n. Los caracteres y combinaciones de caracteres dentro de un bloque de información literal alfanumérica pierden su significado dentro del lenguaje. Las excepciones las constituyen el carácter barra invertida que se utiliza para incluir dentro del bloque literal caracteres especiales como la comilla doble, la propia barra invertida, los espacios en blanco especiales (tabulador y salto de línea) y los caracteres no imprimibles o que no se pueden producir en el teclado<sup>4</sup>.

**Letra literal.** Es un único carácter textual de cualquier tipo interpretado como el compilador como un dato textual formado por un único carácter. Se escribe entre dos comillas simples (' quote, carácter que comparte la tecla del cierre de interrogación ? en el teclado en español). Nótese que el uso de la barra invertida puede dar lugar a varios caracteres dentro de las comillas simples. Así, por ejemplo, 'n' significa la letra *n*, pero '\n' significa un único carácter *salto de línea* (la *n* de *newline*) y aparentemente aparecen dos letras entre las comillas.

**Comandos del preprocesador.** Se introducen mediante un carácter almohadilla (#). Su sintaxis y funcionamiento se explican más adelante.

Existe otro elemento básico del lenguaje que tiene difícil clasificación porque no es una palabra pero tampoco se introduce mediante una secuencia identificable de signos de puntuación. Son las constantes o literales numéricos, es decir, números que se escriben en C como tales dentro del programa y constituyen datos constantes del mismo. Se definen de la siguiente manera:

**Número literal.** Es una secuencia de caracteres que empieza por un carácter numérico y cuyo significado es un número entero o en coma flotante. Nótese que existen algunas representaciones numéricas especiales. Por ejemplo, el siguiente número es correcto: 1e3 y significa 1000. La e viene de exponente y aparece seguida del exponente del 10 en un número escrito en notación científica. Es posible utilizar sistemas de numeración especiales (binario, octal o hexadecimal) así como modificadores que imponen el tipo del número escrito. Así, por ejemplo, 0x11, es el número 17 ( $1*16+1$ ) expresado en hexadecimal o 2.0, es el número 2 considerado no como entero sino como un valor numérico en formato de coma flotante, es decir, 2,0. Para evitar errores accidentales es muy conveniente saber que los números enteros literales que empiezan por cero se interpretan en octal. Por ejemplo, el número 011 es el 9 decimal. La explicación de todas las posibles formas de escribir números de forma literal en C queda fuera del alcance de este texto. No obstante, los ejemplos que se han indicado son significativos y su uso es bastante común y útil.

## 3.2. Uso y significado de los elementos básicos del lenguaje

En este apartado se van a analizar los elementos básicos del lenguaje en cuanto a cómo se usan y qué significado tienen.

Los signos de puntuación pueden usarse con los siguientes significados:

**Marcadores de bloque, ámbito o agrupación de elementos.** Estas marcas se utilizan por pares. En este grupo se engloban las comillas, dobles ("") y simples ('), los paréntesis y las llaves, así como signos de puntuación formados por una secuencia de dos caracteres como puede ser el marcador de comentario /\* y \*/. La apertura y cierre de corchetes ([ ]) square brackets no se considera un marcador de bloque sino una operación. Existe un caso muy concreto, la llamada a una función, que se verá más adelante, donde la apertura y cierre de paréntesis también se considera una operación.

**Representación de operaciones.** Permiten representar tanto operaciones en el sentido matemático como otras que son propias de la sintaxis y herramientas de los lenguajes de programación. Tal es el caso de los signos de desigualdad, el igual, el tanto por ciento, la barra, el carácter *ampersand* (&), el asterisco, etc. Todos ellos representan operaciones.

<sup>4</sup>En una cadena literal se pueden incluir caracteres que normalmente no aparecen en el teclado, por ejemplo, caracteres chinos, escribiendo su código en UTF-8. Pruebe a mostrar por pantalla la cadena: "\xE0\xA4\x95\n" (es posible que no funcione en Windows porque la línea de comandos no funciona directamente con UTF-8)

**Separadores obligatorios de palabras.** En determinados contextos es necesario usar signos de puntuación concretos para separar palabras. Estos separadores no aportan significado, pero ayudan a que el lenguaje sea más claro. Dos ejemplos típicos de estos separadores obligatorios son las comas y el punto y coma que son necesarios y obligatorios en determinados lugares del código fuente. Nótese que no se puede usar cualquier separador para separar palabras. En general, el separador será un espacio en blanco o una secuencia de ellos salvo que se indique lo contrario.

Los caracteres alfanuméricos se usan para formar palabras. Las palabras del lenguaje se pueden clasificar en dos grandes grupos:

**Palabras reservadas.** Son palabras cuyo significado se define en lenguaje C para cualquier programa y ámbito y no se puede redefinir ni cambiar en ninguna circunstancia. Las palabras reservadas se muestran en la tabla 3.2 según el estándar C89, y en la tabla 3.3 las añadidas en el C99 (ISO/IEC 9899:1999).

Cuadro 3.2: Palabras reservadas en C89

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

Cuadro 3.3: Palabras reservadas añadidas en C99

inline	_Bool
main	_Imaginary
restrict	_Complex

Además de las palabras de esta lista, un compilador específico de C puede emplear unas cuantas más. En ese caso, es recomendable echar un vistazo a la documentación que acompaña al correspondiente manual del compilador. El significado de la mayoría de las palabras reservadas se explicará más adelante.

**Identificadores.** Son palabras cuyo significado lo establece el propio programador al escribir el código fuente mediante la escritura de un texto denominado *declaración*. Por supuesto, dado que los identificadores los define el programador pueden repetirse con distinto significado en otro programa, pero dentro del mismo programa y ámbito<sup>5</sup> deben estar definidos de manera unívoca. Los identificadores nombran de forma inequívoca algunos elementos característicos de los programas que se irán introduciendo posteriormente: variables, constantes, funciones, tipo de datos,... En C los identificadores deben cumplir las siguientes *reglas*:

1. Deben estar formados por caracteres alfanuméricos pero no pueden empezar por un dígito numérico, ni incluir signos de puntuación o espacios en blancos, ni tampoco pueden incluir la ñ o las vocales con tilde.
2. Sólo son significativos los 31 primeros caracteres. Si un identificador tiene más de 31 caracteres el resto es ignorado por el compilador.
3. Las *palabras reservadas* no se pueden emplear como identificadores.

Además, se deberían tener en cuenta las siguientes *recomendaciones*:

1. El carácter de guión bajo puede emplearse como parte de un identificador y facilita la lectura del código cuando se usa para separar en apariencia palabras del lenguaje natural dentro de un identificador. Los programadores lo emplean con frecuencia. Como la mayoría de desarrolladores de compiladores utilizan el carácter de guión bajo como primer carácter para identificadores de variables internas del sistema no se recomienda este uso para evitar coincidencias. Para su distinción, se reservan los identificadores con dos

<sup>5</sup>la definición de ámbito se dará más adelante, por ahora se usarán identificadores únicos en todo el programa.

caracteres de subrayado como iniciales para uso propio del compilador, así como los identificadores que empiezan por un carácter de subrayado y continúan con una letra del alfabeto mayúscula por lo que, en general, no se recomienda en ningún caso que los identificadores empleados en un programa empiezan por el carácter de guión bajo.

2. Se deben emplear identificadores descriptivos para las variables, funciones, etcétera. Esto facilita la lectura del código fuente del programa. Es más, en programas grandes y desarrollados por un equipo suele ser una buena costumbre evitar el uso de abreviaturas.

Una característica común a todas las palabras en C es que son distintas si cambia alguna letra de mayúscula a minúscula o viceversa. En inglés se dice que en C las palabras (identificadores y palabras reservadas) son *case sensitive*. En la lengua inglesa *case* es la palabra que se emplea para nombrar a la cualidad minúscula - mayúscula. Las palabras reservadas se escriben todas en minúsculas excepto algunos tipos especiales que se han incorporado en las últimas versiones del estándar de C.

### 3.3. Comandos del preprocesador

Los comandos más básicos y usados del preprocesador son el comando para incluir archivos y el comando para definir macros:

**#include** este comando del preprocesador seguido por la ruta de un archivo entre comillas dobles (" ") o ángulos (< >) inserta todo el contenido del archivo en el lugar donde aparece el comando **#include**. Nótese que el texto entre las comillas dobles no es parte del lenguaje de C y, por tanto, no es necesario incluir la barra doble invertida en la ruta de archivo cuando sea necesario especificar la ruta o vía de acceso completa<sup>6</sup>.

**#define macro texto** este comando tiene dos efectos. Por un lado hace que **macro** esté *definida* (ver ejemplo más adelante) y por otro provoca la sustitución de todas las ocurrencias de **macro** en el código fuente por el **texto** indicado. La inclusión de **texto** es opcional y, en ese caso, sólo se *define* la **macro**.

### 3.4. Estructura de un programa

Una vez que se conoce cuáles son los elementos más básicos del lenguaje se está en condiciones de estudiar la estructura de un programa a más alto nivel desde un punto de vista más formal.

Un programa en C está formado por uno o varios archivos de código fuente que se compilan y enlazan de forma conjunta. En adelante, se va a simplificar esta característica de C y se va a emplear un único archivo. Nótese que esta simplificación es muy fuerte y no se encuentra en ningún programa real medianamente complejo fuera del ámbito académico. Se introduce esta limitación para simplificar algunos conceptos que aparecen al usarse varios archivos.

Un programa en C está formado por la declaración de distintos elementos de programación: variables, constantes, tipos de datos y especialmente funciones. De todos ellos, el elemento más importante son las funciones. De hecho, la declaración e implementación de una función llamada **main** es indispensable para escribir el programa.

Se va a analizar la estructura de un programa a través del ejemplo 3.1.

Ejemplo 3.1: Estructura de un programa básico

---

```

1  /* Comentarios iniciales explicando
2   el contenido o detalles del archivo */
3
4 #include "stdio.h"
5
6 int main()
7 {
8     printf("hola mundo\n");
9     return 0;
10 }
11

```

---

Salida por pantalla en la ejecución:

hola mundo

<sup>6</sup>En todo caso se recomienda usar la barra normal, carácter /, ya que los compiladores para Windows reconoce este carácter como separador de carpetas en la especificación de una ruta.

Lo primero que hay que entender es que los comentarios y los comandos del preprocesador no forman parte de la estructura del programa:

1. Los comentarios directamente se eliminan antes de realizar ningún otro proceso.
2. Los comandos del preprocesador se aplican para realizar algún tratamiento al texto antes de pasar a la compilación y luego se eliminan.

En el ejemplo 3.1, aparecen distintos comentarios y el comando `#include`. El comando `#include` se sustituye por el contenido del archivo correspondiente. En este caso el archivo `stdio.h` tiene la declaración de la función `printf` y del resto de funciones de la entrada y salida estándar de C, por ese motivo, es necesario para compilar. Así pues, al compilar, en el lugar del comando `#include` se encontrarán una serie de declaraciones de funciones<sup>7</sup>.

El código fuente queda como una serie de declaraciones de funciones seguido de la declaración e implementación de una función importante, la función `main`. Esta función es fundamental porque es el punto de entrada al programa, es decir, la ejecución del programa equivale a ejecutar la función `main`.

Aunque la estructura de declaración e implementación de funciones se verá más adelante, se adelantan los siguientes conceptos:

1. La *declaración* de la función es la línea de texto donde aparece el nombre de la función, es decir, `main`, hasta la apertura de la llave. En la mayoría de los compiladores es obligatorio que la función `main` devuelva un entero, el `int` inicial.
2. La *implementación* de la función es el código fuente que se encuentra en la apertura y el cierre de llave.
3. Dentro de la implementación de la función se pueden encontrar, en primer lugar, nuevas declaraciones de tipos, constantes, variables, pero no de nuevas funciones. Después de las declaraciones pueden aparecer sentencias que indican los pasos que el programa realizará secuencialmente al ejecutar la función. Las sentencias se distinguen porque acaban en punto y coma, excepto en algunos casos donde se agrupan sentencias utilizando llaves y no se escribe el punto y coma final.
4. En algunas versiones de C se pueden mezclar declaraciones y sentencias (C99), pero en otras no (ANSI). Para asegurar una mayor portabilidad es conveniente poner siempre las declaraciones primero y luego las sentencias y así se hará en adelante.

En conclusión, se puede resumir que para estudiar la estructura de un programa primero se deben quitar todos los comentarios, luego se deben aplicar los comandos del preprocesador y una vez realizadas estos dos pasos lo que queda debe ser una secuencia de declaraciones de tipos, datos (variables y constantes) y funciones. Entre las funciones, una imprescindible, la función `main`, que es el punto de entrada al programa. Dentro de las funciones se escribirán, en primer lugar las declaraciones internas propias de la función y luego la secuencia de pasos, mejor llamados sentencias, que se realizan al ejecutarse una función. Las sentencias de la función `main` son las que constituyen el programa de C.

Aunque estrictamente no forma parte del lenguaje, en ocasiones los programadores usan las funcionalidades del preprocesador para trabajar con constantes (normalmente constantes literales) mediante el uso de macros. Este es un uso que se desaconseja porque escapa al control del compilador y puede dar lugar a confusiones y errores. Normalmente el error proviene de que el preprocesador no comprueba de ninguna manera la definición o redefinición de una macro, simplemente sustituye con el último valor definido. En todo caso se podría incluir un tipo de constante adicional.

## Ejercicios de programación

1. Escribir, compilar y ejecutar el siguiente programa. Explicar el comportamiento del programa a partir de las aclaraciones que aparecen en los comentarios.

Ejemplo 3.2: Precaución en el uso de macros (`define`)

---

```

1
2 #include <stdio.h>
3
4 /* El define solo sustituye el texto, 3+2,
5    de la macro (CINCO) en el resto del
6    programa */

```

<sup>7</sup>se recomienda buscar y abrir el archivo `stdio.h` mediante un editor de texto para comprobar su contenido en el que debe aparecer la declaración de `printf`. Su ubicación depende del sistema operativo y, en ocasiones, del compilador.

```
7 #define CINCO 3+2
8
9 /* La constante es un dato que se guarda
10 como tal en el código del programa */
11 const int cinco = 3+2;
12
13 int main()
14 {
15     printf("define: %d \n", 3*CINCO);
16     printf("constante: %d \n", 3*cinco);
17     return 0;
18 }
```

---

2. Escoger cualquiera de los programas de ejemplo y señalar (por ejemplo con marcadores de distintos colores) los siguientes elementos básicos de programación: palabras reservadas, identificadores, constantes literales alfanuméricas, números literales y signos de puntuación.
3. Introducir comentarios en cualquiera de los programas de ejemplo. Compruebe si los comentarios se pueden insertar en cualquier lugar del código fuente.
4. Comprobar si una constante literales alfanumérica se puede escribir en 2 líneas de código fuente.
5. Comprobar si un comentario se puede escribir en varias líneas de código fuente.
6. Comprobar si una macro (#define) se puede escribir en varias líneas de código fuente. Buscar una manera de escribir una macro en varias líneas de código.
7. En cualquiera de los programas de ejemplo, sustituir `main` por `Main` y explicar el resultado obtenido al intentar compilar el programa.
8. En cualquiera de los programas de ejemplo añadir y quitar caracteres de espacio en blanco (incluyendo saltos de línea y tabuladores) en distintas partes del programa y luego compilar. Explicar cuando un espacio en blanco es imprescindible y cuando no.
9. Incluir vocales con tilde o la ñ en cadenas literales alfanuméricas. Si se trabaja en Windows, explicar el resultado. Utilizar las opciones de codificación de texto del Notepad++ para modificar el resultado del programa. En GNU-linux utilice las opciones de codificación de la *shell* para modificar el resultado.
10. Utilizar la opción `-E` del compilador `gcc` para examinar el resultado de aplicar el preprocesador a un archivo de código fuente. Por ejemplo, el comando: `gcc -E codigo.c -o codigo.txt` ejecuta el preprocesador sobre `codigo.c` y guarda el resultado en `codigo.txt`.
11. Escribir un programa que muestre por pantalla la siguiente cadena literal alfanumérica: "`\x41\n`". Explicar el resultado de la ejecución del programa. Probar con otros números distintos del 41. Probar con la cadena: "`\x4A\n`". Nótese que los números utilizados deben tener siempre dos cifras.
12. Encontrar la codificación UTF-8 para una vocal con tilde (buscando en la Web).



# Capítulo 4

## Datos y tipos básicos

Objetivos:

1. Comprender el concepto de *dato* y de *tipo* (Comprendión)
2. Distinguir el concepto de *constante* y de *variable* (Conocimiento)
3. Describir los tipos de dato básicos en C y su formato de representación (Conocimiento)
4. Describir la forma de la declaración de constantes y variables de los tipos de dato básicos (Conocimiento)
5. Entender en qué consiste la conversión de datos (Conocimiento)
6. Seleccionar el tipo de dato adecuado para la declaración de variables (Aplicación)
7. Realizar declaraciones de constantes y variables de los tipos adecuados (Aplicación)
8. Distinguir y clasificar los datos presentes en un programa (Comprendión)

El presente capítulo introduce uno de los elementos básicos de programación: los datos y sus tipos y cómo se pueden emplear en un programa en C. También se introduce un concepto fundamental: la variable como la forma de dato más sencilla. Luego se clasifican los datos más simples según su naturaleza y formato de representación (su tipo): enteros, reales, etc.

### 4.1. Datos

En informática, se denomina dato<sup>1</sup> a un fragmento de información elemental, codificada, que se guarda en un espacio de memoria conocido y limitado.

La información, antes de ser utilizada por el ordenador, necesita ser codificada, es decir, transformada en una serie de símbolos procesables por el ordenador (en realidad ceros y unos, las cifras binarias), con un tamaño de ocupación de memoria y un conjunto de operaciones permitidas perfectamente preestablecidas.

Un programa no maneja información en general sino datos, es decir, información concreta, individualizada y con un significado preciso. Por ejemplo, de un programa concreto sobre frutas no se dirá que maneja “números” en general sino que trabaja con *tres números enteros*: el número de naranjas, peras y manzanas de una cesta. La concreción, individualidad y significado de cada uno de estos tres números los convierte en datos. Cuando estos datos se codifican en binario y se almacenan en la memoria del ordenador ya se tienen los datos de un programa.

### 4.2. Tipos de datos

Como se ha indicado, la información en un ordenador (digital) está codificada, en último término, en forma de unos y ceros (binario) y en un tamaño conocido expresado en bits o bytes. Ambas características son importantes:

1. La codificación servirá para:
  - a) interpretar el dato almacenado

---

<sup>1</sup>En el documento que describe el estándar de C este concepto se indica con la palabra *object*, el concepto de dato normalmente es más amplio de la descripción que se indica en este texto.

b) determinar las operaciones aplicables al dato

2. El tamaño servirá para organizar la memoria, es decir, distribuir, colocar y localizar distintos datos en memoria.

La unión de ambas características recibe el nombre de *tipo de dato*. El tipo de cualquier dato que maneje un programa es una característica muy valiosa. De hecho, muchos errores que se encuentran en programas realizados con lenguajes de programación donde los tipos no se definen de forma explícita se deben a fallos en la determinación del tipo de un dato.

El lenguaje C pertenece a la familia de los lenguajes *fuertemente tipados*, es decir, lenguajes donde el programa determina sin ambigüedad el tipo de todos los datos que maneja. Otros lenguajes fuertemente tipados son Java, C++, Visual Basic, ... Un ejemplo típico de lenguaje no tipado es el lenguaje propio de Matlab.

## 4.3. Los datos clasificados por posibilidad de modificación

Se pueden clasificar los datos de un programa según la posibilidad de modificarlos durante la ejecución del mismo. Naturalmente cualquier dato se puede introducir como nuevo dato en un programa, así como modificar, cambiar o eliminar datos anteriores mediante la edición del código fuente del programa. No obstante, dado un código fuente determinado y fijo, se pueden encontrar datos cuyo valor puede variar durante la ejecución del programa y otros que en los que no. De esta manera, se puede introducir la primera clasificación de datos en *constantes*, cuyo valor no se puede modificar durante la ejecución del programa y *variables*, cuyo valor puede cambiar durante la ejecución del programa. Para cambiar el valor de las variables se usa un operador que se llama asignación y cuya explicación detallada se encuentra en el capítulo siguiente.

### 4.3.1. Constantes

En C se distinguen dos tipos de datos constantes, las *constantes literales* y las *constantes con identificador*.

**Constantes literales:** son aquellas cuyo valor aparece como tal en el código fuente: números y cadenas de caracteres que se pueden observar tal cuál en el texto del código fuente. Estas constantes se insertan directamente sobre el código objeto generado por el compilador y naturalmente, una vez generado, no se puede cambiar durante la ejecución del programa. Las constantes literales tienen asociado un tipo de dato de forma implícita. No obstante, salvo que se utilicen modificadores especiales en la escritura de la constante, este tipo no se encuentra de forma explícita en el código fuente. El compilador es el encargado de elegir el tipo de dato según la forma en que se ha escrito la constante.

**Constantes con identificador:** son aquellas que aparecen declaradas al estilo de las variables, pero con el modificador `const` en la declaración y siendo obligatoria la inicialización de su valor, por supuesto utilizando una expresión constante. El compilador impide el uso de constantes con identificador en el lado izquierdo de una asignación, por lo demás son muy similares a las variables.

**Macros:** como se ha visto cuando se define una macro su texto se sustituye en el código fuente en todos los lugares donde aparezca. Si la macro se define como una constante literal, a una expresión constante o a una constante con identificador funcionará como si fuese una constante. Este es un uso muy frecuente de las macros.

Cuando se escriben constantes literales de números enteros se pueden utilizar sistemas de numeración distintos del decimal, se usan los prefijos `0x` para enteros en hexadecimal o `0`, para enteros en octal. Así, el número `0x10` es el 16 decimal y el `010` es el 8 decimal.

### 4.3.2. Variables

Los datos más habituales en un programa son las variables, es decir, son datos cuyo valor se puede modificar durante la ejecución de un programa mediante un operador específico: la asignación.

Entre las características de las variables se destacan las siguientes:

1. Una variable tiene un espacio reservado en memoria de acuerdo con el tipo al que pertenece. El momento en el que se produce la reserva (y posterior liberación) de la memoria depende de otras características de las variables que se verán más adelante.
2. Una variable siempre guarda un único valor dentro de los posibles para su tipo de dato. Una variable no puede dejar de tener valor. Incluso cuando no esté inicializada, tendrá un valor desconocido, posiblemente aleatorio. Una variable jamás puede tener más de un valor. La actualización o reasignación de valor, es decir, la modificación de la variable, provoca la pérdida definitiva del valor anterior sin posibilidad alguna de recuperación.

## 4.4. Declaración de constantes y variables

La sintaxis simplificada de declaración de una constante con identificador de tipo básico es:

```
const Tipo_de_dato id_const = valor;
```

Donde **const** es una palabra reservada que indica que se declara una constante, **Tipo\_de\_dato** es el tipo de la constante, **id\_const** es su nombre o identificador y **valor** es el dato que se guarda en la constante. La declaración acaba con punto y coma.

La sintaxis simplificada de la declaración de una variable de tipo de dato básico es:

```
Tipo_de_dato id_var ;
```

Por ejemplo:

```
double x;
int n;
```

Como se ha comentado anteriormente, las variables se declaran, es decir, el programador establece, en una línea de código fuente como la anterior, que va a utilizar un dato. En la declaración de una variable estática aparecen los dos conceptos clave asociados a una variable: su tipo de dato y su identificador.

1. El tipo de dato se declara explícitamente. Como se ha dicho anteriormente C es un lenguaje fuertemente tipado, entre otras cosas, esta característica implica que todas las variables tienen obligatoriamente un tipo. Para la declaración de una variable cuyo tipo de dato es uno de los básicos simplemente se debe sustituir **Tipo\_de\_dato** por el tipo de dato elegido para la variable.
2. El identificador de la variable es el nombre que recibe la variable para que el programador utilice ese dato en el resto del programa. Se dice que es un identificador porque es único, es decir, no puede haber posibilidad de confusión con otro dato. Normalmente, los programadores utilizan como identificador una descripción muy breve del significado del dato, una palabra, dos como mucho. Cuando se declaran los identificadores de esta manera el programa resulta más fácil de leer y entender. En la declaración anterior hay que sustituir **id\_var** por una palabra que cumpla las restricciones aplicables a los identificadores.

La declaración anterior de variables de tipo de datos básicos se puede ampliar para incluir más variables de mismo tipo:

```
Tipo_de_dato id_var1, id_var2, id_variableN;
```

Donde **id\_var1**, **id\_var2**, **id\_variableN** son identificadores de variables del mismo tipo de dato. Se pueden declarar tantos identificadores de variables como se desee separadas por comas. La declaración se termina en punto y coma.

Las variables así declaradas no reciben un valor conocido en el momento en el que se reserva su sitio en memoria. En un programa no es conveniente dejar nada al azar o con un valor desconocido que pueda causar errores. El uso de un valor imprevisto en un programa puede ocasionar que el programa tenga errores de funcionamiento graves. Por esta razón es frecuente inicializar las variables, es decir, asignar un valor inicial a la variable en el momento de la declaración. De esta manera la primera vez que se utilice la variable su valor está definido y es conocido. La sintaxis de la declaración se completa de esta manera:

```
Tipo_de_dato id_var = valor;
```

donde **= valor** se dice que es la *inicialización* de la variable y **valor** es una constante literal (lo más frecuente) o una expresión. En cualquier caso, **valor** debe ser del mismo tipo de dato que la variable o ser convertible al tipo de dato de la variable.

Por ejemplo:

```
double z = 136.25;
int a = 1250;
```

Cuando se declaran varias variables en la misma declaración se pueden inicializar cualquiera de ellas a nuestra discreción, simplemente añadiendo la inicialización correspondiente.

Es una buena costumbre inicializar los valores de las variables. Afortunadamente, cuando se compila el programa con las opciones que sirven de referencia, el compilador avisa si se emplea una variable sin inicializar (un aviso o *warning*, no un error). De hecho el programa se compila y se puede ejecutar y, naturalmente, el valor de la variable es totalmente imprevisible.

## 4.5. Tipos básicos de datos

La información más básica que se puede representar en un ordenador son números y letras. Sea cual sea el lenguaje de programación o sistema estas dos familias de datos siempre están presentes y son los datos elementales sobre los que se construyen datos más complejos o avanzados.

### 4.5.1. Tipos de datos numéricos

En informática, de forma natural, los datos numéricos y más concretamente los tipos de datos numéricos se relacionan con los conjuntos de números que se estudian en matemáticas. No obstante, hay ciertas propiedades particulares que son características de los números que se manejan en los lenguajes de programación de propósito general, es decir, lenguajes no del álgebra computacional<sup>2</sup>:

1. En un lenguaje de programación no existen números en forma de símbolo, es decir, no existe  $\Pi$ , ni el número  $e$ , ni tampoco la unidad imaginaria,  $i$ .
2. No existen “incógnitas” ni variables sin valor. En matemáticas, es frecuente encontrar símbolos que representan números en todo tipo de expresiones, pero que no tienen un valor concreto. Esto no es posible en un lenguaje de programación no algebraico.
3. Todos los números representados por los tipos simples tienen unas limitaciones de precisión y rango debidas a la limitación de tamaño y codificación binaria propia de los tipos de datos del lenguaje de programación.
4. Las operaciones numéricas están sujetas a errores numéricos, por ejemplo, de redondeo. Especialmente hay que tener en cuenta que las operaciones se realizan en la codificación propia de los ordenadores, es decir, en binario.

Salvadas estas diferencias, en informática, se encuentran los siguientes conjuntos de números en forma de tipos de dato numéricos:

1. Naturales con el cero. O como se suelen denominar en informática: *enteros sin signo*.
2. Enteros. En informática se resalta que se codifica el signo y se les suele denominar *enteros con signo* o simplemente enteros.
3. Reales. Los números reales se encuentran en informática aproximados (redondeados o truncados) a un cierto valor numérico concreto según el tipo de dato empleado, su precisión y rango de representación. En realidad, es más correcto decir que se usan números con coma decimal.

Normalmente, estos tipos de datos se encuentran codificados respectivamente en:

1. Base 2 o binario natural. Simplemente mediante el correspondiente cambio de base, el número es el mismo, pero expresado en binario y añadiendo los ceros necesarios a la izquierda para completar el tamaño dado por el tipo de dato empleado.
2. Complemento a 2. Para codificar el signo de los números enteros el sistema más frecuente por sus ventajas en la implementación de las operaciones aritméticas es el C2. Igual que el caso anterior se añaden los dígitos necesarios a la izquierda para completar el tamaño de la representación del tipo de dato elegido.
3. Coma flotante. Se utiliza este sistema con algunas variaciones para aproximar los números reales y los números con coma decimal. Cada tipo de dato reparte los bytes empleados en mantisa y exponente, normalmente cada una de ellas en C2. Los números en coma flotante permiten un rango muy amplio de representación, es decir, permite almacenar números muy grandes y muy pequeños. En cualquier caso están sujetos a una precisión dada por una limitación de cifras significativas, es decir, se pueden representar números muy grandes, intermedios o muy pequeños. Pero, en todos los casos, las cifras del número serán las mismas dando lugar a una precisión distinta para cada valor. Esta precisión viene dada por la magnitud de la última cifra significativa cuando se tiene en cuenta el valor del exponente.

Naturalmente los tipos numéricos suelen tener implementadas las operaciones que les son propias, es decir, todas las aritméticas y en el caso de los números con coma decimal aquellas propias de los números reales: funciones trigonométricas, logarítmicas, etc. Es importante tener en cuenta que los tipos enteros y reales en coma flotante tienen una implementación distinta de las operaciones aritméticas. Esta diferencia se nota principalmente en la división: en los

---

<sup>2</sup>Los sistemas de álgebra computacional manejan los números de forma distinta y más compleja y se suelen considerar a distinto nivel de los lenguajes de programación habituales.

tipos enteros es la división entera (resto y cociente de la división entera) y en los tipos en coma flotante es la división decimal o fraccionaria.

De acuerdo con la clasificación descrita, los tipos básicos de C son:

**Enteros sin signo:** `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`.

**Enteros con signo:** `signed char`, `short int`, `int`, `long int`.

**Reales en coma flotante:** `float`, `double`, `long double`.

Estos tipos se distinguen entre sí por el tamaño y, por tanto, por su rango de representación y precisión.

En los tipos enteros se puede abreviar el nombre del tipo eliminando la palabra `int`. De esta manera el tipo `long int` es exactamente igual que `long`. Por otro lado, en la declaración de tipos con signo ya se ha abreviado la palabra `signed` que se podría utilizar en las declaraciones, pero no es necesaria. Así, los tipos: `signed long int`, `long int`, `signed long` o `long` son exactamente el mismo.

#### 4.5.2. Tipos de datos alfanuméricos

La otra gran familia de tipos de datos en los lenguajes de programación son los datos alfanuméricos. De ellos, el tipo más sencillo es el carácter que equivale en general a una letra, un dígito del 0 al 9, un signo de puntuación o un símbolo gráfico.

Los tipos básicos en C para datos alfanuméricos son:

**Alfanuméricos:** `char`, `unsigned char`, `signed char`.

Se puede mostrar el valor de un dato alfanumérico por pantalla utilizando el especificador de formato `%c` en la función `printf`. Los tipos alfanuméricos empleados como tales son totalmente equivalentes.

La codificación de los tipos alfanuméricos se realiza mediante una serie de estándares que son independientes de los lenguajes de programación, de los sistemas operativos o incluso de la propia máquina.

La forma de codificación de todos estos sistemas consiste en la ordenación de las letras y símbolos gráficos en tablas donde se establece una equivalencia entre el símbolo (letra, número o carácter gráfico) y un número entero.

La primera de estas tablas fue el estándar o tabla ASCII y servía para codificar las letras del alfabeto inglés junto con algunos símbolos especiales en 7 bits que en la implementación se ampliaban a 8 bits para formar el byte. De hecho, la ampliación a 8 bits sirvió para incluir letras propias de otros idiomas, por ejemplo, el mismo español, mediante lo que se conocía por ASCII ampliado.

La progresiva inclusión de nuevos idiomas a las tablas que sirven para codificar texto y la difusión de sistemas informáticos a nivel mundial ha obligado a un complejo proceso de estandarización (veáse la codificación de texto en el material de consulta).

#### 4.5.3. Comentarios sobre tipos alfanuméricos

En la clasificación anterior hay algunos tipos cuya adscripción debería sorprender. Se debe tener en cuenta que C es un lenguaje de programación muy próximo a la máquina, en el sentido de que permite representar o manipular datos teniendo en cuenta la representación de los mismos en el hardware. Así se puede observar que:

1. Los tipos `signed char` y `unsigned char` aparecen como enteros con y sin signo. En la práctica esto significa que podemos sumar, restar, multiplicar o convertir a y desde entero estos datos sin ningún tipo de problema. Estas posibilidades no se encuentran en otros lenguajes de programación porque sencillamente no tiene sentido una suma de letras. Sin embargo en C es posible, incluso corriente, encontrar números enteros guardados en estos tipos. Una de las razones para ello es el tamaño habitual de la representación de estos tipos: 1 byte. Este tamaño para representar números enteros sólo se puede utilizar mediante los tipos `char`. A pesar de todo esta característica de C es conveniente usarla con precaución.
2. El tipo `char` también aparece clasificado dentro del tipo entero más general. La idea es la misma que para las variantes con y sin signo. Para C todo lo que no es un número en coma flotante es un número entero, tal y como ocurre realmente en los ordenadores cuando manejan estos datos a bajo nivel.
3. Por otro lado, C no establece ni presupone una codificación de texto predeterminada. De nuevo esta flexibilidad implica que el programador tiene que ser consciente de cuál es la codificación que se está empleando, tanto para escribir el código fuente como en el momento de la ejecución del programa. A partir de ahora se asumirá que el juego de caracteres empleado es el adecuado tanto para el código fuente como para el momento de la ejecución del programa y, en todo caso, se evitarán en la medida de lo posible los caracteres que no forman parte del ASCII reducido (caracteres del 0 al 127).

Algunos programadores (librerías y sistemas) para distinguir las situaciones donde utilizan los tipos `char` y poder codificar números enteros empleando un sólo byte definen un tipo `BYTE` como equivalente al correspondiente `char` normalmente sin signo.

Para inicializar o dar valor a datos alfanuméricos se puede utilizar como valor un carácter constante literal, es decir, un carácter entre comillas simples, por ejemplo: `'a'` o un número entero dentro del rango, por ejemplo, 97. Se puede usar un valor entero porque los enteros son convertibles a los tipos alfanuméricos. Naturalmente, también se puede hacer a la inversa e inicializar el valor de un entero con una letra. Los enteros y los datos de tipo alfanuméricos son convertibles entre sí.

#### 4.5.4. El tamaño de los tipos básicos en C

Al contrario que en otros lenguajes de programación el tamaño de los tipos básicos en C no lo determina el lenguaje sino el sistema, es decir, el conjunto máquina (procesador) + sistema operativo. En algunos casos también puede depender del compilador elegido. Esta característica hace que C sea mucho más flexible, pero también obliga a conocer los sistemas donde se va a ejecutar el programa y a prever posibles restricciones a la capacidad de representación de los tipos de datos empleados. Naturalmente, como se ha indicado previamente, dado un sistema, el tamaño de los tipos de datos está fijado sin posible ambigüedad porque el compilador usa esta información para gestionar la reserva de los datos en memoria.

Una gran ayuda en la gestión del tamaño de los tipos es el operador `sizeof`. Aunque los operadores se tratan en el capítulo correspondiente este operador es tan característico y está asociado de tal forma a los tipos de datos que resulta conveniente anticipar su explicación. El operador `sizeof` se puede aplicar a una constante, a una variable o a un tipo de dato. Su resultado es un número entero sin signo que indica el tamaño de la representación binaria en bytes de la constante, variable o tipo de dato. El resultado de `sizeof` se puede utilizar para realizar determinadas comprobaciones sobre los tipos de datos más críticos en un programa y también para realizar reservas de memoria más flexibles. En el ejemplo 4.1, se muestra un programa que escribe por pantalla el tamaño de todos los tipos simples. Se recomienda que el lector ejecute este programa en su sistema para conocer las características de su sistema.

Ejemplo 4.1: Tamaño de tipos de datos básicos

---

```

1  #include <stdio.h>
2
3
4 int main()
5 {
6     printf("char      ocupa %d bytes\n", sizeof(char) );
7
8     printf("short int  ocupa %d bytes\n", sizeof(short int) );
9     printf("int        ocupa %d bytes\n", sizeof(int) );
10    printf("long int   ocupa %d bytes\n", sizeof(long int) );
11
12    printf("float      ocupa %d bytes\n", sizeof(float) );
13    printf("double     ocupa %d bytes\n", sizeof(double) );
14    printf("long double ocupa %d bytes\n", sizeof(long double) );
15
16    return 0;
17 }
```

---

Salida por pantalla en la ejecución:

```

char      ocupa 1 bytes
short int  ocupa 2 bytes
int       ocupa 4 bytes
long int   ocupa 4 bytes
float      ocupa 4 bytes
double     ocupa 8 bytes
long double ocupa 12 bytes
```

En la especificación del estándar se establece que los tipos enteros con y sin signo tienen obligatoriamente el mismo tamaño en memoria y por eso no se han mostrado en el programa. Por otro lado, es obligatorio que los tipos de datos se ordenen por tamaño, es decir, es obligatorio que el `short int` sea más pequeño o igual que el `int` y éste menor

Cuadro 4.2: Sistema de 32 bits

Nombre del tipo	Bytes	Intervalo
<code>char</code>	1	
<code>signed char</code>	1	-128 a 127
<code>unsigned char</code>	1	0 a 255
<code>short</code>	2	-32,768 a 32,767
<code>unsigned short</code>	2	0 a 65,535
<code>int</code>	4	-2,147,483,648 a 2,147,483,647
<code>unsigned int</code>	4	0 a 4,294,967,295
<code>long</code>	4	-2,147,483,648 a 2,147,483,647
<code>unsigned long</code>	4	0 a 4,294,967,295
<code>float</code>	4	+/-3.4E+/-38 (7 dígitos)
<code>double</code>	8	+/-1.7E+/-308 (15 dígitos)
<code>long double</code>	10	+/-1.2E+/-4932 (19 dígitos)

o igual que el `long int`. No obstante, frecuentemente se encuentra que el tamaño de algunos tipos es el mismo. De hecho, es corriente que los tipos `int` y `long int` tengan el mismo tamaño aunque esté desaconsejado.

Cuadro 4.1: Sistema de 16 bits

Nombre del tipo	Bytes	Intervalo
<code>char</code>	1	
<code>signed char</code>	1	-128 a 127
<code>unsigned char</code>	1	0 a 255
<code>short</code>	2	-32,768 a 32,767
<code>unsigned short</code>	2	0 a 65,535
<code>int</code>	2	-32,768 a 32,767
<code>unsigned int</code>	2	0 a 65,535
<code>long</code>	4	-2,147,483,648 a 2,147,483,647
<code>unsigned long</code>	4	0 a 4,294,967,295
<code>float</code>	4	+/-3.4E+/-38 (7 dígitos)
<code>double</code>	8	+/-1.7E+/-308 (15 dígitos)
<code>long double</code>	10	+/-1.2E+/-4932 (19 dígitos)

Naturalmente el tamaño de los tipos de datos tiene una repercusión directa en su capacidad para representar datos concretos. A mayor tamaño, mayor rango o precisión de los posibles valores que se pueden guardar. No debe olvidarse que un dato, restringido a un tamaño en memoria, tiene un número finito de posibles valores dado por  $2$  elevado al número de bits. A modo de referencia en las tablas 4.1 y 4.2 se muestran los tamaños, rangos de representación y precisión de los tipos básicos en C en dos sistemas: uno de 16 bits (en la actualidad apenas se encuentran) y otro en 32 bits. Nótese que la única diferencia entre ambas listas se encuentra en el tamaño de los tipos de dato `int` (con y sin signo).

Aunque no es obligatorio el estándar recomienda que el tipo entero `int` tenga el mismo tamaño que los registros del microprocesador. Esta característica tiene como consecuencia que el tipo de dato `int` sea especialmente eficiente y la elección preferente de los programadores cuando se necesita un dato entero.

De esta manera, en un sistema de 16 bits el tipo `int` ocupa 2 bytes, en uno de 32 bits ocupa 4 bytes y en uno de 64 bits debería ocupar 8 bytes.

#### 4.5.5. Selección del tipo de dato adecuado

No existe una regla fija o un conjunto de criterios establecidos para decidir cuál es el tipo de dato adecuado para representar los datos de un problema. Además, algunos programas pueden necesitar representaciones especiales de tipos de datos diseñados precisamente para resolver un problema muy concreto. Normalmente el sentido común del programador y su capacidad de prever el uso de los datos que maneja el programa deben ser suficientes para realizar una elección correcta. No obstante, a continuación se proponen una serie de normas muy elementales para la elección de tipos de datos:

1. En primer lugar se debe considerar si el dato es un número<sup>3</sup> u otro tipo de información.
2. Si es un número se debe analizar si el número tiene parte decimal, en cuyo caso será un dato en coma flotante, o no tiene parte decimal, en cuyo caso será un número entero.
3. Una vez fijado el grupo numérico se debe prever el rango y precisión necesarios para representar los posibles valores del dato y elegir un tipo de dato con el tamaño suficiente para representarlo. En la actualidad escoger datos de gran tamaño no es un problema para la rapidez del cálculo y el uso de memoria en un ordenador convencional, incluso de uso doméstico. Solo en aplicaciones con muchos datos (estamos hablando de miles de datos como mínimo) se debe estudiar la utilización de tipos de datos más pequeños. En todo caso, para la mayor parte de programas la elección de `int` para números enteros y `double` para números con parte decimal suele ser la mejor y la más empleada cuando no hay necesidades especiales.
4. Si la información no es numérica en primer lugar se debe analizar si es un texto, en cuyo caso se utilizarán tipos de datos alfanuméricos.
5. Si la información tiene un número finito de alternativas se debe considerar el empleo de un número entero y la utilización de una tabla de equivalencia entre los posibles valores o alternativas de la información y el número entero.
6. Otra información normalmente corresponderá con tipos de datos compuestos que se verán más adelante.

Aunque estas normas parecen obvias hay datos cuyo análisis para la determinación del tipo puede no resultar evidente. Por ejemplo, ¿el dinero tiene decimales? ¿la unidad imaginaria es un número? ¿y la  $x$  que se escribe en los polinomios? ¿una fecha es un único número entero o está compuesta de varios? Se anima al lector a tratar de analizar estos ejemplos para encontrar el tipo adecuado.

En conclusión, normalmente la elección del tipo de dato se reduce a elegir entre `int`, `double` o `char`, donde en primer lugar se decide si es un número o una letra y en el primer caso si tiene decimales. Se recuerda que el tipo `int` es especialmente eficiente, de manera que es la mejor elección en muchos casos donde se quiere representar un número u otro tipo de información codificable como un entero.

Por otro lado no es corriente encontrarse con información alfanumérica que conste de una sola letra, sino a través de su tipo compuesto: las cadenas alfanuméricas. Hasta el capítulo donde se van a tratar las cadenas alfanuméricas en detalle, se usarán estos datos sólo a través de constantes literales entre comillas dobles. Al usar de esta manera las cadenas alfanuméricas, el compilador determina su tipo de dato y las guarda en memoria sin necesidad de una declaración explícita.

#### 4.5.6. Conversión de datos

Como se ha indicado anteriormente todos los datos de un programa en C tienen asociado un tipo de dato dado por el programador explícitamente o determinado por el compilador por la aplicación de una serie de normas. Ahora bien, el tratamiento de estos datos obliga a realizar operaciones sobre los mismos con las que se obtienen nuevos datos. Es decir, habitualmente es necesario mezclar los datos. Naturalmente al hablar de datos se entiende que son tanto constantes como variables.

Tarde o temprano, en este proceso de mezcla o tratamiento de los datos se tendrá que guardar un valor determinado en una variable o se necesitará realizar una operación sobre dos datos que no tienen el mismo tipo. En estos y otros casos cabe preguntarse si es posible la conversión de un dato a otro tipo de dato diferente al original. En el estándar de C se dan una serie de normas muy complejas para determinar cómo y con qué restricciones se pueden realizar las conversiones entre tipos de datos. La descripción detallada de estas normas quedan fuera del alcance de este texto. A continuación se van a enumerar una serie de reglas basadas en la intuición y el sentido común:

1. Se puede convertir un dato representado en un determinado tipo de dato de origen a otro tipo de dato de destino cuando el valor representado está dentro del rango de representación del tipo de destino.
2. Para realizar una operación sobre datos de distinto tipo se convierten al tipo de rango de representación superior entre ambos.
3. Todos los datos enteros son convertibles entre sí salvo por restricciones del rango de representación del tipo de dato de destino.

---

<sup>3</sup>Se recuerda que una incógnita o un símbolo matemático normalmente no se consideran números porque no tienen un valor concreto y conocido. Solo puede ser un dato una información que tiene un valor concreto.

4. Todos los datos en coma flotante son convertibles entre sí salvo por restricciones del rango de representación y precisión del tipo de dato de destino.
5. Todos los datos enteros se pueden convertir a un tipo en coma flotante.

Cuando no se respetan estas reglas se pueden producir los siguientes errores o problemas:

1. Problemas en la ejecución del programa. Normalmente ocurre porque el valor convertido no está dentro del rango del tipo de dato de destino y en la conversión se cambia drásticamente su valor. Puede que cambie de signo o que tome un valor muy alejado del original, es difícil de prever. Estos errores son muy peligrosos y detectarlos suele ser costoso una vez que se han cometido. La mejor forma de evitarlos es ser conscientes en todo momento de las conversiones que se están realizando y de si son correctas.
2. Errores o avisos al compilar. En algunos casos el compilador es capaz de detectar conversiones potencialmente peligrosas y avisa con el mensaje correspondiente. Es muy importante eliminar, en la medida de lo posible, todos los avisos por conversiones de tipo. O, en caso contrario, estar seguros de que los avisos no tienen importancia.

Mención especial merece la conversión desde números enteros a datos alfanuméricos y viceversa. Dada su condición de tipos enteros, todos los datos alfanuméricos se pueden operar como números enteros y convertir a otros números enteros. No obstante, es necesario ser cuidado con estas operaciones y conversiones, asegurando que son correctas y se corresponden con lo que queremos realizar. Una precaución adicional cuando se utilizan los tipos alfanuméricos como enteros es marcar si son con o sin signo. De esta manera se puede prever más fácilmente las conversiones posibles y el rango de representación.

Como se ha dicho, siempre se permite la conversión de un tipo entero a uno en coma flotante, pero normalmente se pasa por alto que los tipos en coma flotante son aproximados. Actualmente, con el aumento del tamaño de los registros de los microprocesadores, los enteros tienen rangos cada vez mayores y puede haber algunos casos donde exista una pérdida importante de precisión en la cantidad representada al hacer la conversión. Un número entero en C2 en 64 bits puede tener hasta 19 cifras. Éste es un fenómeno relativamente nuevo porque hasta muy recientemente los rangos de representación de números enteros estaban dentro del número de cifras significativas de todos los tipos en coma flotante. A pesar de todo la conversión siempre será posible.

## 4.6. Funciones de entrada y salida con formato para datos simples

### 4.6.1. Aspectos Generales

La librería estándar de C proporciona funciones para realizar operaciones de entrada y salida de datos respecto de los programas. Estas funciones se encuentran declaradas en el archivo de cabeceras `<stdio.h>`. Entre estas funciones se encuentran las funciones de entrada - salida con formato. Se refieren a operaciones de entrada - salida en donde la información se encuentra en forma de texto, pero se pueden extraer o mostrar (entrada o salida) datos de todo tipo. La conversión necesaria de texto al tipo de dato que corresponda o viceversa se realiza dentro de las propias funciones, sin que los programadores se tengan que ocupar de los detalles de esta transformación. Las funciones de entrada y salida de datos con formato necesitan la siguiente información:

1. El formato del texto que se va a leer o escribir. Es decir, una cadena alfanumérica donde se mostrará como es o como queda el texto.
2. Respecto de los datos que hay que convertir: el tipo que se necesita y su lugar en el texto.

Toda esta información se escribe en el programa mediante los argumentos (los datos entre paréntesis y separados por comas) de las llamadas a las funciones `printf` y `scanf`. El primero de los argumentos de ambas funciones se denomina formato, es de tipo cadena alfanumérico (un texto) y normalmente se escribe como una cadena literal constante (con comillas dobles). Dentro de este argumento encontramos:

- Letras y caracteres normales que se utilizan para saber la forma del texto que se va a procesar.
- Especificadores de formato constituidos por el carácter tanto por ciento (%) seguido por otros caracteres que especifican la manera en que se tiene que realizar la conversión de los tipos de datos correspondientes.

Para usar correctamente las funciones de entrada y salida es necesario conocer cómo se escriben estos especificadores de formato.

#### 4.6.2. Especificadores de formato simple

Los especificadores de formato más simples y comunes a las funciones `printf` y `scanf` son los siguientes:

Caracteres	Tipo al que corresponden
<code>%d</code>	<code>int</code>
<code>%f</code>	<code>float</code>
<code>%c</code>	<code>char</code>

El resto de especificadores son más complejos o son diferentes para `printf` y `scanf`.

#### 4.6.3. Uso simplificado de la función `printf`

La función `printf` sirve para mostrar por pantalla (en general para obtener datos o resultados del programa). En su uso más sencillo sirve para mostrar una cadena de texto, por ejemplo:

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

muestra en la pantalla el texto entre las comillas dobles seguido de un salto de línea (`\n`). Este uso es bastante limitado porque la información que se muestra es siempre la misma: la constante literal que hemos escrito entre los paréntesis. Si queremos mostrar el valor de los datos (variables) de nuestro programa debemos utilizar los especificadores de formato en la cadena y argumentos extra con las variables o expresiones cuyo valor queremos mostrar. En este otro ejemplo:

```
printf("La suma de tres y doce es %d.\n", 3+12);
```

se puede observar el uso de uno de los especificadores de formato indicados más arriba. Los caracteres del especificador de formato tipo entero (`%d`) se colocan en el lugar apropiado del texto que se quiere mostrar, `%d` no se mostrarán en pantalla, en su lugar, se mostrará el valor del segundo de los argumentos de la función `printf`, en este caso un 15. Cualquier otro carácter se mostrará tal cual aparece en el texto, exactamente igual que en el uso anterior. Así el mensaje por pantalla será:

La suma de tres y doce es 15.

El programa ha calculado la expresión numérica `3+12`, ha obtenido el número (entero) 15, lo ha convertido en el texto “15” y lo ha colocado en el texto de salida en el lugar indicado por el especificador de formato. Es posible mostrar más de un valor en un solo `printf`. Para ello es necesario colocar más especificadores de formato en el texto y más argumentos. En este ejemplo:

```
printf("El monomio es %f*x^%d\n", 3.98, 3);
```

podemos ver dos especificadores, uno para datos de tipo `float` o `double` (`%f`) y otro para datos de tipo entero (`%i`). Como se puede observar no es necesario que los especificadores se encuentren separados del resto de caracteres por espacios blancos. Por otro lado, el orden en que aparecen los especificadores de formato en la cadena de texto se corresponde con el orden de los argumentos que se deben escribir a continuación. En este ejemplo, el primer especificador (`%f`) se corresponde con el argumento 3.98 y el segundo (`%i`) con 3.

#### Particularidades del uso de especificadores con `printf`

En el caso de usar `printf` encontramos los siguientes comportamientos de los especificadores de formato especiales para esta función:

- El especificador `%f` se puede utilizar para `float` o `double` indistintamente sin ningún problema. Pero para `double` **no mostrará todos los decimales** que el número tiene en memoria. Para esto será necesario usar las opciones del especificador de formato mediante dos números, el primero indica cuánto espacio va a ocupar el número y el segundo es el número de decimales, en el siguiente ejemplo se muestra el número en un espacio de 16 caracteres<sup>4</sup> y con 14 decimales:

```
printf("El número es %16.14f\n", 3.98461315351);
```

<sup>4</sup>Se puede suprimir el ancho y dejar solo el número de decimales, para ello se debe poner directamente el punto y los decimales, así: “`.14f`”.

- Se puede utilizar un dato y un especificador que no se corresponda con el tipo del dato si el dato se puede convertir al tipo del especificador. Este uso *no se recomienda* y puede producir avisos del compilador.
  - Un dato tipo **char** con un especificador **%d** produce el número en la tabla ASCII del carácter.
  - Un dato de tipo número entero con un especificador **%c** produce el carácter correspondiente al número según la tabla ASCII.

#### 4.6.4. Uso simplificado de la función **scanf**

Las llamadas a **scanf** normalmente ponen en el argumento de formato, al menos, un especificador de formato y tienen un argumento extra que consiste simplificadamente en una variable. El programa recibe la información (por ejemplo, un número que teclea el usuario) a través del valor de esa variable. Así pues, cuando **scanf** termina y se ejecutan las sentencias posteriores, el valor de la variable es la información recibida (escrita en el teclado).

Los datos que se escriben por teclado no se envían inmediatamente a la función **scanf**, antes es necesario “confirmar” la entrada. Esto se hace mediante la pulsación de la tecla “*intro*” o “*enter*”. Por supuesto, la pulsación de esta tecla también introduce el carácter salto de línea en la entrada de datos. Para usar la función **scanf** sin dominar el uso de paso de parámetro de tipo puntero es necesario aprender una regla muy sencilla. Para usar **scanf**:

Con datos de tipo simple (**int**, **char**, **double**, **float**) se debe utilizar siempre como argumento una variable del tipo correspondiente precedida por el signo *umpersand* (**&**).

En el siguiente ejemplo se piden por teclado dos valores: un entero y uno en coma flotante.

```
int num; float valor;
scanf("%d%f", &num, &valor);
```

Como antes, el (**%d**) se corresponde por su orden de aparición con **&num**, el primer argumento que sigue al formato. Los caracteres que el usuario introduzca en primer lugar se convertirán (si es posible) a un número entero y ese valor se guardará en la variable **num**. Análogamente los caracteres que se tecleen después se convertirán a un número en coma flotante (**%f**) y se guardarán en **valor**. Al leer la información el programa se saltará todos los blancos que el usuario escriba. Pero, si introduce algún carácter que no pueda formar parte de los números se producirá un error y **scanf** terminará sin completar la lectura de todos los datos requeridos. Si se presenta alguna dificultad con la lectura de datos en primer lugar muestre por pantalla los valores de las variables leídas para comprobar que son correctos antes de realizar otros cálculos.

#### Particularidades del uso de especificadores con **scanf**

El uso de **scanf** debe ser más cuidadoso que el de **printf**, en concreto es necesario saber que:

- Los argumentos de **scanf** son siempre variables (con o sin **&**, para tipos simples siempre con **&**).
- El tipo de los especificadores de formato y de las variables pasadas como argumentos debe corresponder *exactamente*. Si no se corresponden se pueden producir **errores graves** de ejecución.
- Para pedir datos de tipo **double** hay que utilizar el especificador de formato: **%lf**.

#### Caracteres distintos de especificadores de formato

En el uso de **scanf** más básico no se suele usar otros caracteres en el texto de formato distintos de blancos y especificadores de formato. No obstante, es conveniente saber cómo interpreta **scanf** estos caracteres:

- A efectos de **scanf**, vamos a decir que un carácter es *blanco* si es uno de los siguientes caracteres: espacio (de la barra espaciadora), salto de línea (de la tecla “*intro*” o con la flecha hacia abajo y a la izquierda) o tabulador (tecla con dos flechas). Este término, “blanco”, es la traducción del inglés “*blank*” y no se debe confundir con el espacio de la barra espaciadora (*space*).
- Un blanco (un espacio, tabulador o salto de línea) en el texto de formato le indica a **scanf** que debe saltarse todos los blancos hasta que encuentre un carácter distinto de blanco. El efecto es el mismo si se escriben varios blanco en la cadena de formato, por esto, lo normal es escribir un solo blanco y que sea un espacio. Este espacio equivale en la entrada a ningún blanco, uno o varios.

- Cualquier otro carácter que se encuentre en el texto de formato indica a `scanf` que ese carácter se debe encontrar en la entrada (y por tanto se debe escribir mediante el teclado). Si el carácter no se encuentra en la entrada `scanf` no lee el resto de datos.

Por ejemplo, si el texto de formato es: “`%f , %i`”, `scanf` interpreta que se debe escribir: un número en coma flotante, seguido de cualquier número de blancos (incluido ningún blanco), seguidos de un carácter coma ( , ), seguido de cualquier número de blancos y seguido de un número entero. Si el usuario no escribe la coma ( , ), `scanf` nunca leerá el número entero.

## 4.7. Ejemplos

En el ejemplo 4.2 se pueden observar distintos ejemplos de declaraciones de variables de tipos básicos enteros, alfanuméricos y en coma flotante. Se han inicializado todas las variables tal y como se ha recomendado. En las inicializaciones se han usado distintos literales para ilustrar distintas maneras de dar valores a las variables. En este ejemplo también se puede observar que para mostrar por pantalla enteros *largos* es necesario poner el especificador de formato `%ld`, mientras que para todos los números en coma flotante sirve el especificador de formato `%f`.

Ejemplo 4.2: Declaraciones de variables de tipo básico

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a = 4; unsigned long b = 1, c = 33400;
7     short d = -3; unsigned e = 023; /*19*/
8
9     char c1 = 'a'; unsigned char c2 = 0x61; /*97*/
10
11    float x = 1; double y = 12.4e-4, z = 3.1416;
12
13    printf("Numeros enteros: %d %ld %ld \n",a,b,c);
14    printf("Numeros enteros: %d %d \n",d,e);
15    printf("Letras: %c %c %d \n",c1,c2,c2);
16    printf("Coma flotante %f %f %f \n", x, y, z);
17
18    return 0;
19 }
```

---

Salida por pantalla en la ejecución:

```

Numeros enteros: 4 1 33400
Numeros enteros: -3 19
Letras: a a 97
Coma flotante 1.000000 0.001240 3.141600
```

En el ejemplo 4.3 se pueden observar distintas declaraciones de constantes con identificador. Nótese que en el caso de las constantes la inicialización del valor es obligatorio. Se han realizado menos declaraciones que en el caso de las constantes puesto que la única diferencia es escribir el modificador `const` en la declaración.

Ejemplo 4.3: Declaraciones de constantes de tipo básico

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     const int A = -1, B = 2;
7     const char C = 'A';
8     const float X = 12.334;
9
10    printf("Enteros %d %d \n", A, B);
11    printf("Letra %c \n", C);
```

```
12     printf("Coma flotante %f \n", x);  
13  
14     return 0;  
15 }
```

---

Salida por pantalla en la ejecución:

```
Enteros -1 2  
Letra A  
Coma flotante 12.334000
```

## Ejercicios de programación

1. Escribir un programa que muestre por pantalla el valor decimal del número E5C (escrito en hexadecimal). Añadir otra sentencia para mostrar el valor del número 3564 (escrito en octal).
2. Escriba un programa que pida un dato y lo muestre por pantalla. Cambie el tipo de dato entre los datos simples que hemos introducido: `int`, `float` y `char`. Compruebe que ocurre si introduce un dato que no se corresponde con lo que pide el programa, por ejemplo, se pide un número y se escribe una letra.
3. Compruebe si los números en coma flotante, al mostrarlos por pantalla, se redondean o se truncan.
4. Compruebe el tamaño de los tipos de datos simples en su ordenador. ¿Coinciden con alguno de los tamaños que aparecen en las tablas o ejemplos?
5. Intente guardar enteros con signo en un entero sin signo. ¿Qué valor se guarda? ¿El compilador da algún aviso?
6. Busque una solución ingeniosa para mostrar por pantalla el valor máximo de una variable de tipo `unsigned short`.
7. Busque cuál es el tipo de dato adecuado para la siguiente información: cantidad monetaria, temperatura, día del mes, día de la semana.



# Capítulo 5

## Expresiones y Asignaciones

Objetivos:

1. Definir el concepto de expresión (Conocimiento)
2. Describir los operadores (de asignación, aritméticos, de relación, lógicos y de bit) y los tipos de dato sobre los que actúan (Conocimiento)
3. Evaluar expresiones que empleen datos de tipos básicos, operadores y paréntesis (Comprendión)
4. Construir expresiones que empleen combinaciones de datos simples, operadores y paréntesis (Aplicación)

En este capítulo se presentan los siguientes elementos de la programación: las expresiones y los operadores. Se define el concepto de expresión y se continúa con el estudio de los distintos tipos de operadores: de asignación, aritméticos, de relación, lógicos y de bit. En el apartado final se analizan las reglas de prioridad de los operadores que se siguen en la evaluación de expresiones de todo tipo.

### 5.1. Expresiones

Las *expresiones* son una parte fundamental de la programación ya que sirven para realizar una o varias operaciones sobre un dato o un conjunto de datos, obteniéndose otro dato como resultado. Los operadores definen algunas de las operaciones que pueden realizarse dentro de una expresión.

Una expresión es una secuencia de *operandos* y *operadores*.

Los datos u operandos pueden ser constantes, variables y, en general, cualquier elemento de programación que produzca un valor. Además, dentro de una expresión pueden encontrarse subexpresiones encerradas entre paréntesis.

Cuando se ejecuta una sentencia de código que contiene una expresión, ésta se evalúa. Al evaluarse la expresión toma un valor que depende del valor asignado previamente a las variables, las constantes y los operadores y funciones utilizadas y la secuencia de la ejecución de las operaciones correspondientes. Este valor resultante de la evaluación de la expresión será de un determinado tipo de dato. Por ejemplo, de un tipo numérico entero o de un tipo real. Como en el capítulo anterior ya se trató sobre los datos simples que constituyen los posibles operandos que pueden emplearse en C, este capítulo se centrará en los operadores.

Obsérvese la diferencia entre una expresión en un contexto matemático y en el contexto de un programa informático. En una expresión matemática, los datos no necesariamente tienen un valor, ni necesariamente se calcula el resultado en cuanto se escribe la expresión. Sin embargo, en informática, al escribir el código fuente del programa se ha de pensar que el resultado de la expresión que se indica va a calcularse cuando la ejecución del programa alcance la expresión y esto significa que:

1. Todos los operandos de la expresión tienen un valor concreto.
2. Se evalúa el resultado de la expresión de forma inmediata, naturalmente, dando lugar a un valor concreto.

Por otro lado en la escritura de expresiones matemáticas es corriente utilizar representaciones abreviadas y signos que tienen un significado ligeramente diferente al informático (en programación). Son especialmente significativos las siguientes diferencias:

1. En lenguaje matemático generalmente el signo de la operación aritmética *multiplicar o producto* se omite, no así en informática donde, además, para evitar confusiones y que se vea claramente se usa el carácter asterisco (\*).
2. El signo igual, =, se usa en matemáticas para representar ecuaciones, es decir, una expresión que cumple que sus dos términos son iguales en cualquier caso. Este significado no tiene equivalente en informática. En programación existe un operador que comprueba si la expresión se cumple o no, y que se llama de *igualdad* (se escribe con una secuencia de 2 caracteres igual: ==). Este operador no coincide con el *igual* matemático, entre otras cosas porque produce un valor verdadero o falso (este falso en matemáticas no se contempla o se dice que es una incompatibilidad).
3. En programación existe una operación muy importante que consiste en guardar el valor de un dato concreto en una variable. Esto en matemáticas no se contempla de forma especial o no se le da especial importancia. Esta operación se denomina *asignación* y en C también se representa por un operador que se corresponde con el carácter igual (=).

## 5.2. Operadores

En el código fuente de un programa un operador es un carácter o una secuencia de caracteres, normalmente símbolos o signos de puntuación, y representa la evaluación de la operación que corresponda. Así pues, los operadores definen las operaciones que van a realizarse con los datos u operandos.

En C existen distintos tipos de operadores. Pueden clasificarse por el número de operandos, en *unarios* o *unitarios* (un operando) y *binarios* (dos operandos). A su vez, los operadores unarios se puede aplicar por la derecha o por la izquierda (*post-fix* o *pre-fix*). También pueden clasificarse por el tipo de operandos y de su resultado, en operadores aritméticos, de relación, lógicos o booleanos....

Algunos caracteres se corresponden con distintos operadores, lo que significa que la operación que representan depende del número o tipos de operandos que aparecen en el contexto. A esta propiedad se denomina *sobrecarga* del operador. Por ejemplo, el carácter asterisco, \*, puede indicar la operación aritmética producto, el operador indirección o un modificador de tipo de dato en una declaración.

## 5.3. Operador asignación

Possiblemente una de las operaciones más importante en cualquier lenguaje de programación es la *asignación*. Entender correctamente el concepto de asignación y aplicarlo de manera adecuada es una de las claves de la programación de aplicaciones informáticas.

La manera de modificar el valor de un dato variable en un programa es el uso de una operación de asignación. Las asignaciones son tan importantes que muchos lenguajes de programación les dan un tratamiento especial, separado del resto de operaciones, no es así en C. A pesar de todo, se recomienda ser consciente de la importancia de esta operación en todo momento y reflejar en el estilo de programación empleada la importancia de las asignaciones, por ejemplo, utilizando una única asignación por expresión.

Para la operación asignación en lenguaje C tenemos que distinguir entre su efecto, es decir, su consecuencia, lo que se realiza y el valor resultado de la operación. **El efecto de una operación de asignación es guardar en la variable un nuevo dato**, es decir, sustituir el valor anterior anterior de la variable por el nuevo. Se insiste en que una variable sólo puede tener un único valor y siempre toma un valor. Éste es el efecto, pero no el resultado de la operación. En C el resultado de una operación de asignación es el nuevo valor que se guarda. En un estilo de programación en C sencillo no se debe usar el resultado de una operación de asignación, simplemente se descarta, en adelante se usará la asignación de esta manera y no se reflejará ni siquiera en la descripción de la sintaxis esta característica de las asignaciones.

### Sintaxis

La sintaxis simplificada de una única asignación en una expresión es como sigue:

```
variable = expresion
```

Cuando se quiere utilizar esta expresión como sentencia (el caso más corriente y recomendable) se añade un punto y coma al final:

```
variable = expresion ;
```

Cuadro 5.1: Operadores aritméticos

Operador	Descripción	Ejemplo	Resultado
-	Cambio de signo	-4	-4
+	Suma	2.5 + 7.1	9.6
-	Resta	235.6 - 103.5	132.1
*	Producto	1.2 * 1.1	1.32
/	División real	0.050 / 0.2	0.25
/	Cociente de la división entera	20 / 7	2
%	Resto de la división entera	20% 7	6

Donde `=` es el operador de asignación, `expresion` es una expresión válida (*right value*) que se evalúa a un tipo de dato *convertible* en el tipo de dato de la `variable` (*left value*). Más adelante se explicarán otras alternativas que se pueden utilizar en el lado izquierdo de la asignación.

Los operaciones de asignación se pueden utilizar con todo tipo de datos. Naturalmente se exige que el tipo del dato que se evalúa en la derecha sea convertible en el tipo de dato de la variable de la izquierda.

## 5.4. Operadores aritméticos

Los operadores *aritméticos* operan sobre valores de tipo entero o coma flotante. Los operadores aritméticos se resumen en la Tabla 5.1. En el caso del operador unario de cambio de signo, el resultado es del mismo tipo que el del operando. En el caso de los tres primeros operadores binarios (suma, resta y producto) si ambos operandos son enteros el resultado es entero. Si alguno de los operandos está representado en coma flotante el resultado es en coma flotante.

Debe prestarse especial atención al operador división. Como se puede comprobar la operación que se realiza, división entera o real, depende del contexto. Es decir, si los operandos son enteros es el cociente de la división entera y si alguno de ellos es real es la división real.

### 5.4.1. Operadores aritméticos combinados

La sintaxis simplificada de un único operador aritmético combinado con asignación en una sentencia es como sigue:

```
variable op_aritmetico_combinado expresion ;
```

Donde `variable` tiene el mismo sentido que en las asignaciones simples y `expresion` es un valor que se puede operar con la variable mediante la operación aritmética que corresponda. En la tabla 5.2 se recogen los operadores aritméticos combinados con *asignación* más comunes. Todos las operaciones aritméticas combinadas con asignación son *binarias*. Tal y como se ha mostrado en la sintaxis, tienen dos operandos, uno a la izquierda y otro a la derecha. Para las operaciones aritméticas es necesario que los operandos sean numéricos enteros o reales y el tipo específico del resultado numérico dependerá del tipo de éstos.

Cuadro 5.2: Operadores aritméticos combinados con asignación

Operador	Descripción	Ejemplo	Resultado
<code>+=</code>	Suma combinada	<code>a+=b</code>	a guarda como dato el resultado de <code>a+b</code>
<code>-=</code>	Resta combinada	<code>a-=b</code>	a guarda como dato el resultado de <code>a-b</code>
<code>*=</code>	Producto combinado	<code>a*=b</code>	a guarda como dato el resultado de <code>a*b</code>
<code>/=</code>	División combinada	<code>a/=b</code>	a guarda como dato el resultado de <code>a/b</code>
<code>%=</code>	Resto combinado	<code>a%=b</code>	a guarda como dato el resultado de <code>a%b</code>

### 5.4.2. Operadores aritméticos incrementales

Los operadores *aritméticos incrementales* son operadores unarios (un único operando). El operando puede ser de cualquier tipo entero o en coma flotante. El resultado es del mismo tipo que el operando. Estos operadores pueden emplearse de dos formas distintas dependiendo de su posición con respecto al operando, antes o después del mismo, que se denominan *prefix* y *postfix*. Se debe distinguir entre el resultado de la operación y el efecto de la misma. Todos los

operadores incrementales tienen el mismo efecto (incrementar o decrementar una variable), pero los operadores *prefix* tienen como resultado el nuevo valor y los *postfix* el valor antiguo.

Los operadores se muestran en la tabla 5.3.

Cuadro 5.3: Operadores aritméticos incrementales

Operador	Descripción	Ejemplo	Resultado
<code>++</code>	incremento	<code>a=4; ++a;</code>	a vale 5
<code>i++</code>	<i>postfix</i> , se incrementa y se devuelve el valor antiguo	<code>a=5; b=a++;</code>	a vale 6 y b vale 5
<code>++i</code>	<i>prefix</i> , se incrementa y se devuelve el valor nuevo	<code>a=5; b=++a;</code>	a vale 6 y b vale 6
<code>--</code>	decremento	<code>a=4; --a;</code>	a vale 3

Estos operadores suelen sustituir al operador asignación en incrementos o decrementos en una unidad de las variables. En este sentido se aplican las mismas restricciones que en las asignaciones, es decir, el operando debe ser una *variable*. El resultado de estos operadores es un *valor*. Como los operadores incrementales equivalen a una asignación que no se muestra de manera explícita, se recomienda no mezclarlos con otras operaciones ni asignaciones para facilitar la lectura del código.

Un operador incremental que se utiliza en solitario es casi indiferente que sea *prefix* o *postfix*. Aunque el uso de los operadores *postfix* está muy extendido, existen distintos motivos (especialmente de eficiencia porque no se tiene que guardar un resultado extra intermedio) para que los operadores *prefix* se escojan de forma preferente.

## 5.5. Operadores de relación o comparación

Los operadores de relación o comparación realizan comparaciones entre datos convertibles entre sí de tipos básicos y punteros teniendo siempre un resultado de *verdadero* o *falso* (representado mediante un valor de tipo `int`). Si los tipos de los operandos no coinciden se convierten al tipo de mayor rango de representación y en caso de haber números enteros y en coma flotante a coma flotante. Es decir, cuando la operación tiene como resultado *verdadero* el entero es uno, 1, mientras que cuando el resultado es falso el entero es cero, 0.

Los operadores de relación o comparación se muestran en la tabla 5.4.

Cuadro 5.4: Operadores de relación o comparación

Operador	Descripción	Ejemplo	Resultado
<code>==</code>	igual que	<code>7 == 38</code>	0
<code>!=</code>	distinto que	<code>'a' != 'k'</code>	1
<code>&lt;</code>	menor que	<code>'G' &lt; 'B'</code>	0
<code>&gt;</code>	mayor que	<code>'b' &gt; 'a'</code>	1
<code>&lt;=</code>	menor o igual que	<code>7 &lt;= 7.38</code>	1
<code>&gt;=</code>	mayor o igual que	<code>38 &gt;= 7</code>	1

Al usar operadores de relación hay que tener las siguientes precauciones:

1. No hay que confundir el operador de relación de igualdad `==`, con el operador de asignación `=`, que asigna valores a variables.
2. No se recomienda usar comparaciones de igualdad con números en coma flotante dada su cualidad de números aproximados. Es mucho más correcto calcular la distancia entre los números y comprobar si está por debajo de un límite dado.
3. Algunos datos compuestos que se verán más adelante (en concreto, vectores y cadenas alfanuméricas) **no** tienen un operador `==`, pero son convertibles a un valor de tipo puntero que sí tiene este operador y, por tanto, el compilador realiza la conversión y luego utiliza esta operación que no tiene nada que ver con lo que el programador seguramente pueda estar pensando.

## 5.6. Operadores lógicos

Los operadores *lógicos* realizan operaciones con operandos de tipo escalar (aritméticos o punteros). Antes de usarse, los operandos se convierten a un valor booleano, 0 o 1, según sean iguales o distintos de cero respectivamente. El

resultado es un dato de tipo `int`. Los operadores lógicos definidos en C se resumen en la tabla 5.5. Los paréntesis que aparecen en los ejemplos de la tabla no son necesarios, pero contribuyen a hacer la expresión más clara.

Cuadro 5.5: Operadores lógicos

Operador	Descripción	Ejemplo	Resultado
!	Negación	!5	0
	Suma lógica ( <i>or</i> )	8   (5<4)	1
&&	Producto lógico ( <i>and</i> )	9 &&(5<4)	0

Tanto en el caso de las operaciones de relación como en las operaciones lógicas el resultado es un entero, `int`. Sin embargo este entero en realidad codifica un resultado de verdadero o falso que estrictamente hablando no es un número entero sino un booleano. Por esta razón se desaconseja utilizar los enteros que proceden de operadores de relación o lógicos como operandos de operaciones de otro tipo. Por ejemplo, este resultado no se debe usar para sumar o multiplicar.

## 5.7. Operadores de bits

Los operadores *de bit* o en inglés *bitwise* se aplican a operandos de tipo entero. El resultado de los mismos es del tipo con mayor rango de representación, con lo que los operandos se convierten al tipo mayor cuando sea necesario. Las operaciones de bit se realizan con los ceros, 0, y los unos, 1, de las representaciones binarias correspondientes a los operandos. Los operadores de bit definidos en C se resumen en la tabla 5.6.

Cuadro 5.6: Operadores de bit

Operador	Descripción	Ejemplo	Resultado
~	Negación lógica de bits - <i>NOT bitwise</i>	~20	-21
	Suma lógica de bits – <i>OR bitwise</i>	12   10	14
^	Suma lógica exclusiva de bits – <i>XOR bitwise</i>	12 ^ 10	6
&	Producto lógico de bits – <i>AND bitwise</i>	12 & 10	8
<<	Desplaza a la izquierda los bits del 1º operando tantas veces como indica el 2º operando	7 << 2	28
>>	Desplaza a la derecha los bits del 1º operando tantas veces como indica el 2º operando	7 >> 2	1

## 5.8. Evaluación en cortocircuito en expresiones lógicas

Cuando se evalúa una expresión lógica compuesta, la evaluación se lleva a cabo de izquierda a derecha, y tan pronto como se asegura el resultado final la evaluación se deja de evaluar términos. Esto significa que:

1. Para el operador `||` cuando uno de los términos es verdadero, la evaluación se detiene ya que los términos adicionales no pueden modificar el resultado que será siempre verdadero.
2. Para el operador `&&`, si alguno de los términos es falso, la evaluación también se detiene porque el resultado final será siempre falso independientemente de los demás términos.

En el caso de términos anidados mediante paréntesis en distintos niveles, las reglas anteriores se aplican a cada nivel de anidación.

## 5.9. Operadores especiales

En C existen una serie de operadores que son característicos de este lenguaje y que en otros lenguajes se consideran como elementos sintácticos con entidad propia y distinta de una operación, otros sencillamente no existen. Algunos de estos operadores son relativamente crípticos, su uso no se recomienda, pero es conveniente conocer que existen. Véase operadores especiales para conocer la lista completa de operadores.

Aquí se va a considerar dos operadores por su utilidad en determinados problemas y porque aparecen con mayor frecuencia.

**Operador coma**, el operador coma tiene dos operandos de cualquier tipo, generalmente expresiones. Se evalúa la primera expresión u operando y su resultado se descarta, a continuación se evalúa la segunda expresión u operando y se utiliza como resultado. No se debe confundir con la coma de separación en otros contextos.

**Operador conversión**, en inglés *cast*, este operador está formado por una pareja de paréntesis con un tipo de dato entre ellos y se aplica por la izquierda.

## 5.10. Subexpresiones

Las expresiones en el código fuente se deben escribir en forma de secuencia lineal de operandos y operaciones. Esta limitación tiene como consecuencia que una expresión se pueda leer de manera ambigua respecto del orden en que se tienen que evaluar las distintas operaciones que aparecen en la expresión. Naturalmente, el compilador tiene unas reglas estrictas y conocidas que resuelven esta ambigüedad y establecen el orden de evaluación de las expresiones. No obstante, puede darse el caso de que se necesite un determinado orden de evaluación que no coincida con el establecido por el compilador.

En cualquiera de estos casos, la mejor manera de resolver estos problemas es mediante el uso de *subexpresiones*. Una subexpresión es una expresión puesta entre paréntesis cuyo resultado se evalúa antes de seguir con la evaluación de resto de la expresión. Es posible tener subexpresiones anidadas formando distintos niveles (siempre con paréntesis). En este caso se resuelven siempre los paréntesis de dentro a fuera: primero los más internos y luego los externos. Lo cual es razonable, se necesita el valor de las expresiones más internas para resolver las expresiones más externas.

## 5.11. Prioridad de operadores

Aunque en el estándar de C no se define un nivel de prioridad explícito, la manera en que se definen sintácticamente las expresiones produce un nivel de prioridad. Los niveles de prioridad son un concepto comúnmente manejado en todos los lenguajes de programación y en consecuencia se va a emplear para establecer el orden de las operaciones presentes en una secuencia de operadores y operandos.

Dada una expresión o subexpresión donde aparecen distintos operandos y operaciones, las primeras operaciones que se realizan son las de prioridad más alta, a continuación sus resultados se utilizan para realizar las operaciones de nivel más bajo. A igual de prioridad las primeras operaciones que se realizan son las situadas más a la izquierda.

Los niveles de prioridad son arbitrarios, pueden cambiar (y cambian) de un lenguaje de programación a otro. Dado que el orden de prioridad no es algo deducible sino que es necesario conocer de memoria y tener muy presente en la construcción de las expresión se recomienda encarecidamente que se usen subexpresiones, es decir, paréntesis, para evitar cualquier tipo de duda sobre el orden de evaluación de las operaciones en una expresión.

En la tabla 5.7 se muestra el orden de prioridad de mayor a menor de los operadores descritos anteriormente. Los niveles de prioridad indicados en esta tabla pueden no ser precisos<sup>1</sup>. En todo caso se recomienda el uso de paréntesis.

Los operadores unarios, las asignaciones y el operador condicional se agrupan de derecha a izquierda. Esto significa que cuando se encuentran dos operadores de estas clases se realiza en primer lugar la operación que se encuentra escrita más a la derecha. Para todos los demás, la agrupación es de izquierda a derecha, es decir, se realizan en primer lugar las operaciones escritas más a la izquierda<sup>2</sup>.

## 5.12. Precauciones en la escritura de expresiones

Escribir expresiones correctas en un lenguaje de programación no siempre es evidente. A ello contribuye su naturaleza de secuencia lineal y al hecho de que no se produce ningún tipo de elipsis u omisión de operaciones u operandos. A estos factores se añade en C la sobrecarga de determinados operandos y su aplicación por izquierda o derecha que no siempre resulta la opción más clara.

En este sentido se recomienda tener en cuenta las siguientes notas:

1. En C, no hay operandos implícitos, es decir, si se quiere saber si tres valores son iguales se tienen que especificar dos operaciones de comparación y una lógica en este orden, por ejemplo:

<sup>1</sup> El estándar de C no especifica la manera en que se evalúan las expresiones a través de prioridades sino a través de una serie de reglas sintácticas muy complejas.

<sup>2</sup> De nuevo el estándar de C no especifica la manera en que se agrupan los operadores, esta agrupación es una consecuencia de la manera en que describen las reglas sintácticas de las expresiones.

Cuadro 5.7: Nivel de prioridad de operadores

Operadores	Significado
( ) [ ] -> . ++ --	Operador llamada a función, indexación de matrices, operadores flecha y punto de estructuras. Operadores incrementales <i>postfix</i> . Todos ellos operadores por la derecha. Es decir, el operando se encuentra a la izquierda y el operador a la derecha
! ~ ++ -- - + * & sizeof ( <i>tipo</i> )	Operadores negación, incrementales <i>prefix</i> , menos unario, más unario, indirección, dirección de memoria de, de tamaño del tipo o variable y conversión o <i>cast</i> . Todos operadores por la izquierda. Es decir, el operando se encuentra a la izquierda y el operador a la derecha.
* / %	Operadores <i>multiplicativos</i> , multiplicación, división real o cociente de la división entera y resto de la división entera
+ -	Operadores <i>aditivos</i> , suma y resta
<< >>	Operadores de desplazamiento de bits
< <= > >=	Operadores de comparación
== !=	Operadores de igualdad y desigualdad
&	Operador de bit <i>and bitwise</i>
^	Operador de bit <i>xor bitwise</i>
	Operador de bit <i>or bitwise</i>
&&	Operador <i>and lógico</i>
	Operador <i>or lógico</i>
?:	Operador condicional
= += -= *= /=	Operadores de asignación y combinados

- a)  $(a == b) \&\& (a == c)$
- b) el código  $a == b == c$ , es un **error** puesto que el resultado de la primera igualdad (un 1 ó 0) es lo que se opera con el segundo operador de igualdad.
2. En C (y en el resto de lenguajes no algebraicos) no hay *castillos* de fracciones ni se omite la multiplicación, lo que significa que las expresiones de esta forma han de tener un gran número de subexpresiones para lograr el mismo resultado y, por supuesto, se tienen que escribir todos los productos.
3. Se debe distinguir claramente la asignación y la comparación de igualdad (uno y dos signos igual). Téngase en cuenta que la comparación se puede sustituir sin error aparente por la asignación y su efecto y resultado es totalmente diferente.

## 5.13. Ejemplos de expresiones

En el ejemplo 5.1 se muestran distintas expresiones con aritmética entera y en coma flotante. Así mismo, se muestra el funcionamiento de asignaciones y de los operadores combinados. Es conveniente prestar especial atención a las expresiones donde cambian los valores de las variables para entender el funcionamiento de programa con precisión. Así mismo, es interesante la manera en que hay que expresar la fórmula matemática:

$$\frac{\frac{a-7}{5,0} + x}{y}$$

Obsérvese especialmente el uso de los paréntesis para determinar el orden de operaciones.

Ejemplo 5.1: Operadores aritméticos enteros y en coma flotante

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     short a = 7, b = 3, c;
7     double x = 3.3, y = 5.5, z;

```

```

8      c = a / b;
9      a %= b;
10     ++b;
11     printf("a = %d, b = %d, c = %d \n", a, b, c);
12
13     z = (( a - 7 ) / 5.0 + x ) / y;
14     printf("z = %f \n", z);
15
16     return 0;
17 }

```

---

Salida por pantalla en la ejecución:

```
a = 1, b = 4, c = 2
z = 0.381818
```

En el ejemplo 5.2 se muestran distintas expresiones con operadores de relación o comparación y operaciones lógicas. Debe considerarse especialmente la cualidad de números booleanos (0, 1) codificados como enteros (`int`) de los resultados y operandos de algunos de estos operadores. Por otro lado, estos ejemplos ilustran expresiones comunes en los programas informáticos: en la segunda expresión aparece una comparación entre números en coma flotante con una precisión, en la tercera una expresión para determinar si un número está dentro de un intervalo de valores y en la cuarta se puede ver una aplicación de la evaluación en cortocircuito donde se protege una operación potencialmente peligrosa (una división por cero).

#### Ejemplo 5.2: Operadores de comparación y lógicos

---

```

1
2 #include <stdio.h>
3 #include <math.h>
4
5 int main()
6 {
7     short a = 7, b = 3;
8     double x = 3.3, y = 3.300001;
9     int bool_1, bool_2, bool_3, bool_4;
10
11    bool_1 = ( a == b ) || ( a > 5 );
12    printf("booleano 1 = %d \n", bool_1);
13
14    bool_2 = fabs( x - y ) < 1e-3;
15    printf("booleano 2 = %d \n", bool_2);
16
17    bool_3 = ( x >= 0.0 ) && ( x <= 3.0 );
18    printf("booleano 3 = %d \n", bool_3);
19
20    bool_4 = ( a - 7 != 0 ) && ( y / ( a - 7 ) < 1.0 );
21    printf("booleano 4 = %d \n", bool_4);
22
23    return 0;
24 }

```

---

Salida por pantalla en la ejecución:

```
booleano 1 = 1
booleano 2 = 1
booleano 3 = 0
booleano 4 = 0
```

En el ejemplo 5.3 se muestran los resultados de distintos operadores de bits. Se recomienda encontrar las representaciones booleanas de los números que intervienen en las mismas para comprobar los resultados obtenidos.

#### Ejemplo 5.3: Operadores de bits o bitwise

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     short a = -359, b = 64, c = 68;
7
8     printf("not a + 1: %d \n", ~a + 1);
9     printf("b >> 2: %d \n", b >> 2);
10    printf("b << 2: %d \n", b << 2);
11
12    printf("b & c: %d \n", b & c);
13    printf("b | c: %d \n", b | c);
14    printf("b ^ c: %d \n", b ^ c);
15
16    return 0;
17 }

```

---

Salida por pantalla en la ejecución:

```

not a + 1: 359
b >> 2: 16
b << 2: 256
b & c: 64
b | c: 68
b ^ c: 4

```

## Ejercicios propuestos del capítulo de Expresiones y operadores

- Construir un programa que, dado un número total de horas, devuelve el número de semanas, días y horas equivalentes. Por ejemplo, dado un total de 1000 horas debe mostrar 5 semanas, 6 días y 16 horas. El número total de horas se debe introducir durante la ejecución del programa por teclado. Para ello debe emplearse `scanf`. Por ejemplo: `scanf("%d", &horas);`
- Construir un programa que incluya una expresión con resultado numérico entero de 1 (verdadero) o 0 (falso) si el valor de una variable entera `n` cumple o no las dos condiciones siguientes: (a) ser divisible entre 4 o divisible entre 5 y (b) ser estrictamente mayor que 10 y estrictamente menor que 100. Por ejemplo, si `n` vale 12, 15 ó 95 la expresión debe tener como resultado 1, mientras que si `n` vale 5, 31 ó 104 la expresión debe tener como resultado 0.

## Soluciones a los ejercicios propuestos del capítulo de Expresiones y operadores

- Programa que devuelve el número de semanas, días y horas equivalentes a un número total de horas

Ejemplo 5.4: Horas equivalentes en semanas, días y horas

---

```

1 #include <stdio.h>
2 int main()
3 {
4     int horas, ss, dd, hh;
5     scanf("%d",&horas);
6     printf("El total de horas es %d\n", horas);
7     ss = horas/(24*7);
8     dd = horas % (24*7)/24;
9     hh = horas % 24;
10    printf("Equivale a %d semanas, ", ss);
11    printf("%d dias ", dd);
12    printf("y %d horas.\n", hh);
13    return 0;
14 }

```

---

Al introducir por teclado en la ejecución:

1000

Salida por pantalla en la ejecución:

```
El total de horas es 1000
Equivale a 5 semanas, 6 dias y 16 horas.
```

## 2. Programa con expresión numérica

Ejemplo 5.5: Expresion numerica compleja

---

```
1 #include <stdio.h>
2 int main()
3 {
4     int n,i;
5     scanf("%d",&n);
6     printf("El entero introducido es %i\n", n);
7     i = (n % 4 == 0 || n % 5 == 0) && (n > 10 && n < 100);
8     printf("La expresion vale %d\n",i);
9     return 0;
10 }
```

---

Al introducir por teclado en la ejecución:

12

Salida por pantalla en la ejecución:

```
El entero introducido es 12
La expresion vale 1
```

## Otros ejercicios de programación

1. Escribir un programa que intercambie el valor de dos variables de tipo numérico.
2. Escribir una expresión que indique si un número pertenece a un intervalo abierto.
3. Escribir una expresión que indique si un número pertenece a un intervalo cerrado.
4. Escribir una expresión que indique si un número es par.
5. Escribir una expresión cuyo resultado sea la última cifra de un número.
6. Escribir una expresión cuyo resultado sea todas las cifras de un número excepto la última.
7. Escribir una expresión cuyo resultado sea las dos últimas cifras de un número.
8. Escribir una fórmula con fracciones y traducirla a una expresión de C.
9. Escribir una expresión que multiplique un número entero por 4 sin emplear operadores aritméticos.
10. Escribir una expresión cuyo resultado sea el bit situado en la posición 4 de un entero unsigned char.
11. Escribir una expresión cuyo resultado sea la letra inglesa siguiente a una dada. Más difícil: la siguiente letra de la 'z' es la 'a'. Aún más, devolver el mismo carácter si no es una letra.
12. Escribir una expresión cuyo resultado sea la letra mayúscula de una letra inglesa dada. Más difícil: solo cambiar las letras mayúsculas.
13. Escribir una expresión cuyo resultado sea la letra (alfanumérico) que corresponde a un dígito (número).

14. Escribir una expresión cuyo resultado sea el dígito como dato numérico de una letra entre '0' y '9'.
15. Escribir una expresión que indique si tres valores enteros son iguales
16. Escribir una expresión que indique si dos valores reales son iguales
17. Escribir una expresión que indique si un año es bisiesto o no
18. Verificar las expresiones anteriores mediante los programas correspondientes



# Capítulo 6

## Sentencias selectivas o condicionales

Objetivos:

1. Describir el funcionamiento de las sentencias selectivas o condicionales: **if-else** y **switch** (Conocimiento)
2. Interpretar y predecir el resultado de una secuencia de sentencias que pueden incluir sentencias selectivas (Comprendión)
3. Codificar una tarea sencilla convenientemente especificada, utilizando la secuencia y combinación adecuada de sentencias selectivas (Aplicación)

### 6.1. Sentencias o instrucciones

Las *sentencias*, también llamadas *instrucciones* o *proposiciones*

1. definen la lógica de un programa y de una función, subprograma o subrutina. En un programa sencillo que conste de varias sentencias simples, la ejecución comenzaría por la primera sentencia de la función **main**, una vez terminada comenzaría la siguiente y así sucesivamente hasta la última sentencia.
2. manipulan los datos descritos anteriormente para producir el resultado deseado por el usuario del programa.

Hay dos tipos de sentencias en C:

1. sentencias *simples*: una expresión que se termina en punto y coma se transforma en una sentencia. Ésta es el tipo de sentencia más habitual que se encuentra en el código fuente de programas y rutinas. Naturalmente, cuando se describan otras sentencias se entenderá que una sentencia simple cualquiera *incluye* el punto y coma final. La sintaxis es:

`expresion;`

2. sentencias *compuestas*: si se agrupan varias sentencias simples entre llaves se forman sentencias o proposiciones *compuestas* o *bloques* como, por ejemplo, las sentencias que forman la rutina principal **main**. En este caso se entiende que la sentencia abarca desde la apertura de la llave hasta el cierre ambas incluidas. Naturalmente, cuando se describan otras sentencias se entenderá que un bloque cualquiera *incluye* las llaves. La sintaxis es:

```
{  
    sentencia_1  
    sentencia_2  
    sentencia_3  
    sentencia_n  
}
```

Obsérvese que detrás de las sentencias no se ha puesto punto y coma. Se sobreentiende que cuando escribimos **sentencia1** ésta ya incluye el carácter de punto y coma (ya que forma parte de la sentencia simple), o ya lleva la apertura y cierre de llaves (ya que forman parte de otra sentencia compuesta).

Como se ha comentado anteriormente cuando se escribe un programa, se introduce la secuencia de sentencias dentro del código del `main` de un archivo fuente. Sin sentencias de *control del flujo*, el intérprete ejecuta las sentencias conforme aparecen en el programa de principio a fin. Las sentencias de control de flujo se emplean en los programas para ejecutar sentencias condicionalmente, repetir un conjunto de sentencias o, en general, cambiar el flujo secuencial de ejecución. Las sentencias *selectivas* o *condicionales* de C (sentencias `if-else` y `switch`) se verán en este capítulo y las sentencias *repetitivas* o *bucles* y otros tipos de sentencias en el siguiente.

## 6.2. La sentencia if

La sentencia `if-else` es una bifurcación con dos ramas. En primer lugar, al ejecutarse la sentencia evalúa una condición o expresión lógica. Después, si la condición o expresión booleana es cierta (`true`) ejecuta la `sentencia_1`, en caso contrario (si es falsa - `false`), ejecuta la `sentencia_2` si ésta existe (ya que la parte `else` es opcional). La sentencia `if` sigue la siguiente sintaxis y el flujograma mostrado en la Figura 6.1.

Sintaxis, una a elegir entre:

```
if (expresión-lógica) sentencia_1
if (expresión-lógica) sentencia_1 else sentencia_2
```

Obsérvese que no se explicitan los caracteres de punto y coma finales de las sentencias 1 y 2 (si éstas fuesen sentencias simples) ni las aperturas y cierres de llave (si fuesen sentencias compuestas). De esta manera la sintaxis indicada describe cualquier combinación posible de sentencia simple o compuesta para las sentencias 1 y 2.

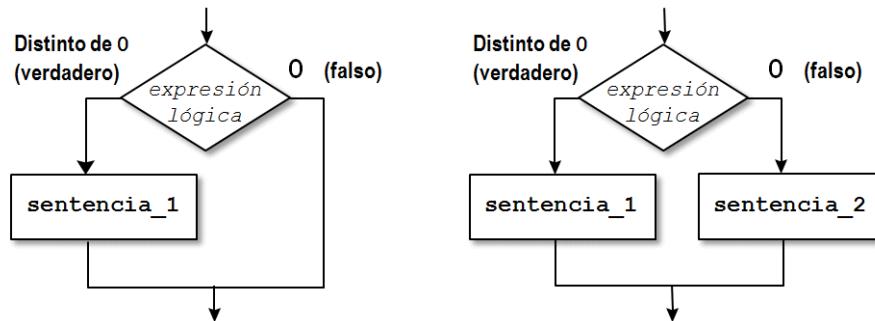


Figura 6.1: Flujograma de la sentencia `if`. Con una rama (a la izquierda) y con dos ramas (a la derecha)

Un ejemplo completo pero muy sencillo que muestra la utilización de este tipo de sentencia se encuentra en el código fuente 6.1:

Ejemplo 6.1: Sentencia if muy sencilla

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a = 27, b = 39;
7     if ( a > b ) printf("a es mayor que b\n"); /*sentencia 1*/
8     else printf("a no es mayor que b\n"); /*sentencia 2*/
9     return 0;
10 }
```

---

Salida por pantalla en la ejecución:

a no es mayor que b

## 6.3. Más ejemplos de uso de if

El código fuente 6.2 del programa `ejemif.c` muestra otro ejemplo de uso de la sentencia `if` con una única rama.

## Ejemplo 6.2: Sentencia if con una rama

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int n = 27;
7     printf("Inicio del programa\n");
8     if (n % 2 == 0)
9         printf("El valor de n, %d, es par\n", n);
10    printf("Fin del programa\n");
11    return 0;
12 }
```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
Fin del programa
```

La sentencia **if** se inicia con la palabra reservada **if** seguido de una expresión entre paréntesis. La expresión se evalúa y si es cierta se ejecuta la sentencia simple que viene a continuación en este caso. Si la expresión es falsa la sentencia simple se salta. Aquí también podría incluirse una sentencia compuesta en lugar de una simple incluyendo aquella entre llaves. La expresión **n % 2 == 0** evalúa si el resto de dividir el valor de la variable **n** entre 2 es igual a 0. Esta expresión es significativamente distinta a **n % 2 = 2** en la que se trata de hacer una asignación. Este es un error muy común en las expresiones que se utilizan en la sentencia **if**, por lo que se recomienda ser especialmente cuidadoso con el operador igualdad.

En el código fuente 6.3 del programa **ejemif2.c** se muestra un ejemplo de uso de la sentencia **if** con dos ramas.

## Ejemplo 6.3: Sentencia if con dos ramas

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int n = 27;
7     printf("Inicio del programa\n");
8     if (n % 2 == 0)
9         printf("El valor de n, %d, es par\n", n);
10    else
11        printf("El valor de n, %d, es impar\n", n);
12    printf("Fin del programa\n");
13    return 0;
14 }
```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
El valor de n, 27, es impar
Fin del programa
```

Este segundo ejemplo de uso de la segunda sentencia **if** es similar al primero pero introduce algo adicional: la palabra reservada **else** seguida de una sentencia. Esta diferencia provoca que si la expresión entre paréntesis es cierta, se ejecuta la primera sentencias, mientras que si es **false** se ejecuta la sentencia que sigue al **else**. De esta forma, una de las dos expresiones siempre se ejecuta y la otra se salta. En el primer ejemplo, la única sentencia o se ejecutaba o se saltaba.

El código fuente 6.4 del programa **ejemif3.c** muestra un tercer ejemplo que emplea varias sentencias **if** seguidas.

## Ejemplo 6.4: Sentencias if una detrás de otra

---

```

1
2 #include <stdio.h>
3
```

```

4 int main()
5 {
6     int n = 27;
7     printf("Inicio del programa\n");
8
9     if (n % 2 == 0)
10        printf("El valor de n, %d, es par\n", n);
11    else
12        printf("El valor de n, %d, es impar\n", n);
13
14    if (n > 15)
15        printf("El valor de n, %d, es mayor de 15\n", n);
16    else
17        printf("El valor de n, %d, no es mayor de 15\n", n);
18
19    printf("Fin del programa\n");
20    return 0;
21 }

```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
El valor de n, 27, es impar
El valor de n, 27, es mayor de 15
Fin del programa

```

## 6.4. Sentencias if anidadas

Las sentencias `if-else` pueden ir anidadas unas dentro de otras en el código fuente del programa. Es decir, las sentencias 1 y 2 pueden a su vez ser nuevas sentencias `if`. Cuando dentro de las sentencias 1 ó 2 se encuentre una rama `else`, son posibles diferentes interpretaciones respecto de a qué sentencia `if` corresponde. El estándar C99 establece que:

*An else is associated with the lexically nearest preceding if that is allowed by the syntax.*

Traducción: Una rama `else` está asociada con la sentencia `if` precedente más cercana permitida por la sintaxis.

Lo que significa que un `else` pertenece al `if` anterior que se encuentre más próximo al `else`. Por ejemplo:

```

if (expresion1) sentencia1
else if (expresion2) sentencia2
else sentencia3

```

Según el estándar, el `else` de la sentencia 3 corresponde con el `if` de la expresión 2. La escritura de secuencias `if-else if`, como la indicada, es frecuente porque sirve para programar algoritmos donde se tiene que ejecutar distintas sentencias de forma excluyente según el valor de distintas expresiones.

En el código 6.5 se muestra un ejemplo completo de dos sentencias `if` anidadadas.

Ejemplo 6.5: Sentencias if anidadas

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int n = 26;
7     printf("Inicio del programa\n");
8
9     if (n % 2 == 0)
10        if (n % 4 == 0)
11            printf("El valor de n, %d, es divisible entre 4\n", n);
12        else

```

```

13         printf("El valor de n, %d, es par pero no divisible entre 4\n", n);
14     else
15         printf("El valor de n, %d, es impar\n", n);
16
17     printf("Fin del programa\n");
18     return 0;
19 }
```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
El valor de n, 26, es par pero no divisible entre 4
Fin del programa
```

Las anidaciones pueden hacerse tanto dentro de la primera rama como de la segunda como muestra otro ejemplo en el código 6.6.

Ejemplo 6.6: Sentencias if anidadas en rama else

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int n = 29;
7     printf("Inicio del programa\n");
8
9     if (n % 2 == 0)
10        if (n % 4 == 0)
11            printf("El valor de n, %d, es divisible entre 4\n", n);
12        else
13            printf("El valor de n, %d, es par pero no divisible entre 4\n", n);
14    else
15        if (n % 10 != 7)
16            printf("El valor de n, %d, es impar pero no acaba en 7\n", n);
17        else
18            printf("El valor de n, %d, acaba en 7\n", n);
19
20     printf("Fin del programa\n");
21     return 0;
22 }
```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
El valor de n, 29, es impar pero no acaba en 7
Fin del programa
```

## 6.5. La sentencia switch

La sentencia **switch** es una bifurcación de ramas múltiples. Dependiendo del valor de una variable o expresión entera permite ejecutar una o varias sentencias de entre muchas. Consiste en una expresión de selección y un conjunto de etiquetas con valores posibles de la expresión, los valores deben ser constantes. La expresión puede ser de un tipo *ordinal* (de tipo entero o **char**), pero no puede ser de un tipo numérico en coma flotante o de un tipo cadena alfanumérica.

Sigue la siguiente sintaxis y el flujo de controlo mostrado en la Figura 6.2. La forma más común de escribir esta sentencia se elige entre:

```

switch (expresion) {
    case valor_1: sentencia_1 break;
    case valor_2: sentencia_2 break;
    case valor_n: sentencia_n break;
}
```

```

switch (expresion) {
    case valor_1: sentencia_1 break;
    case valor_2: sentencia_2 break;
    case valor_n: sentencia_n break;
    default: sentencia_x;
}

```

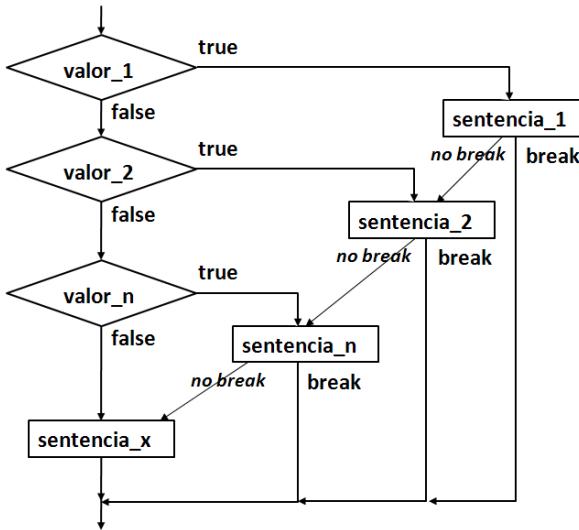


Figura 6.2: Diagrama de flujo de la sentencia `switch`

Notas a la sintaxis:

1. A la palabra reservada `case` le sigue una expresión *constante* y un carácter de *dos puntos*. En principio puede haber cualquier número de etiquetas `case`, aunque en el estándar C99 se especifica que el compilador puede limitar este número. De hecho, normalmente se limita al rango de representación del tipo entero `int`.
2. Cada etiqueta `case` contiene un único valor distinto de la expresión correspondiente a las demás sentencias `case`.
3. Las sentencias que siguen a `case` y que se ejecutan en el caso de que el valor indicado coincida con el de la variable o expresión selector, pueden ser simples, compuestas o incluso no existir. Naturalmente las sentencias simples terminarán en punto y coma y las compuestas se marcarán con las llaves.
4. Las sentencias detrás de un `case` suelen acabar con una sentencia `break`. Si no existe algún `break`, la ejecución continua pasando a través de las siguientes etiquetas hasta el siguiente `break` o hasta el final de la sentencia compuesta que sigue al `switch`. El sentido de las sentencias `break` es, precisamente, que la ejecución del bloque definido por la sentencia `switch` se interrumpa.
5. Si la expresión no coincide con ningún valor de las etiquetas `case` se ejecuta la sentencia (simple o compuesta) que sigue a la etiqueta `default`, aunque dicha etiqueta es opcional.

## 6.6. Ejemplo de uso de `switch`

En el código fuente 6.7 se muestra un ejemplo de uso de la sentencia `switch`.

Ejemplo 6.7: Sentencia `switch`

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     char c = 'e';
7     printf("Inicio del programa\n");

```

```

8
9     switch (c) {
10         case 'a' :
11             printf("Es la vocal a\n");
12             break;
13         case 'e' :
14             printf("Es la vocal e\n");
15             break;
16         case 'i' :
17             printf("Es la vocal i\n");
18             break;
19         case 'o' :
20             printf("Es la vocal o\n");
21             break;
22         case 'u' :
23             printf("Es la vocal u\n");
24             break;
25         case 'A' :
26             printf("Es la vocal A\n");
27         case 'E' :
28         case 'I' :
29         case 'O' :
30         case 'U' :
31             printf("Es una vocal mayuscula\n");
32             break;
33         default :
34             printf("No es una vocal\n");
35             break;
36     } /* fin de switch */
37
38     printf("Fin del programa\n");
39     return 0;
40 }
```

---

Salida por pantalla en la ejecución:

```

Inicio del programa
Es la vocal e
Fin del programa
```

Comienza por el identificador o palabra reservada **switch** seguido de una variable entre paréntesis que es la variable *selección*, en este caso, la variable **c** que es de tipo **char**. La palabra reservada **case** precede cada caso, seguida del valor de la variable en ese caso, un carácter de dos puntos y las sentencias a ejecutar. En el ejemplo, si la variable **c** tiene el valor '**e**' al llegar a la sentencia **switch**, se ejecutará la sentencia **printf("Es la vocal e\n")** y **break** hará que la ejecución salte fuera de la sentencia **switch**. Una vez que se encuentra el punto de entrada, se ejecutarán todas las sentencias que se encuentren a continuación hasta que se encuentre una sentencia **break** o hasta que se llegue al final de la sentencia que sigue a **switch** (delimitada por la llave de cierre). Si la variable **c** tiene el valor **A**, se ejecutan las sentencias encontradas a partir de **case 'A':** hasta encontrar la siguiente sentencia **break**. Los valores de los distintos casos pueden indicarse en cualquier orden y si la variable selección no tiene ninguno de esos valores entonces se ejecuta la sentencia que sigue a **default:** . Debe volver a indicarse que las sentencias anteriores pueden ser anidadas unas dentro de otras o puestas unas a continuación de otras en función de las necesidades de la programación en cada momento.

## Ejercicios resueltos del capítulo de Sentencias selectivas o condicionales

1. En el código fuente de un programa escrito en C, en general las sentencias se terminan utilizando...
  - a) comentarios
  - b) un carácter de punto y coma (;
  - c) un salto de línea, es decir, se escribe una sentencias por línea

- d) con guiones (-)  
e) Ninguna de las anteriores.
2. Indicar cuáles de las siguientes palabras reservadas pueden encontrarse en una sentencia condicional de ramas múltiples o de selección múltiple de C:
- a) `default`  
b) `break`  
c) `switch`  
d) `case`  
e) Ninguna de las anteriores.
3. Construir un programa que indique cuál es el mayor de tres números reales.
4. Construir un programa que calcule el índice de masa corporal de una persona ( $IMC = \text{peso} [\text{kg}] / \text{altura}^2 [\text{m}^2]$ ) e indique el estado en el que se encuentra esa persona en función del valor de IMC
5. Construir un programa que calcule y muestre por pantalla las raíces de la ecuación de segundo grado de coeficientes reales. El programa debe diferenciar los diferentes casos que puedan surgir: la existencia de dos raíces reales distintas, de dos raíces reales iguales y de dos raíces complejas. Nota: se recomienda el empleo de una secuencia de sentencias `if-else-if`.
6. Construir un programa que simule el funcionamiento de una calculadora que puede realizar las cuatro operaciones aritméticas básicas (suma, resta, producto y división) con valores numéricos enteros. El usuario debe especificar la operación con un carácter: `S` o `s` para la suma, `R` o `r` para la resta, `P`, `p`, `M` o `m` para el producto y `D` o `d` para la división. Los valores de los operandos se deben indicar previamente. Nota: Se recomienda el empleo de una sentencia `switch`.

## Soluciones a los ejercicios del capítulo de Sentencias selectivas o condicionales

1. (b)
2. (a), (b), (c) y (d)
3. Programa que calcula el mayor de tres valores numéricos reales

Ejemplo 6.8: Calculo del mayor de tres valores numericos reales

---

```

1 /* programa supremo.c */  

2 /* Programa ejemplo de sentencia if anidadas */  

3  

4 #include <stdio.h>  

5  

6 int main()  

7 {  

8     float a, b, c, supremo;  

9     printf("Programa que indica el mayor de tres reales\n");  

10    a = 6.7; b = 13.4; c = -5.9;  

11    if ( a > b ) {  

12        if ( a > c )  

13            supremo = a;  

14        else  

15            supremo = c;  

16    }  

17    else {  

18        if ( b > c )  

19            supremo = b;  

20        else

```

```

21           supremo=c;
22     }
23   printf("El mayor es %f.\n", supremo);
24   return 0;
25 }
```

---

Salida por pantalla en la ejecución:

Programa que indica el mayor de tres reales  
El mayor es 13.400000.

4. Programa en C que calcula el índice de masa corporal de una persona.

Ejemplo 6.9: Cálculo de índice de masa corporal

```

1  /* programa imc.c                               */
2  /* calcula el IMC de una persona               */
3
4  #include <stdio.h>
5
6  int main()
7 {
8   double peso;        /* Peso en kg          */
9   double altura;      /* Altura en centimetros */
10  double imc;         /* Indice IMC          */
11  peso = 80; altura = 175;
12  imc = peso/(altura*altura);
13  printf("Calculo del indice de masa corporal\n");
14  imc = peso/(altura/100*altura/100);
15  printf("Para un peso de %f kilogramos y ", peso);
16  printf("una altura de %f centimetros\n", altura);
17  printf("el indice de masa corporal es de: %f \n",imc);
18  if ( imc < 16 )
19    printf("Necesita ingresar en un hospital\n");
20  else if (imc<17)
21    printf("Usted tiene infrapeso\n");
22  else if (imc<18)
23    printf("Usted tiene bajo peso\n");
24  else if (imc<26)
25    printf("Usted tiene un peso saludable\n");
26  else if (imc<30)
27    printf("Tiene sobrepeso de grado I\n");
28  else if (imc<35)
29    printf("Tiene obesidad de grado II\n");
30  else if (imc<40)
31    printf("Tiene obesidad premorbida (III)\n");
32  else
33    printf("Usted tiene obesidad morbida o de grado IV\n");
34
35  printf("Fin del programa\n");
36  return 0;
37 }
```

---

Salida por pantalla en la ejecución:

Calculo del indice de masa corporal  
Para un peso de 80.000000 kilogramos y una altura de 175.000000 centimetros  
el indice de masa corporal es de: 26.122449  
Tiene sobrepeso de grado I  
Fin del programa

5. Programa en C que calcula y muestra por pantalla las raíces de la ecuación de segundo grado de coeficientes reales.

## Ejemplo 6.10: Cálculo de las soluciones reales de la ecuación de segundo grado

```

1
2 #include <stdio.h>
3 #include <math.h>
4
5 int main()
6 {
7     double a;           /* Coeficiente de grado 2 */
8     double b;           /* Coeficiente de grado 1 */
9     double c;           /* Coeficiente de grado 0 */
10    double discriminante; /* Discriminante */
11    double x1;          /* Primera raiz real */
12    double x2;          /* Segunda raiz real */
13    double preal;        /* Parte real raiz compleja */
14    double pimag;        /* Parte imag. raiz compleja */
15    a = 2;
16    b = -6;
17    c = 4;
18    discriminante = b*b - 4*a*c;
19    printf("Soluciones de la ecuacion de segundo grado: \n");
20    printf("%f x2 + %f x + %f\n",a,b,c);
21    if (discriminante>0) {
22        /* Dos raices reales */
23        x1 = (-b + sqrt(discriminante))/(2*a);
24        x2 = (-b - sqrt(discriminante))/(2*a);
25        printf("Tiene dos raices reales\n");
26        printf("La primera raiz es x1 = %f\n", x1);
27        printf("La segunda raiz es x2 = %f\n", x2);
28    }
29    else if (discriminante<0) {
30        /* Dos raices imaginarias */
31        preal = (-b)/(2*a);
32        pimag = sqrt(-discriminante)/(2*a);
33        printf("Tiene dos raices complejas\n");
34        printf("La primera raiz es x1 = %f + i %f\n",preal,pimag);
35        printf("La segunda raiz es x2 = %f - i %f\n",preal,pimag);
36    }
37    else {
38        /* Dos raices iguales */
39        x1 = (-b)/(2*a);
40        printf("Dos raices identicas, x1 = x2 = %f\n", x1);
41    }
42    printf("Fin del programa\n");
43    return 0;
44 }
```

Salida por pantalla en la ejecución:

```

Soluciones de la ecuacion de segundo grado:
2.000000 x2 + -6.000000 x + 4.000000
Tiene dos raices reales
La primera raiz es x1 = 2.000000
La segunda raiz es x2 = 1.000000
Fin del programa
```

6. Programa en C que simula el funcionamiento de una calculadora con operaciones aritméticas básicas.

## Ejemplo 6.11: Programa que simula una calculadora

```

1
2 #include <stdio.h>
3
4 int main()
```

```

5  {
6      double a; double b; char c;
7      a = 17.6;
8      b = 5.9;
9      c = 'R';
10     printf("Primer operando: %f\n", a);
11     printf("Segundo operando: %f\n", b);
12     switch (c) {
13         case 'S':
14         case 's':
15             printf("Resultado de la suma: %f\n", (a+b));
16             break;
17         case 'R':
18         case 'r':
19             printf("Resultado de la resta: %f\n", (a-b));
20             break;
21         case 'M':
22         case 'm':
23         case 'P':
24         case 'p':
25             printf("Resultado del producto: %f\n", (a*b));
26             break;
27         case 'D':
28         case 'd':
29             printf("Resultado de la division: %f\n", (a/b));
30             break;
31         default:
32             printf("Por favor, indique otra operacion\n");
33             break;
34     }
35     printf("Fin del programa\n");
36     return 0;
37 }
```

---

Salida por pantalla en la ejecución:

```

Primer operando: 17.600000
Segundo operando: 5.900000
Resultado de la resta: 11.700000
Fin del programa
```

## Ejercicios propuestos del capítulo de Sentencias selectivas o condicionales

1. Construir un programa que indique el menor de tres valores numéricos reales.
2. Construir un programa que indique el valor central de tres valores numéricos reales.
3. Construir un programa que indique si un valor numérico es el cuadrado de otro valor numérico entero.
4. El sistema de cobro de agua de una compañía suministradora considera una cuota mensual fija de servicio (10 euros) y una cuota mensual variable que penaliza el consumo excesivo de la forma que se indica en la tabla siguiente. Construir un programa para que, dados unos metros cúbicos consumidos (valor introducido por teclado durante la ejecución del programa), devuelva el coste mensual total del servicio suministrado, considerando que en la tabla se indica lo que hay que cobrar en la cuota variable por los metros cúbicos que se encuentran en el intervalo correspondiente. Por ejemplo, si se han consumido 55 m<sup>3</sup> se debería pagar en total: 10 + 20\*0.6 + 20\*0.8 + 15\*1.0 , es decir, 53.00 euros. Por ejemplo, si se han consumido 55 m<sup>3</sup> se debería pagar en total: 10 + 20\*0.6 + 20\*0.8 + 15\*1.0 , es decir, 53.00 euros.
5. Construir un programa para que calcule el salario neto semanal de un trabajador en función del número de horas trabajadas (introducidas por teclado por el usuario durante la ejecución del programa) y de los impuestos de acuerdo a las siguientes condiciones: (a) Las primeras 37.5 horas se pagan a la tarifa bruta estándar (18 euros

Cuadro 6.1: Precio por m<sup>3</sup> para la cuota variable según el consumo mensual de agua

Consumo (m <sup>3</sup> )	Precio (euros/m <sup>3</sup> )
Primeros 20 m <sup>3</sup>	0.6
De 20 a 40 m <sup>3</sup>	0.8
De 40 a 60m <sup>3</sup>	1.0
Más de 60 m <sup>3</sup>	1.2

por hora); (b) Las horas de trabajo a partir de 37.5 se pagan a 1.6 veces la tarifa bruta estándar y (c) Impuestos sobre el salario bruto: Los primeros 275 euros están libres de impuestos, el segundo tramo de salario hasta los siguientes 250 euros tienen un 22 % de impuestos y el resto paga el 33 % de impuestos. Por ejemplo, si el usuario introduce 41.5 horas trabajadas, el salario neto semanal es de 647.68 euros

6. Construir un programa que muestre por pantalla el número de días de un mes introducido previamente por teclado. El usuario debe introducir el mes mediante un valor entero. Debe emplearse una sentencia **switch**.
7. Construir un programa que transforme un carácter correspondiente a una cifra hexadecimal en el entero correspondiente. Debe emplearse una sentencia **switch**.

# Capítulo 7

## Sentencias repetitivas o bucles

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Describir el funcionamiento de las sentencias iterativas o bucles: `while`, `do-while` y `for` (Conocimiento).
2. Interpretar y predecir el resultado de una sentencia repetitiva y de una secuencia de varias sentencias de control combinadas o no (Comprensión).
3. Codificar una tarea sencilla, convenientemente especificada, utilizando la secuencia y combinación adecuada de sentencias de control básicas (Aplicación).

### 7.1. Introducción

La programación en el lenguaje C tiene varias estructuras para bucles así como otras que pueden modificar su secuencia de ejecución que se cubrirán en este capítulo.

### 7.2. El bucle while

La sentencia `while` es un bucle o sentencia repetitiva con una condición al principio. Se ejecuta una sentencia mientras una condición o expresión sea distinta de cero (es decir, que sea *cierta*). Cuando la condición deja de cumplirse el bucle finaliza. La sentencia puede que no se ejecute ni una sola vez. La sintaxis es como sigue:

```
while (expresion) sentencia
```

Nótese que la sentencia debe ser simple (terminada en punto y coma) o compuesta (entre llaves).

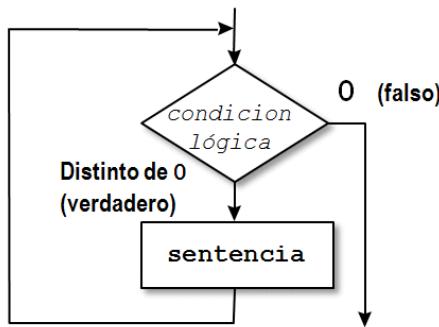
El uso normal de una sentencia `while` es:

```
inicializacion;
while (expresion) {
    expresion_1;
    expresion_2;
    expresion_n;
    iteracion;
}
```

Por otro lado las expresiones de la `1` a la `n` son, en realidad, sentencias porque todas acaban en punto y coma. Al describir el uso se pretende mostrar el aspecto típico del bucle y no tanto su sintaxis. El fluograma se muestra en la Figura 7.1.

El nombre del bucle es muy descriptivo: `while` significa *mientras*. Con este mecanismo, la sentencia puede no ejecutarse ni una sola vez, si la condición es 0 (es decir, *falsa*) nada más llegar al bucle por primera vez. Por otro lado, es importante asegurarse de que, en algún momento de la ejecución de la sentencia repetitiva, la condición va a ser 0 (*falsa*), pues de lo contrario se caería en un *bucle infinito*.

El ejemplo 7.1 muestra el uso de un bucle `while` en el código fuente de un programa en C.

Figura 7.1: Diagrama de flujo de la sentencia de control `while`

## Ejemplo 7.1: Sentencia while

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int contador = 0;
7     while ( contador < 6 ) {
8         printf("El valor de contador es %d\n", contador);
9         contador = contador + 1;
10    } /* Fin de while */
11    return 0;
12 } /* Fin de main */
  
```

---

Salida por pantalla en la ejecución:

```

El valor de contador es 0
El valor de contador es 1
El valor de contador es 2
El valor de contador es 3
El valor de contador es 4
El valor de contador es 5
  
```

El código fuente 7.1 del programa `mientras.c` declara una variable entera `contador` en el cuerpo principal del programa. Se le asigna, en principio, un valor igual a 0 y a continuación se llega al bucle `while`. Al identificador `while` le sigue una expresión entre paréntesis seguida de un bloque de otras sentencias entre llaves. Mientras la expresión entre paréntesis sea cierta se ejecutarán repetidamente todas las sentencias del bloque. En este caso, se ejecutarán seis veces, ya que la variable `contador` irá incrementando su valor desde 0 hasta llegar a 6, momento en el cuál la expresión (`contador<6`) deja de cumplirse y el bucle finaliza. El programa continúa en la sentencia que sigue al bloque (después del cierre de llave). Si a la variable `contador` se le hubiera dado un valor superior a 5 inicialmente, las sentencias incluidas en el bucle no se habrían ejecutado ni una sola vez. Es decir, es posible que las sentencias dentro de un bucle `while` no se ejecuten ni una sola vez. Por otro lado, si el valor almacenado en la variable `contador` no se modificara dentro de las sentencias del bucle, la expresión de control sería siempre cierta, la repetición de la sentencia sería interminable y el programa nunca se acabaría de ejecutar (*bucle infinito*). Finalmente si sólo hubiera una sentencia dentro del bucle se podría utilizar una sentencia simple, pero este caso es muy infrecuente. Es aconsejable compilar y ejecutar el programa anterior hasta asegurarse que se ha entendido el mecanismo completamente.

### 7.3. El bucle do-while

La sentencia `do-while` es un bucle condicional con test al final. Ejecuta una sentencia o sentencias mientras una condición o expresión sea distinta de cero (es decir, que sea *cierta*). Con este mecanismo, la sentencia siempre se ejecuta al menos una vez.

La sintaxis queda como sigue:

```
do sentencia while (expresion) ;
```

El uso normal de un bucle `do while` queda de la siguiente manera:

```
do {
    expresion_1;
    expresion_2;
    expresion_n;
    iteracion;
}
while (condicion);
```

Igual que el caso anterior con la expresión *uso* del bucle se entiende la apariencia que tiene el bucle normalmente y no su sintaxis estricta. El flujo de control se muestra en la Figura 7.2.

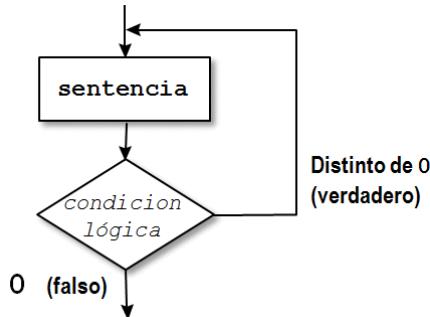


Figura 7.2: Diagrama de flujo de la sentencia de control `do-while`

Como en el tipo de bucle anterior es importante asegurarse de que, en algún momento de la ejecución, la condición va a ser 0 (es decir, falsa), pues de lo contrario se caería en un bucle infinito.

El código fuente 7.2 del programa `repite.c` presenta un ejemplo de utilización del bucle `do-while`.

Ejemplo 7.2: Sentencia do-while

---

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int contador = 0;
6     do {
7         printf("El valor de contador es %d\n", contador);
8         contador = contador + 1;
9     } while (contador < 5);
10    return 0;
11 }
```

---

Salida por pantalla en la ejecución:

```

El valor de contador es 0
El valor de contador es 1
El valor de contador es 2
El valor de contador es 3
El valor de contador es 4
```

El programa es muy parecido al ejemplo 7.1, exceptuando que el bucle comienza con el identificador `do` seguido de una serie de sentencias entre llaves y luego el identificador `while`, una expresión entre paréntesis y el punto y coma final. Las sentencias entre llaves se ejecutan repetidamente hasta que la expresión entre paréntesis deja de ser cierta. En ese momento, el bucle deja de repetirse y continúa la ejecución en la sentencia siguiente. Como la verificación de la expresión entre paréntesis se realiza por primera vez al final de la primera ejecución de las sentencias del bucle, las sentencias dentro del bucle se ejecutan al menos una vez. Si la variable `contador` del ejemplo no se modificara dentro del bucle, éste sería infinito.

## 7.4. El bucle for

El bucle **for** se utiliza generalmente para repetir sentencias cuando el número de veces que se van a repetir es conocido y se usa un contador para gestionar las veces que se repite. La sintaxis es como sigue:

```
for ( expresion_opc_1 ; expresion_opc_2 ; expresion_opc_3 )
    sentencia
```

En la figura 7.3 se muestra el diagrama de flujo del bucle **for**.

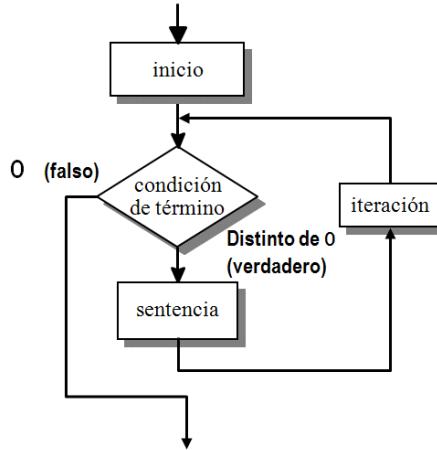


Figura 7.3: Diagrama de flujo de la sentencia de control **for**

Nótese que las tres expresiones son opcionales y que, como habitualmente, la sentencia puede ser simple acabada en punto y coma o compuesta entre llaves. Un ejemplo de uso *habitual* del bucle **for** tiene la siguiente forma:

```
for ( i = 0; i < 10; i++ ) {
    expresion_1;
    expresion_n;
}
```

La sentencia **for** es un bucle o sentencia repetitiva que:

1. ejecuta la sentencia de *inicio* o *inicialización* (**expresion\_opc\_1**)
2. verifica la expresión booleana de *término* (**expresion\_opc\_2**)
  - a) si es cierta, ejecuta la sentencia entre llaves y la sentencia de *iteración* (**expresion\_opc\_3**) para volver a verificar la expresión booleana de *término* (**expresion\_opc\_2**)
  - b) si (**expresion\_opc\_2**) es falsa, sale del bucle.

El código fuente 7.3 del programa **durante.c** presenta un ejemplo de utilización del bucle **for**.

Ejemplo 7.3: Sentencia for

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int contador;
7     for ( contador = 0; contador < 5; contador = contador + 1)
8         printf("El valor de la variable contador es %d\n", contador);
9     return 0;
10 }
```

---

Salida por pantalla en la ejecución:

```

El valor de la variable contador es 0
El valor de la variable contador es 1
El valor de la variable contador es 2
El valor de la variable contador es 3
El valor de la variable contador es 4

```

El código fuente del programa incluye un identificador **for** seguido de una expresión compuesta entre paréntesis. Esta expresión se compone de tres campos o términos separados por caracteres de punto y coma. El primer campo del ejemplo contiene la expresión **contador=0** y es sentencia de inicio o *inicialización*. Cualquier expresión que se encuentre en este primer campo se ejecutará *antes* de pasar al bucle. No hay un límite teórico acerca de lo que se puede poner introducir aquí, pero un buen programador no lo hará demasiado complejo (pueden incluirse varias expresiones de inicialización en este campo separadas por comas). También podrían declararse variables cuyo uso se permite exclusivamente dentro del bucle (variable de ámbito *local*). Pero dado que esta declaración no se permite en el estándar C89, en principio se desaconseja esta práctica. El segundo campo (*término*) contiene la expresión **contador<5** que es el *test* que se hace al principio de cada repetición del bucle. Esta expresión debe evaluarse como 1 ó distinto de 0 (verdadera) o como 0 (falsa). La expresión contenida en el tercer campo (*iteración*) se ejecutará al final de la repetición de las sentencias del bucle. Este campo, también puede componerse de varias expresiones separadas por comas. A continuación de la expresión **for()** aparece la sentencia simple o compuesta que se ejecutará en el bucle.

Los bucles **while** y **do-while** son convenientes cuando no se sabe a priori cuantas veces debe ejecutarse el bucle mientras que el bucle **for** debe emplearse en aquellos casos en los que se desea realizar un número fijo de repeticiones. El bucle **for** tiene además la ventaja de que pone toda la información de control en un mismo sitio entre paréntesis. En cualquier caso la elección dependerá de problema a resolver. Dependiendo de cómo se usen es posible que en ninguno de los dos bucles (**while** y **for**) la sentencia correspondiente se repita ni una sola vez, ya que el test se realiza al principio de los mismos. En el bucle **do-while**, sin embargo, como la sentencia se ejecuta antes del test, el código se ejecutará al menos una vez.

Es de resaltar, por otro lado, que los bucles pueden estar anidados. Es decir, un bucle puede estar incluido dentro de otro, no estando, en principio, el nivel de anidamientos limitado, aunque en la práctica resulta difícil encontrar problemas con más de 3 o 4 niveles de anidamiento de bucles en un mismo bloque.

## 7.5. Combinación de distintos tipos de sentencias anidadas

El programa en el ejemplo 7.4 es una utilidad que genera una lista de temperaturas con valores correspondientes en las escalas centígrada y Fahrenheit y visualiza un mensaje con los valores de los puntos de congelación y de ebullición del agua. Quizás no sea un programa muy complicado pero lo importante es el formato del código incluido.

Ejemplo 7.4: Lista de temperaturas

---

```

1  ****
2  /*
3  /* Este programa de conversion de temperatura esta escrito */
4  /* en el lenguaje de programacion C. Genera y visualiza por */
5  /* pantalla una tabla de temperaturas en grados centigrados */
6  /* y fahrenheit
7  */
8  ****
9
10 #include <stdio.h>
11
12 int main()
13 {
14     int cont;          /* variable de control del bucle */
15     int fahrenheit;   /* temperatura en grados fahrenheit */
16     int centigrados; /* temperatura en grados centigrados */
17
18     printf("Tabla de conversion de Centigrados a Fahrenheit\n\n");
19
20     for ( cont = -2; cont <= 12; cont = cont + 1 ) {
21         centigrados = 10 * cont;
22         fahrenheit = 32 + (centigrados * 9) / 5;
23         printf(" C=%4d F=%4d ", centigrados, fahrenheit);

```

```

24     if (centigrados == 0)
25         printf(" Punto de fusion del agua");
26     if (centigrados == 100)
27         printf(" Punto de ebullicion del agua");
28     printf("\n");
29 } /* Fin del bucle */
30
31 return 0;
32 }
```

---

Salida por pantalla en la ejecución:

Tabla de conversion de Centigrados a Fahrenheit

```

C = -20    F =   -4
C = -10    F =   14
C =  0      F =   32  Punto de fusion del agua
C =  10     F =   50
C =  20     F =   68
C =  30     F =   86
C =  40     F =  104
C =  50     F =  122
C =  60     F =  140
C =  70     F =  158
C =  80     F =  176
C =  90     F =  194
C = 100    F = 212  Punto de ebullicion del agua
C = 110    F = 230
C = 120    F = 248
```

Al principio se incluyen unas líneas de comentarios dispuestos de una forma muy fácil de leer que describen lo que hace el programa. En el bucle **for** las sentencias que se repiten están indentadas unos espacios hacia la derecha lo que también facilita su reconocimiento. Las llaves están alineadas verticalmente con el identificador **for**. Las sentencias **printf()** también están indentadas hacia la derecha lo que hace más fácil su lectura y seguimiento. Se han declarado varias variables en programa utilizando una línea por variable para ello. Se aprovecha esta opción para introducir un comentario para cada una de ellas. Podrían haberse declarado en la misma línea pero se hubiera perdido la posibilidad de incluir el comentario correspondiente a cada una.

## 7.6. Un ejemplo de programación pobre

El programa en el ejemplo 7.5 hace lo mismo que el ejemplo 7.4 visto anteriormente pero tiene un estilo bastante peor porque desaparecen los comentarios y los identificadores de las variables no son descriptivos.

Ejemplo 7.5: Lista de temperaturas, estilo confuso

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int x1, x2, x3;
7     printf("Tabla de conversion de grados centrigados a Fahrenheit\n\n");
8     for(x1=-2 ; x1<=12 ; x1=x1+1)
9     {
10         x3 = 10*x1;
11         x2 = 32+(x3*9)/5;
12         printf(" C =%4d    F =%4d  ", x3, x2);
13         if (x3==0)
14             printf(" Punto de fusion del agua");
15         if (x3==100)
16             printf(" Punto de ebullicion del agua");
```

```

17         printf("\n");
18     }
19     return 0;
20 }
```

---

Salida por pantalla en la ejecución:

Tabla de conversion de grados centrigados a Fahrenheit

```

C = -20   F =  -4
C = -10   F =  14
C =  0    F =  32  Punto de fusion del agua
C =  10   F =  50
C =  20   F =  68
C =  30   F =  86
C =  40   F = 104
C =  50   F = 122
C =  60   F = 140
C =  70   F = 158
C =  80   F = 176
C =  90   F = 194
C = 100  F = 212  Punto de ebullicion del agua
C = 110  F = 230
C = 120  F = 248
```

De este ejemplo pueden aprenderse cosas que no deben hacerse en programación ya que el resultado es un programa que presenta más dificultades de lectura y comprensión.

## 7.7. Diferentes buenos estilos de programación

El programa `estilo.c` del ejemplo 7.6 no hace nada útil salvo ilustrar diferentes estilos de formato en algunas de las sentencias y estructuras vistas en este capítulo. Presenta algunos detalles de formato de escritura que deberían seguirse para ir aprendiendo técnicas de buena programación.

Ejemplo 7.6: Sentencias repetitivas combinadas

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int indice;
7     int cont = 3;
8     int cont_2;
9
10    for (indice = 0 ; indice < 5 ; indice = indice + 1) {
11        printf("El valor de ");
12        printf("indice es %d\n", indice);
13        if (cont < 3) {
14            printf("El valor de cont es %d ", cont);
15            printf(" que es menor que 3\n");
16        }
17        else {
18            cont_2 = 0;
19            do {
20                printf("El valor de cont_2 es %d\n", cont_2);
21                cont_2 = cont_2 + 1;
22            } while (cont_2 < 2);
23            printf("El valor de cont es %d ", cont);
24            printf(" que no es menor que 3\n");
25        }
}
```

```

26     }
27     return 0;
28 }
```

---

Salida por pantalla en la ejecución:

```

El valor de indice es 0
El valor de cont_2 es 0
El valor de cont_2 es 1
El valor de cont es 3 que no es menor que 3
El valor de indice es 1
El valor de cont_2 es 0
El valor de cont_2 es 1
El valor de cont es 3 que no es menor que 3
El valor de indice es 2
El valor de cont_2 es 0
El valor de cont_2 es 1
El valor de cont es 3 que no es menor que 3
El valor de indice es 3
El valor de cont_2 es 0
El valor de cont_2 es 1
El valor de cont es 3 que no es menor que 3
El valor de indice es 4
El valor de cont_2 es 0
El valor de cont_2 es 1
El valor de cont es 3 que no es menor que 3
```

## Ejercicios resueltos del capítulo de Sentencias repetitivas o bucles

1. ¿Qué palabra reservada acompaña siempre a `do` en el bucle correspondiente?
2. Indicar con cuáles de los siguientes grupos de palabras reservadas puede construirse un bucle o sentencia repetitiva en C:
  - a) `switch ... case ...`
  - b) `for ...`
  - c) `while ...`
  - d) `do ... while ...`
  - e) `if ... else ...`
  - f) Ninguno de los anteriores
3. Escribir un programa que escriba el nombre del programador diez veces por pantalla. Desarrollar tres versiones del programa empleando los tres bucles vistos en este capítulo.
4. Construir un programa que calcule el factorial de un valor numérico entero.
5. Construir un programa que calcule la potencia de una base real elevado a un exponente entero utilizando un bucle.
6. Construir un programa que calcule el número de años que tarda en duplicarse una cantidad de dinero invertida a un interés anual constante.
7. Construir un programa que, empleando dos bucles anidados, visualice por pantalla las tablas de multiplicar del 7 y del 8.
8. Construir un programa que muestre por pantalla todos los valores primos entre el 1 y el 300. Def.: un número entero es primo si sólo es divisible por sí mismo y por la unidad. Por ejemplo: 2, 3, 5, 13 y 29 son números primos, mientras que 4, 99, 169 no lo son. El valor entero 1 no se considera primo.

9. Construir un programa que simule el juego de la adivinanza de un número. El ordenador debe generar un número aleatorio entre 1 y 100 y el usuario tiene cinco oportunidades para acertarlo. Después de cada intento el programa debe indicarle al usuario si el número introducido por él es menor, mayor o igual al número a adivinar. Nota: para generar el valor aleatorio puede emplearse las sentencias: `srand (time (0)); /* Genera la semilla */`  
`n = rand() % 100 + 1;`

## Soluciones a los ejercicios del capítulo de Sentencias repetitivas o bucles

1. `while`
2. (b), (c) y (d)
3. Programas que escriben el nombre del programador diez veces empleando diferentes bucles:

Ejemplo 7.7: Repetición con bucle while

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int i = 1;
7     while ( i <= 10 ) {
8         printf("Me llamo Juan Programas\n");
9         i++;
10    }
11    return 0;
12 }
```

---

Ejemplo 7.8: Repetición con bucle do-while

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int i = 1;
7     do {
8         printf("Me llamo Juan Programas\n");
9         i++;
10    } while ( i <= 10 );
11    return 0;
12 }
```

---

Ejemplo 7.9: Repetición con bucle for

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int i;
7     for ( i = 1; i <= 10; i++ ) {
8         printf("Me llamo Juan Programas\n");
9     }
10    return 0;
11 }
```

---

Los tres programas anteriores tienen la misma salida por pantalla al ser ejecutados:

Me llamo Juan Programas  
 Me llamo Juan Programas

```
Me llamo Juan Programas
```

4. Programa que calcula el factorial de un valor numérico entero.

Ejemplo 7.10: Cálculo del factorial de un número

---

```
1 #include <stdio.h>
2
3
4 int main()
5 {
6     int n = 7, f = 1, i;
7     for ( i = 2; i<=n; ++i ) {
8         f = f*i;
9     }
10    printf("El factorial de %d es %d \n", n, f);
11    printf("Fin del programa\n");
12    return 0;
13 }
```

---

Salida por pantalla en la ejecución:

```
El factorial de 7 es 5040
Fin del programa
```

5. Programa que calcula la potencia de una base real elevada a un exponente entero

Ejemplo 7.11: Bucle para calcular una potencia

---

```
1
2 #include <stdio.h>
3 #include <math.h>
4
5 int main()
6 {
7     float b;      /* Base */
8     int e;        /* Exponente */
9     float p = 1;  /* Potencia */
10    int k;        /* Auxiliar */
11    b = 2.73;
12    e = 3;
13    p = 1;
14    for ( k=1; k <= fabs(e); ++k ) {
15        p = p*b;
16    }
17    if ( e < 0 ) {
18        p = 1/p;
19    }
20    printf("El valor %f elevado a %d es %f\n", b, e, p);
21    printf("Fin del programa\n");
22    return 0;
23 }
```

---

Salida por pantalla en la ejecución:

```
El valor 2.730000 elevado a 3 es 20.346416
Fin del programa
```

6. Programa que calcula el número de años para duplicar una cantidad invertida a un interés constante:

Ejemplo 7.12: calcula años para duplicar inversion

---

```

1  /* duplicai.c                                     */
2  /* Ejemplo de bucle while                         */
3  /* Calcula cuantos años deben pasar para duplicar una cantidad */
4  /* invertida a un determinado interés anual constante      */
5
6 #include <stdio.h>
7
8 int main()
9 {
10    double cantidadInicial=1000;
11    double cantidad=cantidadInicial;
12    double interes=4;
13    int anhos=0;
14    printf("La cantidad inicial es: %f\n",cantidadInicial);
15    while (cantidad < 2*cantidadInicial) {
16        anhos++;
17        cantidad += cantidad*interes/100;
18    }
19    printf("El interes es: %f\n",interes);
20    printf("La cantidad final es: %f\n",cantidad);
21    printf("El numero de años es: %d\n",anhos);
22    return 0;
23 } /* Fin de main */
```

---

Salida por pantalla en la ejecución:

```

La cantidad inicial es: 1000.000000
El interes es: 4.000000
La cantidad final es: 2025.816515
El numero de años es: 18
```

7. Programa que visualiza por pantalla las tablas de multiplicar del 7 y del 8.

Ejemplo 7.13: tablas de multiplicar

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int i,j;
7     printf("Tablas de multiplicar:\n");
8     for ( i = 7; i <= 8; i++) {
9         printf("***** Tabla del %d: *****\n", i);
10        for ( j = 0; j <= 10; j++) {
11            printf("%d x %d vale %d\n",i,j,i*j);
12        } /* Fin del for interno */
13    } /* Fin del for externo */
14    return 0;
15 } /* Fin de main */
```

---

Salida por pantalla en la ejecución:

```

Tablas de multiplicar:
***** Tabla del 7: *****
7 x 0 vale 0
7 x 1 vale 7
7 x 2 vale 14
7 x 3 vale 21
```

```

7 x 4 vale 28
7 x 5 vale 35
7 x 6 vale 42
7 x 7 vale 49
7 x 8 vale 56
7 x 9 vale 63
7 x 10 vale 70
***** Tabla del 8: *****
8 x 0 vale 0
8 x 1 vale 8
8 x 2 vale 16
8 x 3 vale 24
8 x 4 vale 32
8 x 5 vale 40
8 x 6 vale 48
8 x 7 vale 56
8 x 8 vale 64
8 x 9 vale 72
8 x 10 vale 80

```

8. Programa que visualiza por pantalla los primos del 1 y al 300.

Ejemplo 7.14: primos entre el 1 y el 300

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int n, i, k, linea = 0;
7     for ( n = 1; n <= 300; n++) {
8         k = 0;
9         for ( i = 1; i <= n; i++) {
10             if (n % i == 0) {
11                 k++;
12             }
13         }
14         if ( k == 2 && linea < 10 ) {
15             printf("%d ",n);
16             ++linea;
17         }
18         else if ( k == 2 && linea >= 10 ) {
19             printf("%d\n",n);
20             linea = 0;
21         }
22     }
23     return 0;
24 }
```

---

Salida por pantalla en la ejecución:

```

2 3 5 7 11 13 17 19 23 29 31
37 41 43 47 53 59 61 67 71 73 79
83 89 97 101 103 107 109 113 127 131 137
139 149 151 157 163 167 173 179 181 191 193
197 199 211 223 227 229 233 239 241 251 257
263 269 271 277 281 283 293

```

9. Programa juego para adivinar un número

## Ejemplo 7.15: Juego de adivinar un numero

---

```

1  /* programa adivina.c */ 
2  /* Juego para adivinar un numero del 1 al 100 */
3  /* Combina diferentes sentencias de control */
4
5 #include <stdio.h>           /* Para poder emplear printf y scanf */
6 #include <stdlib.h>          /* Para poder emplear srand y rand */
7 #include <time.h>            /* Para poder emplear time */
8
9 int main()
10 {
11     int valor, x;
12     int oportunidad = 0;
13     srand (time(0)); /*Genera la semilla aleatoria */
14     x = rand() % 100 + 1; /* Entero aleatorio entre 1 y 100 */
15     do {
16         oportunidad++;
17         scanf("%d",&valor);
18         printf("Valor introducido: %d", valor);
19         if ( valor < x )
20             printf("Demasiado bajo\n");
21         else if ( valor > x )
22             printf("Demasiado alto\n");
23         else
24             printf("Numero acertado\n");
25     } while ((valor!=x) && (oportunidad<5));
26     if ( valor != x )
27         printf("Ha perdido las 5 oportunidades\n");
28     printf("El numero secreto era: %d\n",x);
29     return 0;
30 }
```

---

Al introducir por teclado en la ejecución:

1 25 50 75 100

Salida por pantalla en la ejecución:

Valor introducido: 1Demasiado bajo  
 Valor introducido: 25Demasiado bajo  
 Valor introducido: 50Demasiado bajo  
 Valor introducido: 75Demasiado bajo  
 Valor introducido: 100Demasiado alto  
 Ha perdido las 5 oportunidades  
 El numero secreto era: 90

## Ejercicios propuestos del capítulo de Sentencias repetitivas o bucles

- Construir un programa que visualice por pantalla todos los caracteres correspondientes a letras minúsculas.
- Escribir un programa que cuente de uno a diez, imprima los valores en líneas distintas e incluya un mensaje específico con el valor 3 y otro con el valor 7.
- Construir un programa que calcule y visualice por pantalla el factorial de todos los valores numéricos enteros entre 1 y 10.
- Construir un programa que calcule el número de años en el que se duplica una población que aumenta con una tasa de crecimiento positiva y fija anualmente utilizando un bucle.
- Programa que juegue con el usuario a adivinar un número entre el 1 y el 200. El usuario tiene siete oportunidades para acertar y el ordenador le indica si el número introducido es mayor o menor que el secreto.

6. Declaradas tres variables **a**, **b** y **c** como de tipo **int** en un programa de C, indicar la salida por pantalla al ejecutarse el siguiente código fuente:

```
a = 0; b = 28; c = 4;  
while ( c < b ) {  
    c = c + 7;  
}  
printf("%d",c);
```

## **Parte III**

# **Elementos avanzados de programación**



# Capítulo 8

## Rutinas o funciones

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Describir el mecanismo de funcionamiento de una rutina o función, así como su utilidad (Conocimiento)
2. Definir los conceptos: cabecera, parámetros formales, variables locales, resultado de la función, llamada a la rutina, argumentos reales y paso de parámetros (Conocimiento)
3. Interpretar el resultado de la declaración y llamada de una rutina dentro del código fuente de un programa (Comprensión)
4. Codificar una tarea sencilla convenientemente especificada utilizando una rutina (Aplicación)

### 8.1. Concepto de rutina o función

Las rutinas o funciones son pequeños programas en un lenguaje de programación que pueden ser ejecutados en varios puntos de cualquier programa o de otra rutina sin necesidad de repetir el código fuente correspondiente. Las funciones tienen un *identificador* que representan ese conjunto de instrucciones y pueden devolver un resultado al programa que las ejecute o realizar una serie de tareas, sin devolver un resultado. Para obtener el resultado o realizar las tareas pueden necesitar valores que son soportados por unas variables especiales denominadas *parámetros* de la rutina.

Las rutinas pueden además almacenarse independientemente en colecciones llamadas librerías o bibliotecas, lo cual permite que sean utilizadas por cualquier programa. De hecho, existen muchas funciones que vienen ya construidas para el lenguaje de programación C (*cos*, *sin*, *exp*, *printf*...), que están almacenados en distintos archivos de cabecera de la librería estándar de C (*stdio*, *stdlib*, *string*, *math*, *time*,...) y que el programador puede emplear en sus programas.

El lenguaje C se basa en el uso de funciones. De hecho, siempre debe existir una función denominada *main* dentro del código del programa. Para utilizar funciones dentro del lenguaje C hay que realizar dos tareas:

1. Definir la función, es decir, definir el nombre de la rutina, la información que necesita para calcular su resultado, el tipo de dicho resultado y las instrucciones que se deben ejecutar para que cumpla con su objetivo.
2. Invocar la función por su nombre para ejecutarla con unos valores concretos de los parámetros. La invocación a la rutina también se denomina *llamada a la función* (*function call*).

A continuación, se describen estas dos tareas de forma mas detallada.

### 8.2. Definición de funciones

La sintaxis para definir una función en el lenguaje C es:

```
Ret_t idFunc (Par1_t par1, Par2_t par2, ParN_t parN)
{
    Declaración de variables;
    Instrucciones o sentencias en lenguaje C;
}
```

donde:

**Ret\_t** el tipo del retorno o resultado que devuelve la rutina.

**idFunc** es el nombre o identificador de la función.

( ) entre paréntesis aparece una lista de tipos y parámetros separados por comas.

**par1, par2, parN** son los identificadores de los parámetros formales de la función (*formal parameters*).

**Par1\_t, Par2\_t, ParN\_t** son los tipos respectivos de los anteriores identificadores

{ } llaves que agrupan todas las definiciones de variables y las sentencias que implementan la función.

La definición de la función se divide en dos partes:

1. la primera línea es la cabecera o declaración (*declaration*) de la función donde se indica el nombre, el tipo de retorno de la función y los datos que necesita para ejecutarse correctamente;
2. la segunda parte son las llaves con el código de la función y se denomina implementación (*implementation*) de la rutina.

Notas de sintaxis:

- La obligación de realizar las declaraciones de variables en primer lugar dentro de una función forma parte del estándar C89, también llamado ANSI C. En el estándar C99 esta obligación no existe. No obstante, se recomienda hacerlo de esa manera para mejorar la portabilidad del código fuente.
- En C existe un tipo especial **void** para indicar la ausencia de un dato o la indeterminación de un tipo específico. En el caso de funciones el tipo **void** se puede utilizar para expresar la ausencia del retorno de la función.

### 8.3. Ejecución de funciones

Para ejecutar o *llamar a una función* hay que invocarla por su nombre indicando el valor concreto que deben tomar los parámetros formales declarados en la cabecera de su definición. Se reservará memoria para los parámetros formales y variables locales y se comenzarán a ejecutar una tras otra las instrucciones encerradas entre llaves y finalizará este proceso cuando se encuentre una instrucción **return** que devuelve el resultado de la función o bien, cuando se llegue al final de las llaves <sup>1</sup>. Al finalizar la función se libera la memoria reservada para parámetros formales y variables locales. Los valores concretos que se transfieren a los parámetros formales se denominan **argumentos reales** (*actual arguments*). Los argumentos reales deben incluirse en el mismo orden en que se escribieron sus correspondientes parámetros formales en la cabecera de la definición de la función y ser del mismo tipo o convertibles de forma automática. Ni antes, ni cuando finaliza la ejecución de la función los parámetros formales y las variables locales definidas en la función son accesibles dentro del programa que invocó a la función.

La sintaxis simplificada de la llamada a una función es:

```
identFuncion(arg1, arg2, argN)
```

donde:

**identFuncion** es el identificador de la función.

( ) los paréntesis son, sintácticamente, el operador llamada a función. Normalmente, esto no tiene mayor importancia, simplemente en una llamada a función pondremos los paréntesis y dentro los argumentos.

**arg1, arg2, argN** son la lista de los argumentos separados entre sí por comas. Cada argumento debe corresponderse con uno de los parámetros indicado en la declaración de la función mediante la posición en la que se encuentra. Normalmente esto implica que la función se tiene que llamar con una lista de argumentos muy específica tanto en número de argumentos como en el tipo de cada uno de ellos.

**Retorno** la llamada a una función de resultado distinto de **void** produce un valor que se puede utilizar en una expresión.

Para visualizar este comportamiento podemos imaginar que toda la llamada a la función se sustituye por el valor de retorno.

---

<sup>1</sup>El compilador puede dar avisos o errores si no se devuelve un resultado con **return** en una función con tipo distinto de **void**

## 8.4. Parámetros de funciones

El retorno de la función y los parámetros son los mecanismos que hacen posible el intercambio de información datos entre la función y quién llama a la rutina (otra función, por ejemplo el programa principal, la función `main`).

El intercambio de información entre la función que llama y la que se ejecuta se denomina *paso de parámetros*.

Cuando una función `f1` necesita ejecutar otra función `f2` y se necesita transferir datos de `f1` a `f2` o de `f2` a `f1` entonces el mecanismo que C utiliza son los parámetros formales (*formal parameters*) en la definición de la función `f2` y los argumentos (*actual arguments*) que aparecen en `f1` cuando se escribe la sentencia con la llamada a `f2`. Un caso particular es el caso en que la rutina es la función `main` (el programa principal). Nótese que, además de los parámetros, existe otro mecanismo de intercambio de información entre la función que llama y la que se ejecuta: el retorno de la función. Sintácticamente este esquema se podría observar en el código fuente de la siguiente manera:

```
tipoRetorno f2 ( tipo1 param1, tipo2 param2 ) {
    /* Implementación de f2 */
}

tipoRetorno f1 ( /* Parámetros de f1 */ ) {
    tipoRetorno resultado;
    tipo1 variable1 = valor1;
    tipo2 variable2 = valor2;
    resultado = f2 ( variable1, variable2 );
}
```

Existen tres formas de transferir información de una función a otra:

1. De *entrada*, es decir, la información se transmite de `f1` a `f2`. En inglés se suele utilizar la preposición *in* para indicar este concepto.
2. De *entrada-salida*, es decir, la información se transmite de `f1` a `f2` al iniciarse la ejecución de la función `f2` y luego se transmite de `f2` a `f1` al acabar la ejecución de `f2`. En inglés se suele utilizar *in-out* para indicar este concepto.
3. De *salida*, es decir, la información se transmite de `f2` a `f1` al acabar la función mediante un valor que se puede utilizar en una expresión. En inglés este concepto se suele indicar como *out*.

Entre los dos primeros, en C solo es posible el primero de los mecanismos de forma directa, mediante lo que se conoce como paso de parámetros formales *por valor*. El segundo mecanismo se consigue de manera indirecta a través del uso de punteros. Se pasan parámetros de tipo puntero por valor o como entrada. Sin embargo, la naturaleza de dirección de memoria de los punteros provoca que este tipo de paso de parámetros resulte equivalente a lo que se conoce como parámetros formales *por referencia* de las variables apuntadas por los punteros.

### Parámetros formales por valor

Los parámetros formales por valor son variables que reciben el valor del correspondiente argumento real (*actual argument*). Cuando se ejecuta la función se reserva espacio en memoria para el parámetro formal, es decir, se instancia y el argumento real copia su valor sobre dicha instancia, mediante el equivalente a una asignación.

Las transformaciones que se realizan en el parámetro real durante la ejecución de la función afectan por tanto solo a la instancia que contiene la copia del original, permaneciendo el argumento real protegido.

Este tipo de parámetros tiene como objetivo introducir información en la función que se ejecuta, el flujo de información es por tanto unidireccional desde la función que efectúa la llamada hacia la función que se ejecuta.

Un ejemplo de este tipo de parámetros viene ilustrado por el siguiente programa, en el que la función que efectúa la llamada es la función `main`. El programa utiliza una función `suma1` que añade 3 unidades al parámetro formal `x` e imprime por pantalla su nuevo valor, devolviendo como resultado el valor 0. En el programa principal se definen dos variables `a` y `b`, `a` toma el valor de 5 y `b` almacenará el resultado de la función cuando esta finalice su ejecución. La variable `a` es el argumento real en la llamada a la función `suma1`. La función `suma1` tiene un parámetro formal por valor definido en la cabecera de la función: `x`. Cuando se ejecuta la función en la sentencia `b=suma1(a)`; en la zona de la memoria reservada para `x` se copia el valor que tiene la variable `a`, de forma que todas las instrucciones de la función `suma1` pueden ejecutarse puesto que el parámetro genérico `x` ya tiene un valor concreto. Tras la ejecución de

la sentencia `return x;` el valor de `x` es transferido al programa o función `main` y se asigna a la variable `b`. Las últimas instrucciones visualizan el contenido de las variables `a` y `b`, como se puede observar, `a` mantiene el valor que se le asignó en la primera instrucción del programa. El argumento real, la variable `a` mantiene su valor y la variable `x` aunque fue modificada es inaccesible, una vez finalizada la ejecución de la función `suma1`.

Ejemplo 8.1: Uso de parámetros por valor

---

```

1  #include <stdio.h>
2
3
4  int suma1(int x)
5  {
6      x = x + 3;
7      printf("El valor de x es %d\n",x);
8      return x;
9 }/*suma1*/
10
11 int main()
12 {
13     /* Declaraciones primero */
14     int a, b;
15
16     /* Sentencias despues */
17     a = 5;
18     b = suma1(a);
19
20     printf("El valor de a es %d\n",a);
21     printf("El valor de b es %d\n",b);
22     return 0;
23 }/*main*/

```

---

Salida del programa:

```

El valor de x es 8
El valor de a es 5
El valor de b es 8

```

El proceso de paso de parámetros por valor se muestra también en la figura 8.1, en dicha figura el cuadro mostrado como parámetro real se corresponde con la variable `a`, pasada como argumento a la función, mientras que el cuadro mostrado como parámetro formal es el parámetro `x`.

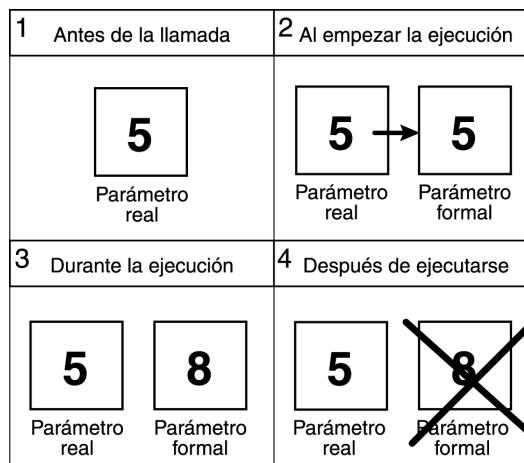


Figura 8.1: Paso de parámetro por valor

## 8.5. Funciones de la librería estándar

Existen numerosas funciones de la librería estándar cuya declaración está disponible en los distintos archivos de cabecera estándar: `stdlib.h`, `stdio.h`, `string.h`, `assert.h`, etc. Todas estas funciones se pueden añadir a cualquier programa para resolver problemas frecuentes de programación. Más aún, C cuenta con una extensa gama de librerías que se pueden añadir en los programas escritos en C. Estas librerías se pueden obtener de formas muy diversas, desde desarrollos en abierto y gratis hasta librerías comerciales cerradas y de pago. Y por muy distintos medios, destacando por su importancia las librerías que se pueden descargar directamente a través de la Web e Internet (Por ejemplo en *SourceForge*) .

En cualquier caso, para usar o llamar a una función es necesario tener su declaración (en un archivo de cabeceras) y su implementación. Esto último normalmente se obtendrá a través de una librería como se explicará en el capítulo correspondiente. No es necesario obtener la librería estándar, ya está incluida con el compilador y se utiliza sin necesidad de indicarlo expresamente <sup>2</sup>.

## Ejercicios propuestos del capítulo de Funciones

- Indicar la salida por pantalla al ejecutar el siguiente programa:

Ejemplo 8.2: Declaracion y uso de la función esPositivo

---

```

1  /* positivos.c */
2  #include <stdio.h>
3  int esPositivo (int n) {
4      if (n<0) return 0;
5      else return 1;
6  }
7  int main() {
8      int i;
9      for (i=5; i>=-5; i--) {
10         printf("%d es positivo: %d\n", i, esPositivo(i));
11     }
12     return 0;
13 }
```

---

- Construir una función que devuelva el cubo de un valor numérico real dado como parámetro de la función.
- Construir una función `decimal` que devuelva como resultado el valor decimal (`int`) correspondiente a cualquier dígito hexadecimal (en mayúsculas) almacenado en el parámetro `c` de tipo `char`. Por ejemplo: la llamada a `decimal('6')` devuelve 6, `decimal('A')` devuelve 10 y `decimal('F')` devuelve 15. Si el parámetro no corresponde a un valor hexadecimal la función debe devolver -1.
- Construir una función para que dado un punto de coordenadas (x,y) devuelva un valor real z. Si el punto pertenece al primer cuadrante entonces devuelve  $z=(x+y)$ , si está en el segundo cuadrante devuelve  $z=y$ , si está en el tercer cuadrante devuelve  $z=0$  y, por último, si está en el cuarto cuadrante devuelve  $z=x$ .
- Construir un programa que visualice por pantalla las tablas de multiplicar del 4 y el 7. El programa debe incluir dos funciones (además de la función `main`). La primera función debe devolver el producto de dos valores numéricos enteros dados como parámetros. La segunda debe visualizar por pantalla la tabla de multiplicar de un número dado como parámetro.
- Indicar la salida por pantalla al ejecutar el siguiente programa:

Ejemplo 8.3: Declaracion y uso de la función sumadigitos

---

```

1  /* sumadigitos.c */
2  #include <stdio.h>
3  #include <stdlib.h>    /* Para poder emplear srand y rand */
4  #include <time.h>       /* Para poder emplear time */
5  int suma(int n) {
```

---

<sup>2</sup>Con el compilador gcc de GNU-Linux hay una excepción, para usar la librería matemática, funciones en `math.h`, se debe indicar explícitamente que se enlace esta librería. En las opciones de compilación hay que añadir `-lm`.

```

6     if (n<10) {
7         return n;
8     }
9     else {
10        return (n % 10 + suma(n/10));
11    }
12 }
13 int main() {
14     int n, i;
15     srand(time(0));
16     for (i=1; i<=10; i++) {
17         n = rand() % 10000;
18         printf("La suma de digitos de %d es: %d\n",n,suma(n));
19     }
20     return 0;
21 }
```

---

7. Construir un programa que incluya la declaración de la función `digito` que devuelve el valor del k-ésimo dígito (tomados de derecha a izquierda y para k>0) del número n. Por ejemplo, la llamada a la función `digito(34567,2)` devolverá el valor 6 y `digito(67,4)` devolverá el valor 0.

## Soluciones a los ejercicios propuestos del capítulo de Funciones

1. La salida por pantalla es:
2. Función que devuelve el cubo de un valor real

Ejemplo 8.4: Funcion cubo de un valor numerico

---

```

1
2 #include <stdio.h>
3
4 float cubo(float a) /* Declaracion */
5 {
6     return a*a*a;
7 }
8
9 int main()
10 {
11     float x,y;
12     x = 2.6;
13     y = cubo(x);          /* Llamada */
14     printf("El cubo de %f es %f\n",x,y);
15     return 0;
16 }
```

---

Salida por pantalla en la ejecución:

El cubo de 2.600000 es 17.575998

3. Programa que convierte un dígito hexadecimal en el valor decimal correspondiente:

Ejemplo 8.5: Pasar de hexadecimal a decimal

---

```

1
2 #include <stdio.h>
3
4 int decimal(char c){
5     if ( (c<='9' && c>='0') || (c>='A' && c<='F') ) {
6         switch (c){
7             case 'A': case 'B': case 'C':
```

```

8             case'D': case'E': case'F':
9                 return c-'A'+10; break;
10            default: return c-'0';
11        }
12    }
13 else
14 {
15     return -1;
16 }
17 }
18
19 int main () {
20     char c;
21     scanf("%c",&c);
22     printf("\n El caracter introducido es: %c",c);
23     printf("\n");
24     printf("Si el caracter introducido no es hexadecimal la solucion sera -1\n");
25     printf(" y si lo es sera el valor correspondiente en decimal\n");
26     printf("Solucion: %d", decimal(c));
27     return 0;
28 }
```

---

Al introducir por teclado en la ejecución:

A

Salida por pantalla en la ejecución:

```

El caracter introducido es: A
Si el caracter introducido no es hexadecimal la solucion sera -1
y si lo es sera el valor correspondiente en decimal
Solucion: 10
```

#### 4. Función dependiente de los cuadrantes

Ejemplo 8.6: Funcion de dependiente de cuadrantes

---

```

1
2 #include <stdio.h>
3
4 float z(float x, float y)
5 {
6     if ( x > 0 && y >= 0 )
7         return x+y;
8     else if ( x <= 0 && y > 0 )
9         return y;
10    else if ( x < 0 && y <= 0 )
11        return 0;
12    else
13        return x;
14 } /* funcion z */
15
16 int main()
17 {
18     float x, y;
19     for (y=-1; y<=1; y=y+1) {
20         for (x=-1; x<=1; x=x+1) {
21             printf("x = %f; y = %f;\n",x,y);
22             printf(" z = %f\n",z(x,y));
23         }
24     }
25     return 0;
26 } /*main*/
```

---

Salida por pantalla en la ejecución:

```
x = -1.000000; y = -1.000000; z = 0.000000
x = 0.000000; y = -1.000000; z = 0.000000
x = 1.000000; y = -1.000000; z = 1.000000
x = -1.000000; y = 0.000000; z = 0.000000
x = 0.000000; y = 0.000000; z = 0.000000
x = 1.000000; y = 0.000000; z = 1.000000
x = -1.000000; y = 1.000000; z = 1.000000
x = 0.000000; y = 1.000000; z = 1.000000
x = 1.000000; y = 1.000000; z = 2.000000
```

5. Programa que visualiza por pantalla las tablas de multiplicar del 4 y del 7:

Ejemplo 8.7: Tablas de multiplicar del 4 y del 7

---

```
1 /* tablas.c */
2 #include <stdio.h>
3 int producto (int a, int b) {
4     return a*b;
5 }
6 void tabla(int n) {
7     int i;
8     printf("Tabla del %d\n",n);
9     for (i=0; i<=10; i++) {
10         printf("%d x %d",n,i);
11         printf(" = %d\n",producto(n,i));
12     }
13     return;
14 }
15 int main() {
16     tabla(4);
17     tabla(7);
18     return 0;
19 }
```

---

Salida por pantalla en la ejecución:

```
Tabla del 4
4 x 0 = 0
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
Tabla del 7
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

6. La salida por pantalla es:

```
La suma de digitos de 3076 es: 16
La suma de digitos de 8105 es: 14
La suma de digitos de 8351 es: 17
La suma de digitos de 5036 es: 14
La suma de digitos de 1851 es: 15
La suma de digitos de 380 es: 11
La suma de digitos de 7607 es: 20
La suma de digitos de 1127 es: 11
La suma de digitos de 9015 es: 15
La suma de digitos de 2103 es: 6
```

## Otros ejercicios propuestos del capítulo de Funciones

1. Construir una rutina que transforma una temperatura dadas en grados Fahrenheit a Celsius
2. Rutina que devuelve el valor central de tres enteros dados como parámetros.
3. Rutina que devuelve verdadero o falso dependiendo de si un carácter es una letra minúscula o no
4. Rutina que devuelve verdadero o falso dependiendo de si una fecha (dando el día, el mes y el año como parámetros) es válida o no
5. Construir una función que devuelva 1 (verdadero) o 0 (falso) dependiendo de si un entero m es divisible por otro n
6. Construir una función que visualice por pantalla y ordenados de menor a mayor los valores de tres valores numéricos dados como parámetros
7. Construir una función que devuelva 1 (verdadero) si un entero es positivo o cero y 0 (falso) si es negativo
8. Construir la función **redondeo** que devuelva el múltiplo de d más próximo por exceso al valor entero n. Los valores n y d son los parámetros de la función y ambos son valores enteros positivos. Por ejemplo: **redondeo(5,2)** devuelve 6, **redondeo(10,4)** devuelve 12 y **redondeo(8,4)** devuelve 8.
9. Construir una función muestre por pantalla la edad de una persona. Los parámetros de la función deben representar dos fechas: una que almacena la fecha actual y otra que almacena la fecha de nacimiento de la persona.
10. Rutina iterativa que devuelve el término enésimo de una progresión aritmética
11. Rutina iterativa que devuelve la suma de los términos de una progresión aritmética
12. Rutina iterativa que devuelve para el término enésimo de una progresión geométrica
13. Rutina iterativa que devuelve la suma de los términos de una progresión geométrica
14. Rutina que devuelve la potencia de una base real elevado a un exponente entero.



# Capítulo 9

## Punteros

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Definir el concepto de puntero o apuntador (Conocimiento)
2. Interpretar el código fuente de un programa que utilice punteros (Comprensión)
3. Escribir la declaración de tipos puntero y de variables de cualquiera de los tipos de datos simples (Aplicación)
4. Interpretar el código fuente de una función con parámetros de tipo puntero (Comprensión)
5. Codificar una tarea sencilla, convenientemente especificada, utilizando punteros (Aplicación)

### 9.1. Variables y direcciones de memoria

Se recuerda que todas las variables de un programa residen en la memoria del ordenador y que ocupan distinto tamaño dependiendo del tipo de dato al que pertenezcan. Se puede imaginar la memoria de un ordenador como una matriz o vector de datos, guardados byte a byte, en binario. A los índices de las componentes de ese supuesto vector se les designa como direcciones de memoria y son, en realidad, números enteros sin signo.

En el ejemplo 9.1 se puede comprobar la organización de las variables del programa en memoria según sus direcciones de memoria. Aunque se verá más adelante detenidamente se emplea el operador *ampersand* (`&`) para obtener la dirección de memoria de una variable.

Nótese que la forma más habitual de mostrar direcciones de memoria es el sistema de numeración hexadecimal. Se puede comprobar el tamaño de las variables declaradas simplemente restando entre sí los números enteros que representan las direcciones de memoria. El número obtenido es el tamaño de memoria en bytes.

Ejemplo 9.1: Distribución de variables en memoria

---

```
1 #include <stdio.h>
2
3
4 int main()
5 {
6     int x; long v; double w;
7     printf("direccion de x es %p \n", &x);
8     printf("direccion de v es %p \n", &v);
9     printf("direccion de w es %p \n", &w);
10    return 0;
11 }
```

---

Salida por pantalla en la ejecución:

```
direccion de x es 0xbff93ca0
direccion de v es 0xbff93ca4
direccion de w es 0xbff93ca8
```

El lenguaje de programación C no solo permite conocer la dirección de una variable declarada en el programa, sino también declarar variables con un tipo específico para almacenar direcciones de memoria. El tipo puntero es el tipo de dato diseñado para contener direcciones de memoria, permitiendo efectuar múltiples operaciones con un significado propio y característico sobre estas direcciones de memoria.

## 9.2. ¿Qué es un puntero?

Así pues, ¿qué es un puntero? La palabra puntero (en inglés *pointer*) en informática es:

El tipo de dato y las variables que se utilizan para guardar direcciones de memoria.

El uso de punteros o direcciones de memoria no tiene sentido por sí mismo. La importancia de los punteros es que sirven de **referencia** para encontrar en memoria otras variables. En informática se dice que un puntero **apunta a una variable** cuando **guarda su dirección de memoria**. Es decir, se puede buscar y utilizar el dato de la variable a partir de un puntero que apunte a la variable. Que el puntero apunta a la variable es equivalente a decir que:

1. el puntero es una referencia de la variable
2. el valor del puntero es la dirección de memoria de la variable
3. la variable está apuntada por el puntero

Los punteros se caracterizan por una codificación, un número natural codificado en memoria en base 2, y un tamaño, que depende del sistema operativo. Para un sistema operativo determinado, todos los punteros tienen el mismo tamaño en memoria: el tamaño adecuado para contener una dirección de memoria. Por ejemplo, en los sistemas Windows de 32 bits el tamaño de los punteros es de 32 bits (4 bytes), lo que permite tener direcciones de memoria para un total de  $2^{32}$  bytes de datos en memoria.

Los punteros en su condición de variables pueden servir para encontrar distintas variables en la memoria simplemente cambiando su valor. Al cambiar el valor del puntero cambia la dirección de memoria y, por tanto, cambia la variable a la que apuntan.

## 9.3. Declaración de punteros

### 9.3.1. Sintaxis

En la declaración de punteros en C se indica el tipo de la variable apuntada y se usa un modificador para indicar que se declara un puntero. Así pues, en la declaración de un puntero primero aparece el tipo de la variable apuntada y luego se aplica el símbolo asterisco (\*) como modificador de la declaración. La sintaxis simplificada<sup>1</sup> es:

```
Basico *id_puntero;
```

Nótese que el asterisco no es ningún operador, ni tiene resultado alguno, ni ninguna relación con la multiplicación de números ni nada de eso. En una declaración, el asterisco es un modificador semántico que se aplica al identificador de la variable para indicar que es un puntero. Este modificador podría haber sido cualquier otro símbolo o palabra, por ejemplo, el acento circunflejo (^) o la palabra *pointer*.

En el ejemplo 9.2 se muestran declaraciones de variables de tipo puntero que apuntan a un tipo básico. Aunque lo correcto es inicializar los valores de estas variables en este caso no se ha hecho, porque son necesarias las operaciones con punteros que se verán más adelante. Naturalmente la compilación del ejemplo muestra dos avisos porque no se usan las variables declaradas.

Ejemplo 9.2: Declaraciones de punteros a tipo básico

---

```

1
2 int main()
3 {
4     int *p_entero;
5     double *p_real;
6     return 0;
7 }
```

---

<sup>1</sup>La sintaxis que se indica solo sirve para declarar punteros que apuntan a variables de tipos básicos.

### 9.3.2. Los punteros y el tipo al que apuntan

La declaración del tipo al que apunta un puntero es muy importante, porque de ese tipo depende el significado concreto o la forma en que se van a realizar algunas de las operaciones posibles sobre punteros, especialmente las operaciones aritméticas sobre punteros: sumas, incrementos, restas, etc.

Al manejar un puntero se trabaja con dos tipos: el tipo puntero propiamente dicho que sirve para guardar una dirección de memoria y el tipo de la variable apuntada. El tipo que aparece en la declaración es, precisamente, este último. Una manera de leer la declaración:

```
int *p;
```

es decir:

`p` es un puntero que apunta a entero (`int`).

### 9.3.3. Punteros genéricos

En C existe un tipo especial de puntero que no apunta a ningún tipo (es decir apunta a `void`). Se dice que son punteros “puros” o “genéricos”. La declaración de un puntero genérico es:

```
void *p;
```

En un puntero genérico no interesa el tipo de la variable a la que apunta el puntero, bien porque no es necesario definirlo (se va a utilizar el valor la dirección de memoria y no se va a utilizar la variable a la que apunta), bien porque no se puede (no se conoce el tipo de la variable a la que apunta).

También se utiliza este tipo de puntero cuando una función es capaz de manejar un puntero a distintos tipos dependiendo de las circunstancias concretas. En estos casos se aprovecha la característica de que todos los tipos punteros se pueden convertir a punteros genéricos y viceversa, aunque no siempre de forma automática o implícita. Es decir, las operaciones del ejemplo 9.3 son correctas.

Ejemplo 9.3: Declaración y conversión de punteros genéricos

---

```

1
2 int main()
3 {
4     int *p_entero;
5     void *p_void;
6
7     /* Se omiten las sentencias necesarias para
8      * para dar valores a los punteros */
9
10    /* Conversiones implícitas */
11    p_entero = p_void;
12    p_void = p_entero;
13
14    /* Conversiones explícitas */
15    p_entero = (int*) p_void;
16    p_void = (void*) p_entero;
17
18    return 0;
19 }
```

---

## 9.4. Operaciones con punteros

Dado que los datos de los punteros son direcciones de memoria las operaciones habituales aplicadas a punteros, por ejemplo, las operaciones aritméticas, pueden tener un significado especial. También existen operaciones que son específicas de punteros.

### 9.4.1. Operador dirección de

El operador unario *dirección de* (en inglés, *address-of operator*) devuelve la posición en memoria de una variable operando situada a la derecha. El símbolo de la operación es la abreviatura de la conjunción y (*and*) en inglés, el símbolo llamado *ampersand* (&).

Este operador se aplica normalmente a una variable para obtener su dirección de memoria (donde se guarda la variable) y poder almacenar el resultado en otra variable puntero.

### 9.4.2. Operador indirección

El operador unario *indirección*, también llamado operador contenido, (en inglés, *indirection operator*) es la operación contraria a la anterior.

El operador indirección determina la variable referida por el puntero (en inglés, *dereferences a pointer*). También podemos decir que determina la variable a la que apunta el puntero o la variable situada en la posición dada por la dirección de memoria del puntero. El tipo de la variable resultado viene dado por el tipo al que apunta el puntero.

El símbolo del operador *indirección* es, de nuevo, el asterisco (\*). Es muy importante diferenciar los distintos usos del asterisco. Esta *sobrecarga*<sup>2</sup> de uso se distingue por el contexto. En este caso el operador *indirección* se diferencia sabiendo que es un operador unario por la izquierda que se aplica obligatoriamente a una variable puntero o un valor de tipo puntero.

El resultado de una operación de indirección se puede usar de dos formas:

1. En el lado izquierdo de una **asignación**, dado que esta operación determina una variable.
2. En cualquier otro contexto se usa el valor almacenado en la variable apuntada.

En el ejemplo 9.4 se muestra el uso de los operadores asociados a punteros. Se volverá sobre ellos en el capítulo correspondiente. En este ejemplo conviene comprobar que la diferencia entre las direcciones de memoria de las variables es, precisamente, el tamaño de tipo **short**. Asimismo se puede observar que el resultado del operador indirección es equivalente a una *variable* y como tal se puede usar en el lado izquierdo de una asignación para cambiar el valor de la variable a la que apunta el puntero.

Ejemplo 9.4: Operadores de punteros

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     short a = -359, b = 64;
7     short *p = &a;
8
9     printf("address of a: %p \n", &a);
10    printf("address of b: %p \n", &b);
11
12    *p = 1;
13    printf("*p = %d, a = %d \n", *p, a);
14
15    return 0;
16 }
```

---

Salida por pantalla en la ejecución:

```

address of a: 0xbfa8d9d8
address of b: 0xbfa8d9da
*p = 1, a = 1
```

### 9.4.3. Operaciones de relación e igualdad

Se pueden aplicar a variables de tipo puntero los operadores de relación habituales: igualdad (==), o desigualdad (!=) con el significado habitual matizado por el hecho de ser punteros.

---

<sup>2</sup>se dice que una función u operador está sobrecargado cuando el mismo símbolo tiene varias definiciones.

El resultado del operador de igualdad es 1 (tipo `int`) cuando ambos operandos son iguales y 0 (tipo `int`) cuando son distintos, recíprocamente para el operador de desigualdad.

La igualdad de punteros se usa:

1. Para determinar si ambos punteros apuntan a la misma variable. Si dos punteros tienen el mismo valor, es decir, la misma dirección de memoria, apuntan a la misma variable o también el resultado de su operación de indirección es el mismo.
2. Para determinar si un puntero apunta a alguna variable. Esta comparación se realiza utilizando como operando la constante `NULL` (ver apartado siguiente). Cuando un puntero tiene el valor `NULL` se puede asegurar que no tiene la dirección de memoria de ninguna variable o dato.

## 9.5. El valor nulo de direcciones de memoria: `NULL`

Los punteros, en su calidad de variables que almacenan direcciones de memoria, pueden llegar a guardar cualquier direcciones de memoria dentro de su rango de representación (como cualquier otra variable). Ahora bien, al manejar variables puntero es especialmente importante que los valores almacenados sean válidos, es decir, representen direcciones de memoria de datos del programa. De no ser así se pueden provocar errores de funcionamiento grave.

Entonces, ¿qué se puede hacer con un puntero que no se desea que apunte a ningún dato? Los motivos para tener un puntero que no apunte a un dato pueden ser muy distintos: (a) se puede marcar de esta manera que hay que reservar una nueva variable dinámica, (b) puede ser la marca del final de una estructura dinámica o (c) sencillamente se va a utilizar el puntero más tarde.

En todos estos casos el problema se resuelve definiendo un valor de dirección de memoria nula, es decir, un valor que no corresponde con ninguna dirección de memoria o dicho de otra manera una dirección de memoria que ningún programa usa para guardar datos, sean de tipo que sean.

Este valor es `NULL`. Según la especificación del estándar, `NULL` es una macro que se define en `<stddef.h>`, aunque también puede estar definida en otros archivos de cabecera. De hecho, no es frecuente tener que incluir este archivo sólo por esta macro. La macro `NULL` se corresponde con el concepto de dirección de memoria nula tal y como se ha descrito en el párrafo anterior. Su valor puede variar según el compilador, aunque normalmente es el valor cero numérico (0).

Normalmente la macro `NULL` se usa como valor de inicialización de punteros (para indicar que no apuntan a nada), en asignaciones de punteros (para lo mismo) y en expresiones de relación (igualdad, desigualdad) con punteros para comprobar si apuntan a un dato.

## 9.6. Uso de punteros en C

En C, las variables de tipo puntero se usan o encuentran en las siguientes situaciones:

1. Como una referencia temporal o redundante de variables estáticas cuya finalidad es, normalmente, el paso de un parámetro de entrada - salida.
2. Como referencia de variables dinámicas tal y como se verá en el capítulo correspondiente.
3. Como referencia a los elementos de un vector (*array*) sea estático o dinámico, especialmente al primer elemento y, en usos particulares, a otros elementos del *array*. Incluido el caso particular de cadenas alfanuméricas. Normalmente, en el uso de punteros con *arrays* se emplean los operadores de aritmética de punteros. Todo ello se verá en el capítulo de vectores y matrices.
4. Como referencia a funciones. Se verá en el capítulo correspondiente dentro del material de consulta.

Aunque existe algún otro uso, es pertinente destacar que el uso de punteros en C aumenta el nivel de dificultad de escribir programas correctamente y, por tanto, se debe restringir a aquellos casos en que está justificado. En la mayoría de las ocasiones se justifica el uso de punteros por encontrarse en alguno de los casos mencionados anteriormente.

### 9.6.1. Referencias de variables estáticas

El uso de referencias a variables estáticas no está justificado en la mayoría de los casos. El identificador de las variables es una referencia segura y clara de los datos que se utilizan en el programa y, por tanto, el uso de referencias adicionales no hace sino complicar las cosas.

Existe la excepción del uso de punteros para paso de parámetros a funciones. A continuación se muestra cómo se utilizan los punteros en este caso a través de distintos ejemplos que presentan el funcionamiento de las referencias a variables estáticas.

En el primer ejemplo 9.5, se puede comprobar el efecto de asignar una variable a otra. Cuando posteriormente se modifica el valor de cualquiera de ellas el valor de la otra permanece inalterado. Sin embargo si se asigna a un puntero la dirección de la variable, cuando se cambie el valor de la variable referencia por el puntero, sí que cambia el valor de la variable. Esta es la razón que desaconseja el uso de punteros tal y como se muestra en este ejemplo, porque se pierde claridad, cuando se modifica una variable normalmente no se espera que otra también sufra esa modificación.

---

Ejemplo 9.5: Puntero que apunta a variable estática

---

```

1
2 #include "stdio.h"
3
4 int main()
5 {
6     /* variables estaticas */
7     int x = 4, y;
8
9     /* un puntero */
10    int *p;
11
12    /* asignacion entre variables */
13    y = x;
14
15    /* al cambiar y, obviamente x no cambia */
16    y = 5;
17
18    printf(" x : %d \t y : %d \n", x, y);
19
20    /* asignacion con un puntero */
21    p = & x;
22
23    /* al cambiar *p, no tan obviamente, x SI cambia */
24    *p = 5;
25
26    printf(" x: %d \t *p : %d \n", x, *p);
27
28    return 0;
29 }/* main */

```

---

Salida del programa:

```
x : 4    y : 5
x: 5    *p : 5
```

Ahora bien, en el caso de paso de parámetros aparece un caso en que sí se puede desechar que se modifique el valor de una variable. Como se ha indicado en el capítulo de rutinas, el paso de parámetros a una función se hace mediante un mecanismo de copia que se denomina *paso por valor*. Este mecanismo no permite que el parámetro sufra modificación alguna dentro de la rutina, incluso si se desea hacerlo de esa manera.

Se considera el caso en que se necesita incrementar el valor de una variable en uno, es decir, el caso de un parámetro de entrada - salida. Al intentar realizar esta función utilizando un parámetro numérico surge el problema que se ilustra en el ejemplo 9.6, donde se observa que el valor de la variable no cambia.

---

Ejemplo 9.6: Paso de parámetro por valor

---

```

1
2 #include "stdio.h"
3
4 void incremento(int x)
5 {
6     x++;
7 } /* incremento */
8

```

```

9 int main()
10 {
11     int y = 4;
12
13     /* el paso por valor equivale a x = y */
14     incremento(y);
15
16     printf(" y : %d \n", y);
17
18     return 0;
19 }/* main */

```

---

Salida del programa:

y : 4

El origen del problema es muy sencillo, basta mirar en la asignación de variables que se encuentran en el ejemplo 9.5 donde al cambiar una variable no se toca el valor de otra que se haya asignado previamente.

Es importante observar qué ocurre cuando el paso de parámetro es mediante una referencia, es decir, un puntero. Al analizar la salida por pantalla del ejemplo 9.7 se puede comprobar que sí se ha modificado el valor de la variable. La razón de nuevo es sencilla si se piensa qué significa tener una copia de una referencia.

Se puede comprender este concepto a través de un ejemplo con un número de teléfono: se supone que Luis tiene como número de teléfono el 91 869 36 58. Es decir, este número es una referencia de Luis. Por otro lado Juan ha guardado una copia de ese teléfono en su móvil. A su vez, Juan quiere que Luis se entere de la próxima reunión de amigos pero no le puede llamar el mismo porque se ha quedado sin batería. Por este motivo le pasa el número a Jorge (es decir, Jorge recibe una copia de la referencia - paso por valor) y éste llama a Luis. Naturalmente, Luis se entera, es decir, si Luis fuese una variable modificaría su valor.

Como se puede comprobar a través del programa o del ejemplo, dos copias de una referencia comparten una única cosa referida. Las dos copias del número de Luis, una en el móvil de Jorge y otra en el de Juan, sirven para llamar al mismo Luis. En variables, un puntero que es una copia (paso de parámetro por valor) del puntero de una variable, apunta a la misma variable y por tanto la pueden modificar.

#### Ejemplo 9.7: Paso de un puntero por valor

---

```

1
2 #include "stdio.h"
3
4 void incremento(int* x)
5 {
6     (*x)++;
7 }/* incremento */
8
9 int main()
10 {
11     int y = 4;
12
13     /* el paso por valor equivale a x = y */
14     incremento(&y);
15
16     printf(" y : %d \n", y);
17
18     return 0;
19 }/* main */

```

---

Salida del programa:

y : 5

Una vez comprendido el mecanismo de paso de punteros como parámetro para la implementación de parámetros de entrada - salida o de salida, es conveniente tomar nota para la programación de estos casos de las siguientes ideas fundamentales:

1. Cuando una función tiene como parámetro un puntero el objetivo puede ser, entre otras cosas, el paso de una variable como entrada - salida o como salida.

2. Cuando se quiere pasar una variable como entrada - salida o como salida y se usa el puntero correspondiente como parámetro, **no** es necesario declarar ningún puntero, en cambio se recomienda pasar directamente la dirección de la variable usando el operador *address-of* (<&>).
3. Si no se pasa la variable a través de su dirección de memoria la función llamada puede producir todo tipo de errores graves de acceso a memoria.

### 9.6.2. Parámetros formales por referencia

Un parámetro formal por referencia es un parámetro formal de tipo puntero que recibe como argumento la dirección de memoria de una variable donde se guarda un dato que se quiere modificar (se pasa como entrada - salida).

La variable o dato que se quiere pasar no se instancia, es decir, no se guarda una nueva variable con el valor sino un nuevo puntero a ese valor. Por otro lado, esto implica que el dato cuya dirección de memoria se pasa debe existir previamente.

Es decir, el puntero, parámetro formal apunta a una zona de memoria donde se almacena la variable que contiene el dato que la función va a utilizar. Normalmente este dato es una variable declarada en la función que realiza la llamada. Como consecuencia, la variable declarada y el resultado de hacer la indirección del puntero son lo mismo.

En la definición de la función el parámetro formal por referencia debe ser de tipo puntero que apunte a un tipo de dato apropiado. El correspondiente argumento debe ser la dirección de la variable que almacene el valor que se desea pasar a la función ejecutada.

Un ejemplo de este tipo de parámetros viene ilustrado por el siguiente programa, el programa es similar al anterior pero se han modificado la definición de la función y la llamada a la misma.

El programa utiliza una función **suma1bis** que añade 3 unidades al contenido apuntado por el puntero **x**, parámetro formal de la función e imprime por pantalla su nuevo valor.

En el programa principal se definen una variable **a**, **a** toma el valor de 5. La dirección de la variable **a**, esto es, **&a** es el argumento en la llamada a la función **suma1bis**.

La función **suma1bis** tiene un parámetro, **x**. Cuando se ejecuta la función en la sentencia **suma1bis(&a)**, en la zona de la memoria reservada para **x** (puntero a entero) se copia el valor que tiene **&a**. De esta forma todas las instrucciones de la función **suma1bis** pueden ejecutarse correctamente puesto que el parámetro **x** apunta a un dato concreto.

Las últimas instrucciones visualizan el contenido de la variable **a**. Como se puede observar se ha modificado el valor que se le asignó en la primera instrucción del programa.

Ejemplo 9.8: Uso de parámetros por referencia

---

```

1
2 #include <stdio.h>
3
4 void suma1bis(int *x)
5 {
6     *x = *x + 3;
7     printf("El valor apuntado por x es %d",*x);
8 }/*suma1bis*/
9
10 int main()
11 {
12     int a = 5;
13     suma1bis(&a);
14
15     printf("\nEl valor de a es %d",a);
16
17     return 0;
18 }/*main*/

```

---

Salida del programa:

El valor apuntado por x es 8  
El valor de a es 8

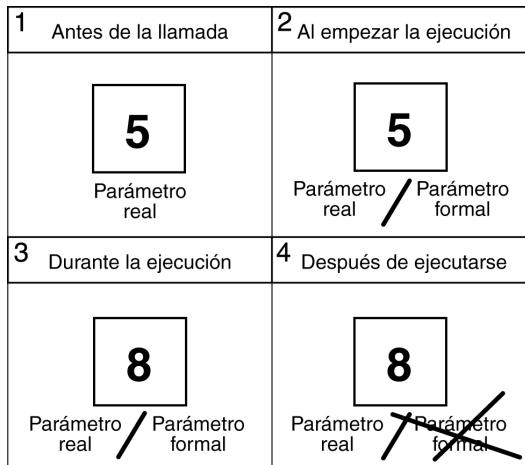


Figura 9.1: Paso de parámetro por referencia

En la figura 9.1 se puede observar los cambios en memoria en el paso de parámetros por referencia. En este caso existe un único recuadro marcado como parámetro formal o real a lo largo del proceso. El parámetro real nuevamente se corresponde con la variable `a`. Como se ha visto pasamos un puntero o una referencia de esta variable no la propia variable. Este puntero no aparece en la figura. Por otro lado, la marca parámetro formal se corresponde con el resultado de aplicar el operador indirección al puntero, es decir, `*x`, la variable apuntada por la referencia o puntero que hemos pasado.

## Ejercicios resueltos del capítulo de Punteros

- Indicar la salida por pantalla al ejecutar el siguiente programa:

Ejemplo 9.9: Ejemplo de uso de punteros

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     float x;
7     float *p, *q;
8
9     p = NULL; x = 12.4; q = &x;
10
11    if (p == NULL)
12    {
13        printf("No apunta a una variable\n");
14    }
15
16    if (p == q)
17    {
18        printf("Apuntan al mismo dato\n");
19    }
20    else
21    {
22        printf("No apuntan al mismo dato\n");
23    }
24    printf("p = %p\n", p);
25    printf("q = %p\n", q);
26
27 }
```

---

- Indicar la salida por pantalla al ejecutar el siguiente programa:

---

Ejemplo 9.10: Ejemplo de uso de parametros por referencia

---

```

1 #include <stdio.h>
2
3 void modifica(int *p, int *q)
4 {
5     *p = 3;
6     *q = 4;
7     return;
8 }
9
10
11 int main()
12 {
13     int m = 1, n = 2;
14     printf("m vale %d; n vale %d\n", m, n);
15     modifica(&m, &n);
16     printf("m vale %d; n vale %d\n", m, n);
17     return 0;
18 }
```

---

3. Construir una función que permita incrementar el valor de un parámetro real (en la llamada) con el valor de un segundo parámetro. Nota: se deben emplear un parámetro formal por referencia y otro por valor.
4. Indicar la salida por pantalla al ejecutar el siguiente programa si se introduce por teclado un valor de 3 para el radio y de 2 para la altura del cilindro:

---

Ejemplo 9.11: Ejemplo de uso combinado de retorno de función y de parametro por referencia

---

```

1
2 #include <stdio.h>
3
4 float cilindro (float radio, float altura, float *volumen)
5 {
6     float area;
7     float PI=3.14;
8     area = 2*radio*altura*PI;
9     *volumen = PI*radio*radio*altura;
10    return area;
11 }
12
13 int main ()
14 {
15     float r, h, s, v;
16     printf("Introduce el valor del radio: ");
17     scanf("%f", &r);
18     printf("\nIntroduce el valor de la altura: ");
19     scanf("%f", &h);
20     printf("\nEl radio es %f y la altura es %f \n", r, h);
21     s = cilindro(r, h, &v);
22     printf("El area es %f y el volumen es %f \n", s, v);
23     return 0;
24 }
```

---

5. Construir una función **maximo** que devuelva la dirección de memoria de la variable numérica real (**double**) que almacene el mayor valor de las dos variables cuyas respectivas direcciones se den como parámetros en la llamada.

## Soluciones a los ejercicios del capítulo de Punteros

1. La salida por pantalla es:

```
No apunta a una variable
No apuntan al mismo dato
p = (nil)
q = 0xbff29bb4
```

2. La salida por pantalla es:

```
m vale 1; n vale 2
m vale 3; n vale 4
```

3. Función que permite incrementar el valor de un parámetro real (en la llamada) con el valor de un segundo parámetro.

---

Ejemplo 9.12: Funcion que incrementa el valor de un parametro

---

```
1
2 #include <stdio.h>
3
4 void incrementa(int *p, int q)
5 {
6     *p += q;
7 }
8
9 int main()
10 {
11     int m = 1, n = 2;
12     printf("m vale %d; n vale %d\n", m, n);
13     incrementa(&m, n);
14     printf("m vale %d; n vale %d\n", m, n);
15     return 0;
16 }
```

---

Salida por pantalla en la ejecución:

```
m vale 1; n vale 2
m vale 3; n vale 2
```

4. La salida por pantalla es:

```
Introduce el valor del radio:
Introduce el valor de la altura:
El radio es 3.000000 y la altura es 2.000000
El area es 37.680000 y el volumen es 56.520000
```

5. Función **maximo** que devuelve la dirección de memoria de la variable numérica real (**double**) que almacena el mayor valor de las dos variables cuyas respectivas direcciones se dan como parámetros en la llamada.

---

Ejemplo 9.13: Funcion que devuelve una direccion de memoria

---

```
1
2 #include <stdio.h>
3
4 double* maximo(double *a, double *b)
5 {
6     if (*a > *b)
7     {
8         return a;
9     }
10    else
11    {
12        return b;
13    }
14 }
```

```

14 }
15
16 int main()
17 {
18     double x = 2.49, y = 3.75;
19     double *p, *q, *r;
20
21     p = &x; q = &y;
22     printf("p vale %p; *p vale %f\n", p, *p);
23     printf("q vale %p; *q vale %f\n", q, *q);
24     r = maximo(p, q);
25     printf("r vale %p; *r vale %f\n", r, *r);
26     return 0;
27 }
```

---

Salida por pantalla en la ejecución:

```
p vale 0xbfaac820; *p vale 2.490000
q vale 0xbfaac828; *q vale 3.750000
r vale 0xbfaac828; *r vale 3.750000
```

## Ejercicios propuestos del capítulo de Punteros

1. Construir una rutina que intercambie el valor de dos variables de tipo `double`.
2. Escribe, compila y ejecuta el siguiente programa e interpreta el error que produce:

```

void par_por_referencia(int *a) {
    *a = 10;
}
int main() {
    int *p = NULL;
    par_por_referencia(p);
    return 0;
}
```

3. Construir un programa que muestre el tamaño en memoria de punteros a distintos tipos de dato.
4. Mostrar por pantalla la dirección de memoria de varias variables dinámicas que se crean en un programa. Muestre por pantalla el tamaño de las misma variables utilizando el operador `sizeof`. Inicialice el valor de las variables. Por último, libere la reserva de memoria de las variables.
5. Escriba un programa para comprender el efecto del operador suma en punteros. Declare dos punteros que apunten a tipos básicos diferentes, inicialice el valor de los punteros a la misma dirección y luego muestre por pantalla la suma del puntero más el mismo un entero.
6. Construir una función `incremento` para que dada una hora, expresada en horas y minutos, devuelva la hora incrementada en x minutos. Los parámetros formales de la función son tres: la hora inicial, el minuto inicial y el incremento de minutos. Pista: Los dos primeros parámetros formales son por referencia (entrada y salida) y el tercero es por valor (entrada). Por ejemplo, si la hora y minuto inicial es 23:57 y x vale 15, entonces la hora y minuto final deben tomar el valor 0:12. Nota: los intervalos válidos para representar las horas y los minutos son 0..23 y 0..59 respectivamente. El incremento de minutos sólo está acotado por el intervalo de representación del tipo de dato correspondiente.

# Capítulo 10

## Vectores y matrices

La programación en el lenguaje C tiene varios tipos de datos estructurados, algunos de los cuales se cubrirán en este capítulo.

Objetivos:

1. Describir los tipos de dato vector y matriz, su estructura y forma de uso (Conocimiento)
2. Interpretar el código fuente de un programa en C donde aparezcan vectores (Comprensión)
3. Codificar una tarea, convenientemente especificada, utilizando vectores y matrices (Aplicación)

### 10.1. Definición de vector

Un *vector* es, en realidad, un conjunto, estructura o secuencia de datos que agrupa una colección de datos del mismo tipo. Pueden ser unidimensionales, denominados también vectores, o multidimensionales, denominados matrices o tablas. En muchos países de Hispanoamérica se emplea habitualmente el término *arreglo* para denominar a este tipo de estructuras.

Cada variable individual, componente o *elemento* del vector se identifica mediante un número entero que se denomina *índice*. Se puede acceder a cualquier elemento de la estructura utilizando el índice entre corchetes.

Dado que el vector en informática es simplemente una estructura secuencial de datos, es decir, una manera de guardar en conjunto varios datos del mismo tipo, no se presupone a los vectores ninguna propiedad algebraica. Es decir, no se pueden sumar ni multiplicar por escalares ni tienen operaciones especiales salvo las que se van a indicar de manera expresa.

### 10.2. Declaración de vectores

La sintaxis simplificada de la declaración de un vector de datos de tipo básico (enteros y números en coma flotante) es:

```
#define DIM 7  
tipoBasico id_vector[DIM];
```

donde:

- **tipoBasico** es un tipo entre los dados en el capítulo correspondiente por ejemplo: **int**, **float**, **char**, etc. Es el tipo de los elementos (o componentes) que constituyen el vector.
- **id\_vector** es el identificador de la variable de tipo vector de datos básicos.
- **DIM** es una constante literal de tipo entero y determina el número de elementos (componentes) que constituyen el vector. Para no poner un número concreto se ha utilizado el comando del preprocesador **#define**.<sup>1</sup>

<sup>1</sup>En C99 si es posible utilizar VLA (Variable Length Arrays), es decir, utilizar una variable para indicar el tamaño de un vector. No así en ANSI C. En C++ se debe evitar el uso de macros para el tamaño de vectores, en su lugar es aconsejable el uso de **enum**.

La declaración de una variable de tipo vector de datos de tipo básico implica la reserva inmediata del espacio en memoria necesarios para guardar todos los datos individuales del tipo básico, elementos o componentes que forman el vector. En cierta manera equivale a la declaración de DIM variables de tipo `tipoBasico`. El tamaño de un vector, `sizeof(id_vector)`, en memoria es: `DIM * sizeof(tipoBasico)`.

Es posible inicializar vectores en la propia declaración. En este caso se puede omitir el tamaño del vector deducible a partir del número de elementos que se indican. La sintaxis simplificada es:

```
tipoBasico id_vector[DIM_opt] = { valor_1, valor_2, valor_DIM };
```

donde dentro de las llaves se escriben los valores iniciales de las componentes del vector separadas por comas. Es posible inicializar menos componentes que la dimensión del vector. En este caso los últimos elementos se inicializan a cero <sup>2</sup>. Por supuesto no se pueden inicializar más componentes que las que determina la dimensión del vector. Si se omite la dimensión el vector tendrá exactamente la dimensión igual al número de valores suministrados.

### 10.3. Limitaciones de los vectores

Para cualquier declaración, la variable `id_vector` es del tipo compuesto vector. En los lenguajes de programación de propósito general los vectores son simplemente una estructura de datos sin operaciones propias. En un contexto matemático los vectores tienen una serie de operaciones definidas (espacios vectoriales). Esto no ocurre así en el ámbito de los lenguajes de programación. De hecho, en el lenguaje C las variables de tipo vector **solo** se pueden utilizar de **dos** maneras:

1. Con el operador indexación, representado por apertura y cierre de corchetes, [ ].
2. Mediante conversión implícita a un puntero que apunta al primer elemento del vector.

En concreto esta circunstancia tiene como consecuencia que las variables de tipo vector, en general, solo se encuentran en expresiones seguidas de los correspondientes corchetes. Se destaca especialmente que los vectores **no se pueden asignar ni comparar** aunque aparentemente el compilador lo admite. En realidad el compilador está realizando la conversión implícita a puntero y realizando la asignación o comparación con el puntero. Si esto no es lo que se pretende se pueden producir errores muy graves.

### 10.4. Estructura de los vectores

Los vectores en C son una estructura que agrupa varias variables del mismo tipo de forma secuencial en memoria. Esto significa que las variables individuales que constituyen el vector se encuentran seguidas, sin huecos, desde la primera a la última. Como las variables se identifican con una serie de índices que empiezan en cero (0) y cada una de ellas ocupa exactamente el mismo tamaño dado por su tipo, se puede encontrar cualquiera de las variables individuales simplemente desplazándose por la memoria del ordenador un número de bytes que se puede calcular como el índice de la variable que se busca multiplicado por el tamaño del tipo de las variables individuales. Por supuesto, para encontrar el primer elemento del vector no es necesario desplazarse (teniendo en cuenta que el índice de ese primer elemento es siempre 0). En la figura 10.1 se muestra un diagrama que ilustra la estructura del vector.

---

<sup>2</sup>En cualquier caso, para evitar errores imprevistos, se recomienda realizar una inicialización explícita, es decir, realizar la inicialización a cero de todas las componentes del vector a pesar de no ser necesario.

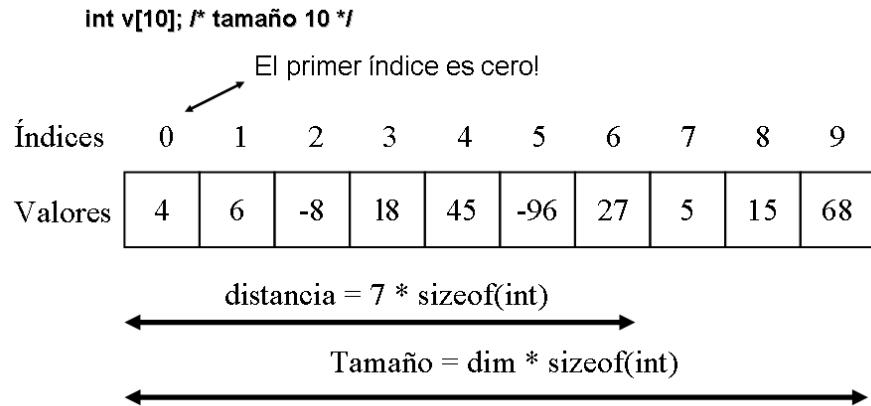


Figura 10.1: Estructura de un vector

## 10.5. El operador indexación

La única operación que se puede realizar con un vector, tal y como se ha dicho, es aplicar un operador indexación. La sintaxis simplificada del operador es:

```
id_vector[exp_entera]
```

donde **id\_vector** es el identificador de la variable de tipo vector y **exp\_entera** es una expresión que tiene como resultado un número entero. Ambos constituyen los operandos de la operación indexación. El resultado de la operación se puede usar de dos maneras:

1. Para determinar una de las variables elementales que forman el vector y esta variable usarla en el **lado izquierdo** de una **asignación** o como operando de una operación *dirección de*.
2. Para obtener el valor que se guarda en una de la variables elementales y usar dicho valor en el proceso de evaluación de una expresión más grande.

Una característica y limitación importante del operador indexación de C es:

El operador `[ ]` no comprueba si se alcanza el límite del vector.

En inglés esta característica se suele denominar *check bounds* (comprobación de límites). La expresión *out of bounds* (fuera de límites) describe el tipo de error que se puede producir. Algunos lenguajes sí generan errores al realizar esta comprobación (se suelen indicar como *out of bounds exceptions*).

Teóricamente esta característica hace que el acceso a una variable individual dentro de un vector sea más rápido en C que en otros lenguajes. Naturalmente a costa de que se puedan producir errores si se realiza el acceso o modificación de un dato de un elemento del vector que no existe como tal. Este matiz es importante, por la propia forma en la que se gestionan los vectores en C *todos* los elementos de un vector *existen*, es decir, en el desplazamiento por memoria que se produce para localizar el dato del vector llegamos a algún punto de la memoria donde, sencillamente, puede estar guardadas otras variables que nada tienen que ver con el vector.

## 10.6. Vectores y bucles

Gran cantidad de problemas informáticos se resuelven con la combinación de vectores como estructuras de datos y bucles como base de desarrollo de algoritmos. Esta combinación se produce de forma muy natural mediante la relación de los índices de los vectores y los contadores de los bucles **for**.

Dentro de estos bucles se encuentran dos categorías elementales de bucles:

1. Aquellos que aplican alguna manipulación a todos los elementos de un vector, de uno en uno y de forma independiente. Por ejemplo, mostrarlos por pantalla.

2. Aquellos que realizan una operación agregada sobre todos los elementos de un vector. Esta operación se realiza elemento a elemento, pero todos influyen en el resultado. Por ejemplo, la suma o producto de todos los elementos de un vector, su media, el máximo o mínimo de todos ellos, etc.

En cualquiera de estos casos suele aparecer un bucle que se repite en muchos ejemplos y que sirve de base para estos problemas y otros relacionados:

```
int i;
for ( i = 0; i < DIM; ++i ) {
    /* Sentencias con id_vector[i] */
}
```

En problema con bucles donde se acceda a expresiones basadas en el índice del bucle, por ejemplo: `id_vector[i+1]` es **muy importante** comprobar que el resultado de la expresión en ningún caso queda fuera del rango de índices del vector. Por ejemplo:

```
int i;
for ( i = 0; i < DIM-1; ++i ) {
    /* Sentencias con id_vector[i] y con id_vector[i+1] */
}
```

## 10.7. Matrices

En C las matrices son vectores de vectores. Dicho de otra manera, una matriz es un vector cuyos elementos o componentes son a su vez vectores. En este sentido, al igual que los vectores las matrices no tienen operaciones propias salvo el operador indexación. Para matrices de orden superior (más de 2 índices), la idea es la misma aumentando el nivel de vectores implicados.

La estructura de las matrices de C en memoria es “por filas”. Con más detalle, cuando se guarda una matriz  $M \times N$ , donde  $M$  es el número de filas y  $N$  el número de columnas, en memoria, se guarda primero los valores de la primera fila, es decir,  $N$  elementos o componentes, después los de la segunda fila de nuevo  $N$  elementos y así hasta las  $M$  filas de la matriz. Como en el caso de los vectores todos los elementos de una matriz aparecen seguidos en memoria.

La declaración de una matriz  $M \times N$  queda:

```
#define M 3
#define N 3
tipoBasico id_matriz[M][N];
```

donde `tipoBasico` es el tipo de las componentes o elementos de la matriz y un tipo básico (entero o real), `id_matriz` es el identificador de la variable, `M` es el número de filas y `N` el número de columnas. Naturalmente `N` es el número de componentes que tiene una fila y `M` el número de componentes de una columna. Tanto `M` como `N` tienen que ser constantes literales, para evitar poner números concretos se puede utilizar el comando del preprocesador `#define`.

Al aplicar una vez el operador indexación a la matriz se obtiene una fila de la matriz, cuando se aplican dos operadores indexación se obtienen los elementos individuales. El primer índice es el de la fila y el segundo el de la columna. Obviamente el límite de índices para el primer operador indexación es  $M-1$  y para el segundo  $N-1$ . De nuevo C no comprueba estos límites en la ejecución del programa.

Se pueden inicializar los valores de una matriz utilizando 2 niveles de llaves para indicar los valores por filas. De esta manera:

```
tipoBasico id_matriz[M][N] =
{ { a11, a12, a1N }, { a21, a22, a2N }, { aM1, aM2, aMN } };
```

donde en la declaración las palabras tienen el mismo significado que en el caso anterior y entre llaves aparecen los valores constantes representados por `a11, a12, ...` que se guardan en la matriz al inicializarlas. El orden en que aparecen estos valores en el código fuente es precisamente el mismo que el que se usará para guardarlos en memoria.

Normalmente las matrices se manipulan mediante bucles anidados. Por ejemplo:

```
int i,j;
for ( i = 0; i < M; ++i ) {
```

```

for ( j = 0; j < N; ++j ) {
    /* Sentencias con id_matriz[i][j] */
}/* for j*/
}/*for i*/

```

## 10.8. Relación entre punteros y vectores

Una de las causas de errores y confusiones más importante entre los programadores en lenguaje C es la correspondencia que existe entre los vectores y los punteros. En C, hasta cierto punto, el tipo vector y el tipo puntero son intercambiables y el compilador los intercambia implícitamente en casos concretos. En este apartado se van a explicar los casos de uso en los que se combinan vectores y punteros.

### 10.8.1. Operador [ ]

En primer lugar se va a establecer el significado del operador corchete aplicado a punteros. Normalmente al aplicar el operador [ ] a un vector o matriz se obtiene el elemento dado por el índice indicado dentro del corchete. Pues bien, al aplicar el operador [ ] a un puntero que apunta al primer elemento del vector y el tipo del vector es que apunta al tipo básico del vector, el resultado es equivalente.

Es decir, el operador [ ] aplicado a un puntero realiza dos operaciones:

1. por un lado realiza una suma del puntero con el entero dentro de los corchetes
2. y después aplica el operador indirección al resultado.

En consecuencia se obtiene la variable de la componente del vector que corresponde con el índice.

### 10.8.2. Aritmética de punteros

Por la propia naturaleza la aplicación de aritmética de punteros debe restringirse a punteros que apunten a variables de tipo *array*. Por otro lado, los casos de aplicación de la aritmética de punteros normalmente coinciden con los casos de uso del operador  $\square$  sobre punteros y este último resulta más sencillo.

Los punteros en su condición de direcciones de memoria no tienen ningún tipo de operación aritmética cuyo significado coincide con el usual de la aritmética sobre los conjuntos de números habituales (enteros, reales, etc.).

No obstante si se define una aritmética sobre punteros con operaciones suma y resta cuyo significado es el desplazamiento izquierda y derecha de la dirección de memoria (y no adición o sustracción como sobre los números).

En este sentido se destaca que las operaciones aritméticas de punteros no son internas. Por ejemplo, la suma, o sea, el desplazamiento hacia arriba (o derecha) obligatoriamente tiene un puntero como operando izquierdo y un número entero como operando derecho, es decir, no es una operación interna. Sin embargo, la resta de punteros puede operar dos punteros pero, en ese caso, devuelve un número entero (no otro puntero) o un puntero como operando a la izquierda y un número como operando a la derecha y en este caso significa desplazamiento abajo (o izquierda) y devuelve un puntero.

Cuadro 10.1: Aritmética de punteros

Símbolo	Significado	Operando Izquierda	Operando Derecha	Tipo Resultado	Ejemplo	Fórmula
+	Desplazamiento hacia arriba o derecha. Es decir, el valor de la dirección de memoria aumenta.	Un puntero	Un número entero	Un puntero	$p2 = p1 + 3;$	$p2 = p1 + 3 * sizeof(tipo)$
-	Desplazamiento hacia abajo o izquierda. Es decir, el valor de la dirección de memoria disminuye.	Un puntero	Un número entero	Un puntero	$p2 = p1 - 3;$	$p2 = p1 - 3 * sizeof(tipo)$
-	Diferencia de posiciones en unidades de memoria del tipo referenciado	Un puntero	Un puntero	Un entero	$dif = p2 - p1$	$dif = (p2 - p1) / sizeof(tipo)$

Dados dos punteros y una variable entera declarados como sigue:

```
tipo *p1, *p2; int dif;
```

Donde **tipo** es el identificador de cualquier tipo de dato válido, con las posibles operaciones que se muestran en la tabla 10.1.

En la columna “Fórmula” de la tabla 10.1 las sumas, restas, multiplicaciones y divisiones se corresponden con las operaciones aritméticas habituales aplicadas a los valores numéricos de las direcciones de memoria de los punteros.

Como se puede deducir de la definición de la aritmética de punteros, las expresiones:

```
p[3]
*(p+3)
```

siendo **p** un puntero que apunta a un vector de, al menos, 3 componentes, son equivalentes, siendo la primera más sencilla. En ambos casos el resultado de la expresión es la tercera componente del vector.

### 10.8.3. Conversión implícita

Las únicas operaciones que se pueden aplicar a variables de tipo *array* son el operador indexación y el *address of*<sup>3</sup>. Cualquier otra operación pasa primero por una conversión implícita del vector a un puntero que apunta al tipo básico de las componentes del vector y al primer elemento del mismo. La forma de esta conversión se muestra a continuación:

```
tipoBasico *p;
tipoBasico vector[DIM];
p = vector;
```

La conversión implícita se muestra en la última línea. La conversión permite convertir el vector al puntero teniendo en cuenta que:

1. El tipo puntero que resulta de aplicar la conversión apunta al tipo de la componente del vector. Es decir, si el vector es de tipo **tipoBasico[DIM]** el puntero es de tipo **tipoBasico\***.
2. El valor del puntero que resulta de la conversión es la dirección de memoria del primer elemento del vector. Dicho de otra forma, el puntero apunta al primer elemento del vector.

<sup>3</sup>No es frecuente el uso de este operador aplicado a vectores

### 10.8.4. Declaración de parámetros formales

Aunque en C se puedan declarar parámetros formales de tipo *array* el compilador no los usa de esa manera sino que utiliza en su lugar un parámetro formal que es un puntero al tipo de las componentes del vector. Esto no tiene demasiadas consecuencias en cuanto al uso de parámetros de tipo *array*, excepto que los *arrays* se pasan en cualquier circunstancia y contexto como referencia, es decir, como entrada - salida.

Desde el punto de vista sintáctico las siguientes declaraciones de funciones son equivalentes:

```
void funcion1(tipoBasico vector[DIM]);
void funcion2(tipoBasico *vector);
```

Debido a la conversión implícita de vectores a punteros en ambos casos se puede pasar una variable de tipo **tipoBasico[DIM]** sin ningún tipo de error o aviso.

Debido a esta equivalencia es muy frecuente encontrar bibliotecas (incluida la biblioteca estándar, especialmente en las funciones de `<string.h>`) que utilicen punteros en la declaración de parámetros formales donde se espera un vector. La descripción de la función debe indicar si se espera un vector o una variable por referencia. Una manera de distinguir los casos en los que se espera un vector es la forma en que aparecen este tipo de declaraciones en algunas bibliotecas:

```
void funcion(tipoBasico *a, int n);
tipoRetorno funcion(const tipoBasico *a, int n);
```

donde el parámetro de tipo **int** indica el tamaño a considerar del vector que se va a pasar como argumento y, por tanto, indica de forma indirecta que se espera un vector como argumento. Cuando se escribe el modificador **const** delante del parámetro de tipo puntero significa que el vector no se va a modificar en la ejecución de la función, es decir, es únicamente un vector de entrada.

### 10.8.5. Vectores de punteros y punteros a vector

Es posible declarar vectores cuyos elementos sean punteros y punteros que apunten a vectores, no obstante la sintaxis se complica respecto de los vectores de tipos básicos.

La sintaxis de un vector de punteros que apuntan a un tipo básico es:

```
tipoBasico* id_vector_punteros[DIM];
```

donde **tipoBasico** es alguno de los tipos básicos de C (**int**, **double**, ...), **id\_vector\_punteros** es el identificador de la variable cuyo tipo es vector de punteros que apuntan a tipo básico y **DIM** es el tamaño de la misma.

La sintaxis de un puntero a un vector es más complicada, queda de esta manera:

```
tipoBasico (*id_puntero_a_vector)[DIM];
```

donde **tipoBasico** es alguno de los tipos básicos de C, **id\_puntero\_vector** es una variable puntero que apunta a un vector y **DIM** es el tamaño de ese vector. De esta manera, **id\_puntero\_vector** es un único puntero que apunta a un vector de componentes de tipo básico.

## 10.9. Paso de vectores a funciones

Aunque existen distintas maneras de declarar un parámetro que admita un vector como argumento, todas ellas se traducen a un mecanismo que consiste en pasar el vector a la función como la dirección de memoria del primer elemento del vector. Es posible encontrar las siguientes maneras de escribir el paso de un vector. Para simplificar se supone que se desea pasar un vector de elementos tipo **double**:

```
void funcion_ejemplo_1(double vector[3]);
void funcion_ejemplo_2(double vector[], unsigned tam);
void funcion_ejemplo_3(double *vector);
void funcion_ejemplo_4(double *vector, unsigned tam);
```

Independientemente de la forma de declarar el parámetro **vector** en todos los casos se pasa un puntero al primer elemento del vector.

Los ejemplos se corresponden con los siguientes casos de uso:

1. La función siempre se llama con un vector de tamaño conocido. No obstante, en el momento de la llamada C no comprueba esta circunstancia. El 3 entre corchetes se ignora al compilar y solo es útil para el programador porque le indica el tamaño del vector. Se puede pasar un puntero que apunte al primer elemento de un vector (tipo `double*`).
2. La función puede recibir un argumento que sea un vector de cualquier número de elementos. En el primer argumento se pasa el vector en el segundo el tamaño del mismo. Se puede pasar un puntero que apunte al primer elemento de un vector (tipo `double*`).
3. La función siempre se llama con un vector de tamaño conocido. No obstante, en el momento de la llamada C no comprueba esta circunstancia ni el programador sabe por la declaración cual es el tamaño del vector a pasar. Se puede pasar directamente el vector porque los vectores son convertibles a puntero.
4. La función puede recibir un argumento que sea un vector de cualquier número de elementos. En el primer argumento se pasa el vector en el segundo el tamaño del mismo. Se puede pasar directamente el vector porque los vectores son convertibles a puntero. Esta es con diferencia la declaración más empleada por los programadores en C, especialmente porque es la que refleja con mayor fidelidad cómo ocurre el paso del argumento y tiene en cuenta el tamaño del vector.
5. Cuando se pasa un vector a una función, el vector tiene que estar declarado. Es un error muy grave declarar un puntero que no apunte a un vector y pasarlo como argumento.

## 10.10. Paso de matrices a funciones

Como en el caso de los vectores, aunque existen distintas posibilidades de declarar una función que admite como parámetro una matriz, el mecanismo interno del paso del argumento siempre es el mismo: se pasa una matriz a través de un puntero a la primera de las filas. Pueden encontrarse las siguientes declaraciones:

```
void funcion_ejemplo_1(double matriz[4][3]);
void funcion_ejemplo_2(double matriz[ ][3], unsigned num_filas);
```

A diferencia del caso de los vectores, la declaración de una función que admite como parámetro el puntero a la primera fila es más compleja y se ha omitido por sencillez. Pero, como el arreglo de una matriz en memoria es equivalente a un vector donde se colocan las filas de la matriz de forma consecutiva, si es posible encontrar una declaración como la siguiente:

```
void funcion_ejemplo_3(double *matriz, unsigned filas, unsigned columnas);
```

Cada una de estas declaraciones se corresponden con:

1. La función siempre tiene como parámetro una matriz de dimensión conocida. Aunque internamente el mecanismo de paso del argumento es a través de un puntero, el programador no necesita conocer ese detalle.
2. La función tiene como parámetro una matriz cuyo número de columnas es constante, pero cuyo número de filas es variable. Nótese que esto es consecuencia del paso del puntero a la fila, es necesario conocer el tamaño de una fila, pero no el número de filas que puede ser variable. No se puede hacer al revés, es decir, no se puede declarar de esta manera el paso de una matriz con número de filas constante y número de columnas variable. Tampoco es posible pasar de esta manera una matriz de dimensión variable, es decir ambos, número de filas y columnas variables.
3. La única manera de pasar una matriz estática a una función sin poner como una constante el número de columnas es pasar la dirección de memoria del primer elemento de la matriz (`mat[0]` ó `&(mat[0][0])`) e indicar como argumentos el número de filas y columnas.

Como en el caso anterior, cuando se pasa una matriz a una función, la matriz tiene que estar declarada, no vale con la declaración de un puntero equivalente.

## 10.11. Ejemplos con vectores y matrices

A continuación, en el ejemplo 10.1, se muestra la declaración y uso elemental de un vector de tres elementos de tipo entero.

Ejemplo 10.1: Declaración y uso de un vector de enteros

---

```

1 #include <stdio.h>
2
3
4 #define DIM 3
5
6 int main ()
7 {
8     int v[DIM]; int i;
9     printf("Inicializa a 1 un vector %d componentes\n",DIM);
10    for ( i = 0; i < DIM; ++i ) {
11        v[i]=1;
12    }/*for i*/
13
14    printf("El vector queda: \n");
15    for ( i = 0; i < DIM; ++i ) {
16        printf("v[%d] = %d\n",i,v[i]);
17    }/*for i*/
18
19    return 0;
20 }/*main*/

```

---

Salida del programa:

```

Inicializa a 1 un vector 3 componentes
El vector queda:
v[0] = 1
v[1] = 1
v[2] = 1

```

Como se puede ver en la salida del programa se han inicializado las componentes del vector a uno, pueden tomar valores como se indica en el programa y referirnos a ellas de distintas formas. El tamaño reservado en memoria para este vector es `3*sizeof(int)`.

En el ejemplo 10.2 se ha inicializado el vector y se ha utilizado las propiedades de tamaño de los vectores para conocer su dimensión. Además, se ha multiplicado por 3 todas sus componentes. Este es un ejemplo de un bucle típico que se encuentra en muchos problemas sencillos, se aplica una misma operación o transformación a cada una de las componentes del vector de forma independiente.

Ejemplo 10.2: Inicialización de un vector y multiplicación por 3

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int v[] = {3,1,4,7,4,23,2,1,6,14};
7     int i, dim = sizeof(v) / sizeof(int);
8
9     for ( i = 0; i < dim; ++i ) {
10        printf("%4d ", v[i]);
11    }/*for i*/
12    printf("\n");
13
14    /* multiplica el vector por 3 */
15    for ( i = 0; i < dim; ++i ) {
16        v[i] *= 3;
17    }/*for i*/
18

```

```

19     printf("El vector multiplicado por 3 es: \n");
20     for ( i = 0; i < dim; ++i ) {
21         printf("%4d ", v[i]);
22     }/*for i*/
23     printf("\n");
24
25     return 0;
26 }/*main*/

```

---

Salida del programa:

```

3   1   4   7   4   23   2   1   6   14
El vector multiplicado por 3 es:
9   3   12  21  12  69   6   3   18  42

```

En el ejemplo 10.3 se muestra una inicialización parcial de las componentes de un vector. Las componentes que no se inicializan se ponen a cero. No obstante se desaconseja esta práctica, porque siempre es mejor inicializar de forma explícita todas las variables incluidas las componentes de vectores. En el bucle se cuenta el número de ceros en el vector. Éste es otro ejemplo típico de cálculo agregado. El resultado depende del valor de todas las componentes y se obtiene acumulando el resultado parcial hasta llegar a una componente en una variable que se suele denominar acumulador. En este ejemplo, el acumulador es la **n**, cuyo valor es el número de ceros hasta una componente **i** durante el bucle y el número total de ceros al finalizar el mismo.

---

Ejemplo 10.3: Inicialización parcial de valores de un vector y cuenta de ceros

---

```

1
2 #include <stdio.h>
3
4 #define DIM 10
5
6 int main()
7 {
8     int v[DIM] = {4,5,0,6,3,2};
9     int i, n = 0;
10
11    for ( i = 0; i < DIM; ++i ) {
12        printf("%d ", v[i]);
13    }/*for i*/
14    printf("\n");
15
16    /* Cuenta las componentes iguales a cero */
17    for ( i = 0; i < DIM; ++i ) {
18        if ( v[i] == 0 ) {
19            ++n;
20        }/*if*/
21    }/*for i*/
22
23    printf("Número de ceros en el vector: %d \n", n);
24
25    return 0;
26 }

```

---

Salida del programa:

```

4 5 0 6 3 2 0 0 0 0
Número de ceros en el vector: 5

```

Es el ejemplo 10.4 se muestra un programa que comprueba si dos vectores están ordenados.

---

Ejemplo 10.4: Comprobar si 2 vectores están ordenados

---

```

1
2 #include <stdio.h>
3

```

```

4 #define DIM 6
5
6 int main() {
7     int a[DIM] = { 4, 5, 0, 6, 3, 2 };
8     int b[DIM] = { 4, 7, 10, 16, 33, 42 };
9     int i, ordenado_a = 1, ordenado_b = 1;
10
11     printf("a = ");
12     for ( i = 0; i < DIM; ++i ) {
13         printf("%4d ", a[i]);
14     }/*for i*/
15     printf("\n");
16
17     printf("b = ");
18     for ( i = 0; i < DIM; ++i ) {
19         printf("%4d ", b[i]);
20     }/*for i*/
21     printf("\n");
22
23     /* Comprueba si a y b estan ordenados */
24     for ( i = 0; i < DIM-1; ++i ) {
25         if ( a[i] > a[i+1] ) ordenado_a = 0;
26         if ( b[i] > b[i+1] ) ordenado_b = 0;
27     }/*for i*/
28
29     printf("a esta %s\n", ordenado_a ? "ordenado" : "desordenado" );
30     printf("b esta %s\n", ordenado_b ? "ordenado" : "desordenado" );
31
32     return 0;
33 }

```

---

Salida del programa:

```

a =    4    5    0    6    3    2
b =    4    7   10   16   33   42
a esta desordenado
b esta ordenado

```

En el ejemplo 10.5 se muestra la inicialización y uso básico de una matriz.

#### Ejemplo 10.5: Inicialización y uso básico de una matriz

---

```

1
2 #include <stdio.h>
3
4 /*numero de filas*/
5 #define M 3
6
7 /*numero de columnas*/
8 #define N 4
9
10 int main ()
11 {
12     int i,j;
13     int matriz[M][N];
14
15     for( i = 0; i < M; ++i) {
16         for( j = 0; j < N; ++j) {
17             matriz[i][j] = i*N + j;
18         }/*for j*/
19     }/*for i*/
20
21     for( i = 0; i < M; ++i) {
22         for( j = 0; j < N; ++j) {

```

```

23         printf("%4d ", matriz[i][j]);
24     }/*for j*/
25     printf("\n");
26 }/*for i*/
27
28     return 0;
29 }
```

---

Salida del programa:

```

0   1   2   3
4   5   6   7
8   9   10  11
```

En el ejemplo 10.6 se muestra como se realiza el producto de la matriz del ejemplo anterior por un vector y se visualiza el resultado por pantalla. Se debe prestar especial atención a los rangos de los índices de la matriz y los vectores, especialmente porque la matriz es rectangular.

#### Ejemplo 10.6: Producto de matriz por vector

---

```

1 #include <stdio.h>
2
3 /*numero de filas*/
4 #define M 3
5
6 /*numero de columnas*/
7 #define N 4
8
9
10 int main ()
11 {
12     int i,j; int matriz[M][N];
13     int v[N], b[M]; /*vectores columna*/
14
15     for( i = 0; i < M; ++i) {
16         for( j = 0; j < N; ++j) {
17             matriz[i][j] = i*N + j;
18         }/*for j*/
19     }/*for i*/
20
21     printf("v = ");
22     for( i = 0; i < N; ++i) {
23         v[i] = i*2+1;
24         printf("%4d ",v[i]);
25     }/*for i*/
26     printf("\n");
27
28     for( i = 0; i < M; ++i) {
29         b[i] = 0;
30         for( j = 0; j < N; ++j) {
31             b[i] += matriz[i][j]*v[j];
32         }/*for j*/
33     }/*for i*/
34
35     printf("b = ");
36     for( i = 0; i < M; ++i) {
37         printf("%4d ",b[i]);
38     }/*for i*/
39     printf("\n");
40
41     return 0;
42 }
```

---

Salida del programa:

```
v =     1     3     5     7
b =   34    98   162
```

## Ejercicios propuestos del capítulo de Vectores

- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 10.7: Ejemplo de uso de un vector de enteros

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a [3]; int i;
7     printf("El tamaño del array es: %d bytes\n", sizeof(a));
8     printf("El tamaño cada elemento es: %d\n", sizeof(a[0]));
9     a[0] = 14; a[1] = -23; a[2] = 4;
10    for (i=0; i<3; i++)
11    {
12        printf("a[%d]: %d\n", i, a[i]);
13    }
14    return 0;
15 }
```

---

- Construir un programa que almacene en un *array* de enteros los valores de 1 al 10 y en otro de valores reales las raíces cuadradas correspondientes y luego los visualice por pantalla.
- Escribir una función que cambie un vector multiplicando todas sus componentes por un número. ¿Se puede evitar que cambie el vector original? ¿Qué ocurre si se escribe *const* en la declaración del vector parámetro de la función?
- Escriba en un programa los siguientes ítems: i) la declaración e inicialización de un vector con valores conocidos, ii) la declaración e inicialización de un puntero para que apunte al primer elemento del vector iii) una sentencia que muestre por pantalla (especificador de formato: %p) el resultado de la suma del puntero más un entero, y iv) otra sentencia que muestre el valor del operador indirección aplicado a la suma anterior. Razone el resultado. Pruebe a hacer lo mismo con el operador corchete aplicado al puntero.
- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 10.8: Ejemplo de función con parámetro de tipo vector

---

```

1
2 #include <stdio.h>          /* Para poder emplear printf */
3 #include <stdlib.h>          /* Para poder emplear srand y rand */
4 #include <time.h>           /* Para poder emplear time */
5
6 double sumatorio(double a[], int n)
7 {
8     double aux = 0;
9     int i;
10    for (i=0; i<n; i++)
11    {
12        aux += a[i];
13    }
14    return aux;
15 }
16
17 int main()
18 {
19     double x[10]; int i;
20     srand(time(0));
21     for (i=0; i<10; i++)
```

```

22     {
23         x[i] = rand()/100.0;
24         printf("x[%d] = %f\n", i, x[i]);
25     }
26     printf("El sumatorio es %f",sumatorio(x, sizeof(x)/sizeof(x[0])));
27     return 0;
28 }
```

---

6. Construir una función que devuelva el producto escalar de dos vectores cuyas coordenadas se almacenan en sendos parámetros formales de tipo array (**double v1[]**). La función debe incluir un tercer parámetro que especifique la dimensión del array.

7. Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 10.9: Simulacion del lanzamiento de dos dados

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     int dado1, dado2, dado3, resultado, i;
8     int frecuencia[] = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
9
10    srand( 20 );
11    for ( i = 1; i <= 100000; i++)
12    {
13        dado1 = rand() % 6 + 1; /* Entre 1 y 6 */
14        dado2 = rand() % 6 + 1; /* Entre 1 y 6 */
15        dado3 = rand() % 6 + 1; /* Entre 1 y 6 */
16        resultado = dado1+dado2+dado3;
17        frecuencia[resultado-3]++;
18    }
19
20    printf("Frecuencias de resultados de las tiradas:\n");
21    for ( i = 0; i <= 15; i++)
22    {
23        printf("Frecuencia de %d = %d\n",i+3,frecuencia[i]);
24    }
25    return 0;
26 }
```

---

8. Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 10.10: Uso de una matriz 2 x 3

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int m [2][3] = { {15, 32, -43}, {94, -52, 46} };
7     int i,j;
8
9     printf("El tamaño del array es: %d\n", sizeof(m));
10    printf("El tamaño de cada fila es: %d\n", sizeof(m[0]));
11    printf("El tamaño de cada elemento es: %d\n", sizeof(m[0][0]));
12
13    for ( i=0; i < 2; i++)
14    {
15        for ( j=0; j < 3 ; j++)
16        {
```

```

17         printf("m[%d,%d]: %d\n", i, j, m[i][j]);
18     }
19 }
20 return 0;
21 }
```

---

## Soluciones a los ejercicios propuestos del capítulo de Vectores

1. La salida por pantalla es:

```

El tamaño del array es: 12 bytes
El tamaño cada elemento es: 4
a[0]: 14
a[1]: -23
a[2]: 4
```

2. Programa con array de enteros del 1 al 10 y array de reales con las raíces cuadradas correspondientes

Ejemplo 10.11: Ejemplos de vector de enteros y vector de reales

---

```

1
2 #include <stdio.h>
3 #include <math.h>
4
5 int main()
6 {
7     int n[10]; float raiz[10]; int i;
8
9     for ( i = 0; i < 10; i++ )
10    {
11        n[i] = i+1;
12        raiz[i] = sqrt(n[i]);
13        printf("Raiz cuadrada de %d = %f\n",n[i],raiz[i]);
14    }
15    return 0;
16 }
```

---

Salida por pantalla en la ejecución:

```

Raiz cuadrada de 1 = 1.000000
Raiz cuadrada de 2 = 1.414214
Raiz cuadrada de 3 = 1.732051
Raiz cuadrada de 4 = 2.000000
Raiz cuadrada de 5 = 2.236068
Raiz cuadrada de 6 = 2.449490
Raiz cuadrada de 7 = 2.645751
Raiz cuadrada de 8 = 2.828427
Raiz cuadrada de 9 = 3.000000
Raiz cuadrada de 10 = 3.162278
```

3. La salida por pantalla es:

```

x[0] = 11318000.070000
x[1] = 13984243.650000
x[2] = 619305.930000
x[3] = 8128426.900000
x[4] = 6968975.600000
x[5] = 11217044.140000
x[6] = 5321321.540000
```

```
x[7] = 845497.820000
x[8] = 5598085.590000
x[9] = 11300268.640000
El sumatorio es 75301169.880000
```

4. Programa con una función que devuelve el producto escalar de dos vectores

Ejemplo 10.12: Ejemplo de función con parámetros de tipo vector

---

```

1  #include <stdio.h>
2
3
4  double productoEscalarN(double v1[], double v2[], int n)
5  {
6      double aux = 0.0; int i;
7
8      for ( i = 0; i < n; i++)
9      {
10         aux += v1[i]*v2[i];
11     }
12     return aux;
13 }
14
15 int main()
16 {
17     double x [] = {2.34, 4.09, 5.6, -1.8, 7.3};
18     double y [] = {0.34, 3.09, 4.6, 9.8, -1.5};
19     printf("El producto escalar para n=1 es ");
20     printf("%f\n", productoEscalarN(x,y,1));
21     printf("El producto escalar para n=3 es ");
22     printf("%f\n", productoEscalarN(x,y,3));
23     printf("El producto escalar para n=5 es ");
24     printf("%f\n", productoEscalarN(x,y,5));
25     return 0;
26 }
```

---

Salida por pantalla en la ejecución:

```
El producto escalar para n=1 es 0.795600
El producto escalar para n=3 es 39.193700
El producto escalar para n=5 es 10.603700
```

5. La salida por pantalla es:

Frecuencias de resultados de las tiradas:  
Frecuencia de 3 = 433  
Frecuencia de 4 = 1443  
Frecuencia de 5 = 2704  
Frecuencia de 6 = 4706  
Frecuencia de 7 = 6954  
Frecuencia de 8 = 9888  
Frecuencia de 9 = 11586  
Frecuencia de 10 = 12372  
Frecuencia de 11 = 12315  
Frecuencia de 12 = 11649  
Frecuencia de 13 = 9749  
Frecuencia de 14 = 6939  
Frecuencia de 15 = 4671  
Frecuencia de 16 = 2751  
Frecuencia de 17 = 1360  
Frecuencia de 18 = 480

6. La salida por pantalla es:

```
El tamaño del array es: 24
El tamaño de cada fila es: 12
El tamaño de cada elemento es: 4
m[0,0]: 15
m[0,1]: 32
m[0,2]: -43
m[1,0]: 94
m[1,1]: -52
m[1,2]: 46
```

## Otros ejercicios propuestos del capítulo de Vectores

1. Construir un programa que pida diez valores numéricos reales (**double**), los almacene en un array y posteriormente los visualice por pantalla así como a su valor máximo.
2. Construir las siguientes funciones que trabajen con un array de enteros (**int v[]**):
  - a) Impresión de valores por pantalla.
  - b) Asignación de valores nulos.
  - c) Asignación de valores aleatorios.
  - d) Asignación de valores por teclado.
  - e) Asignación de valores absolutos.
  - f) Cálculo del sumatorio y del producto
  - g) Verificación de si todos son positivos
  - h) Verificación de si todos los elementos son iguales
  - i) Verificación de si todos los elementos están en orden creciente
  - j) Cálculo del valor del elemento máximo
  - k) Cálculo del índice del elemento máximo
3. Construir las siguientes funciones que trabajen con un array de enteros (**int v[]**):
  - a) Calcular el número de elementos positivos
  - b) Calcular el número de elementos iguales a 0
  - c) Asignar valores iguales al doble del índice correspondiente
  - d) Asignar valores aleatorios en orden creciente
  - e) Asignar valores distintos a todos los elementos
  - f) Verificar si existe un valor determinado (dato como parámetro) asignado a algunos de los elementos
  - g) Indicar cuántas veces se ha asignado un valor determinado (dato como parámetro) a los elementos del array
  - h) Verificar si coincide el valor de todos los elementos con el mismo índice de dos arrays distintos dados como parámetros
  - i) Devolver el índice del primer elemento que coincide con un valor determinado (dato como parámetro)
  - j) Ordenar de forma creciente los valores de los elementos del array
4. Construir un programa que concatene dos vectores de números enteros previamente creados y que visualice el resultado por pantalla.
5. Construir un programa que calcule el determinante de una matriz 2x2 de números reales
6. Construir un programa que calcule el determinante de una matriz 3x3 de números reales
7. Construir un programa que asigne valores aleatorios a los elementos de una matriz 3x3 de números reales

8. Construir un programa que asigne valores por teclado a los elementos de una matriz 3x3 de números reales
9. Asignar la matriz identidad a una matriz 3x3 de números reales (a los elementos de la diagonal se les asigna el valor 1, el resto se le asigna un 0)
10. Construir una función que devuelva la traza de una matriz 3x3 de reales dada como parámetro.
11. Construir una función que devuelva el mayor de los elementos de una matriz 3x3 de reales dada como parámetro.
12. Construir una función que devuelva 1 si la matriz 3x3 de reales almacena la matriz identidad y 0 en caso contrario.
13. Construir una función que devuelva el número de elementos estrictamente positivos de una matriz 3x3 de números reales
14. Construir un programa que genere dos matrices reales con elementos de valor aleatorio, visualice dichas matrices, las sume y, finalmente visualice la matriz suma.
15. Construir un programa que realice el producto de dos matrices cuadradas.
16. Construir un programa que realice la transpuesta de una matriz. Más difícil: si se hace sobre la propia matriz. Más difícil aún: la traspuesta respecto de la diagonal secundaria.
17. Declare un puntero a un vector fila. Se recomienda declarar primero el tipo fila y luego declarar la variable de tipo puntero.
18. Declare un vector cuyas componentes sean del tipo puntero a fila (el tipo declarado anteriormente). Escriba un programa que inicialice este vector para que guarde a través de variables dinámicas los valores de una matriz. Escribir una rutina que intercambie los filas de dicha matriz.

# Capítulo 11

## Cadenas de caracteres

La programación en el lenguaje C tiene varios tipos de datos estructurados y un caso muy particular de vector cuyos elementos son de tipo carácter. En este capítulo se cubrirán las cadenas de caracteres de texto.

Objetivos:

1. Describir el tipo de dato cadena de caracteres, su estructura y forma de uso (Conocimiento)
2. Interpretar el código fuente de un programa en C donde aparezcan cadenas de caracteres (Comprensión)
3. Codificar una tarea, convenientemente especificada, utilizando cadenas de caracteres (Aplicación)

### 11.1. Cadenas de caracteres

Una cadena de caracteres es el tipo de dato compuesto que permite almacenar datos de tipo alfanumérico con cero, uno o más caracteres. Una cadena alfanumérica o de caracteres es una secuencia de letras, dígitos, símbolos o signos de puntuación codificada en alguno de los estándares de codificación de texto.

El lenguaje C utiliza vectores de elementos de tipo **char** para almacenar cadenas alfanuméricas. Todo lo que se ha dicho para vectores se puede aplicar de forma directa a las cadenas de caracteres, pero además:

1. En C se permite una inicialización particular para cadenas de caracteres sin necesidad de las llaves de los vectores, su sintaxis es:

```
char id_cadena[dim_opt] = "texto libre";
```

donde

- **id\_cadena** es el identificador de la variable de tipo cadena
- **dim\_opt** es la dimensión o capacidad máxima de la cadena y es opcional. Si no se indica, la capacidad de la cadena se calcula a través del texto de inicialización con el número de caracteres de éste más uno para poder incluir el carácter nulo (que indica el final de la cadena de texto almacenada) y
- **"texto libre"** es el valor de la cadena alfanumérica se quiere asignar. Para incluir determinados caracteres es necesario utilizar la barra invertida.

2. En C una cadena puede tener una *longitud efectiva* o simplemente longitud distinta del tamaño del vector de caracteres mediante la colocación de un carácter especial cuyo ordinal cero que también se denomina como *carácter nulo* (`0` ó `'\0'`). Cuando se inicializa una cadena de caracteres según la sintaxis anterior este carácter se añade de forma automática a la cadena. El tamaño del vector de caracteres también recibe el nombre de *capacidad* de la cadena.
3. Existen varias funciones predefinidas en C que sirven para realizar operaciones comunes con cadenas de caracteres. Dichas funciones se encuentran declaradas en el archivo de cabecera `<string.h>`. Todas estas funciones (y otras muchas de la librería estándar) solo se comportan correctamente para cadenas de caracteres como las descritas, es decir, acabadas en carácter nulo (en inglés *null terminated string*).

El aspecto diferenciador de las cadenas respecto de los vectores es el carácter nulo como marcador de final de la cadena. Esto provoca que se tenga que distinguir entre dos tamaños de la cadena:

- La *capacidad máxima*, es decir, la dimensión del vector, el número máximo de caracteres que podría guardar en cualquier caso.
- La *longitud efectiva*, es decir, el número de caracteres que tiene la cadena en un momento dado y que se calcula contando los caracteres desde su inicio hasta que se encuentra el carácter nulo. Este número es variable según el contenido actual de la cadena. Por supuesto esta longitud efectiva debe ser siempre al menos una unidad menor que la capacidad máxima de la cadena para poder incluir el carácter nulo.

La estructura de una cadena se muestra en la figura 11.1.

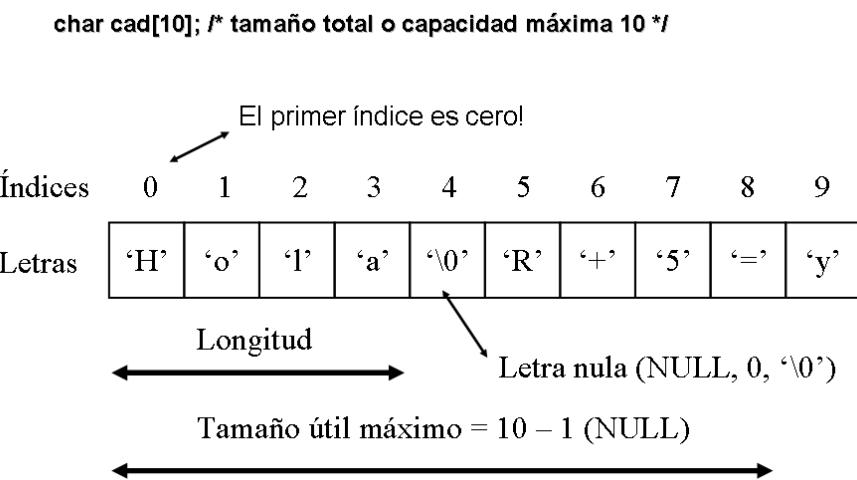


Figura 11.1: Estructura de una cadena

Como en el caso de los vectores no se puede dar valor a una cadena mediante una asignación directa, es decir, **no** se puede indicar que una variable de tipo cadena es igual a una constante literal con comillas dobles. Tampoco es posible comparar cadenas directamente a través del operador de comparación. Todo ello es consecuencia de que las cadenas son convertibles de forma automática al tipo **char\***, es decir a un puntero y dicho puntero es el que se asigna o compara.

## 11.2. Funciones con cadenas alfanuméricas

Existen distintas funciones dentro de la librería estándar de C que trabajan con cadenas alfanuméricas. Todas ellas tienen como característica común el que utilizan punteros para su manipulación y todas suponen que las cadenas que utilicen como datos están terminadas en carácter nulo. Además todas las cadenas que producen como resultado terminan en carácter nulo.

Las declaraciones de estas funciones normalmente se distribuyen entre los archivos de cabecera *<string.h>*, *<stdio.h>* y *<stdlib.h>*. En la tabla 11.1 se muestran los prototipos y el funcionamiento de algunas de estas funciones.

Cuadro 11.1: Funciones básicas con cadenas alfanuméricas en `string.h`

Declaración	Descripción
<code>size_t strlen(const char * str);</code>	Devuelve un entero positivo con la longitud de la cadena, es decir, el número de caracteres hasta el carácter nulo.
<code>char * strcpy(char * destino, const char * fuente);</code>	Copia la cadena <b>fuente</b> a la cadena <b>destino</b> y devuelve de nuevo <b>destino</b> . La copia incluye el carácter nulo. La cadena de <b>destino</b> debe tener el tamaño suficiente para que quepa la cadena <b>fuente</b> .
<code>char * strcat(char * destino, const char * fuente);</code>	Añade la cadena <b>fuente</b> a la cadena <b>destino</b> . El carácter nulo de la cadena <b>destino</b> se sustituye por el primer carácter de la cadena <b>fuente</b> . La cadena <b>fuente</b> se añade hasta el final, incluido su carácter nulo. La cadena <b>destino</b> tiene que tener suficiente tamaño para guardar su contenido anterior y el nuevo. Se devuelve la cadena <b>destino</b> .
<code>int strcmp(const char * str1, const char * str2);</code>	Compara dos cadenas carácter a carácter y devuelve 0 si su contenido es igual, devuelve un entero mayor que cero si el primer carácter de la primera cadena es mayor o un entero menor que cero en caso contrario. Para esta comparación se supone que <b>char</b> no tiene signo.
<code>char * gets(char * str);</code>	Rellena una cadena <b>str</b> con caracteres recogidos de teclado hasta que se introduzca una nueva línea ( <i>enter</i> ). La cadena debe tener el tamaño suficiente para almacenar los caracteres tecleados. El carácter salto de línea no se pone en la cadena, pero sí se añade el carácter nulo al final de la misma. Se devuelve la cadena leída o NULL si se encuentra un error.

Para usar cadenas alfanuméricas en las funciones de entrada - salida estandar con formato (`printf` y `scanf`), se usa su propio especificador de formato: `%s`. Para `printf` el uso es inmediato, simplemente se añade el especificador de formato y el argumento sin más. En el siguiente ejemplo, se muestra una llamada a `printf` y el resultado que produce.

```
char nombre[] = "Juan";
printf("Hola %s\n", nombre);
Hola Juan
```

Para `scanf` la cosa se complica un poco. Hay que tener en cuenta que:

- Una cadena que convierte directamente en un puntero y por eso hay que suprimir el operador & que se usa en `scanf` para las variables de tipos simples.
- La cadena que pasamos tiene un tamaño máximo. En consecuencia, para usar de forma segura `scanf`, es necesario indicar el número máximo de caracteres leídos desde teclado. Si no se impone este límite se puede producir un desbordamiento del límite de la cadena y escribir en memoria en otras variables o provocar otros errores graves. Para imponer este límite se añade entre el `%` y la `s` el número máximo de caracteres que se pueden leer. Por ejemplo: `%31s` indica que se pueden leer hasta 31 caracteres como máximo. Naturalmente, `scanf`, cuando termina de leer los caracteres, inserta la marca de fin de cadena (carácter nulo), así pues la cadena tiene que tener capacidad para 1 carácter más. A modo de resumen, véase el siguiente ejemplo:

```
char cadena[32];
scanf("%31s", cadena);
```

La función `scanf` lee los caracteres de la entrada de datos (teclado) hasta que se encuentra un blanco (espacio, tabulador, salto de línea). Para leer una frase o variables palabras se debe utilizar `gets` que lee toda una línea (hasta que se inserta *intro*). La cadena introducida en `gets` debe tener el tamaño suficiente para guardar toda la línea y el carácter nulo. No hay regla fija, pero es conveniente usar, al menos 100 caracteres, un tamaño del lado de la seguridad puede ser 512.

## 11.3. Uso de funciones básicas con cadenas alfanuméricas

En el ejemplo 11.1 se pueden comprobar distintas características de tamaño y longitud de la cadena, así como funcionalidades muy básicas tales como mostrar la cadena o acceder a los caracteres de la misma.

## Ejemplo 11.1: Definición de cadena alfanumérica

---

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 #define DIM 100
6
7 int main ()
8 {
9     char cad[DIM] = "Cadena de ejemplo";
10    int i = 0;
11
12    printf("printf de la cadena '%s'\n", cad);
13
14    printf("Mostrar todos los caracteres: ");
15    for ( i = 0; cad[i] != '\0'; ++i ) {
16        printf("%c", cad[i]);
17    }
18    printf("\n");
19
20    printf("La longitud de la cadena es %d\n", i);
21    printf("strlen(cad) es %d\n", strlen(cad));
22
23    printf("La capacidad de la cadena es %d\n", DIM);
24    printf("sizeof cad es %d\n", sizeof(cad));
25
26    return 0;
27 }
```

---

Salida del programa:

```

printf de la cadena 'Cadena de ejemplo'
Mostrar todos los caracteres: 'Cadena de ejemplo'
La longitud de la cadena es 17
strlen(cad) es 17
La capacidad de la cadena es 100
sizeof cad es 100
```

En el ejemplo 11.2 se utilizan algunas funciones de las librerías anteriores para manipular información del nombre y apellidos de una persona. En concreto se parte de los apellidos y nombre y se consigue como resultado una única cadena con un saludo, el nombre y apellidos. Es conveniente destacar que el equivalente a la sentencia de asignación a una variable de tipo cadena es una llamada a la función `strcpy` tal y como aparece en la línea 15.

## Ejemplo 11.2: Manipulación de apellidos y nombre

---

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 #define DIM 512
6
7 int main () {
8     /* Apellidos y nombre */
9     char nombre[DIM] = "Juan";
10    char apellido1[DIM] = "Madrid";
11    char apellido2[DIM] = "Carpintero";
12    char salida[DIM];
13
14    /* Se inicializa la cadena */
15    strcpy(salida, "hola ");
16
17    /* Se concatenan el nombre y los apellidos */
18    strcat(salida, nombre);
```

```

19     strcat(salida, " ");
20     strcat(salida, apellido1);
21     strcat(salida, " ");
22     strcat(salida, apellido2);
23
24     /* Se muestra el resultado */
25     printf("%s\n", salida);
26
27     return 0;
28 }
```

---

Salida del programa:

hola Juan Madrid Carpitero

## Ejercicios propuestos del capítulo de Cadenas

- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 11.3: Ejemplo de uso de cadenas

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     char s1[15];
7     char s2[10] = "Hello";
8
9     s1[0]='A'; s1[1]='d'; s1[2]='i'; s1[3]='o'; s1[4]='s';
10    s1[5]='\0'; /* equiv. a s1[5]=0; */
11
12    printf("Cadena 1: %s; %d bytes\n", s1, sizeof(s1));
13    printf("Cadena 2: %s; %d bytes\n", s2, sizeof(s2));
14    printf("bye ocupa %d bytes\n", sizeof("bye"));
15    return 0;
16 }
```

---

- Construir un programa que permita al usuario introducir por teclado su nombre y su apellido y luego los visualice por pantalla
- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 11.4: Ejemplo de uso de funciones de cadenas

---

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char s1[20]="Juan";
8     char s2[20]="Perez";
9     char s3[40]="";
10
11    printf("Cadena 1: %s\n", s1);
12    printf("Longitud: %d caracteres\n", strlen(s1));
13    printf("Dimension total: %d bytes\n", sizeof(s1));
14    printf("Cadena 2: %s\n", s2);
15    printf("Cadena 3: %s\n", s3);
16    printf("Longitud: %d caracteres\n", strlen(s3));
17    strcat(s1, s2);
18    printf("Cadena 1: %s\n", s1);
```

```

19     printf("Longitud: %d caracteres\n", strlen(s1));
20     printf("1 y 2 iguales?: %d\n", strcmp(s1,s2));
21     strcpy(s1, s2);
22     printf("Cadena 1: %s\n", s1);
23     printf("Longitud: %d caracteres\n", strlen(s1));
24     printf("1 y 2 iguales?: %d\n", strcmp(s1,s2));
25     return 0;
26 }
```

---

4. Construir una función que visualice por pantalla en orden inverso una cadena dada como parámetro de la función
5. Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 11.5: Funcion que copia cadena

```

1
2 #include <stdio.h>
3
4 char * copia(char s1 [], char s2 [])
5 {
6     int i = 0;
7     while (s2[i] != '\0')
8     {
9         s1[i] = s2[i];
10    i++;
11 }
12 s1[i] = s2[i]; /* Copia \0 */
13 return s1;
14 }
15
16 int main()
17 {
18     char a[] = "Titulacion oficial UPM";
19     char b[] = "Ingenieria Industrial";
20
21     copia(a, b);
22     printf("Valor de a: %s\n", a);
23     printf("Valor de b: %s\n", b);
24     return 0;
25 }
```

---

## Soluciones a los ejercicios propuestos del capítulo de Cadenas

1. La salida por pantalla es:

Cadena 1: Adios; 15 bytes  
 Cadena 2: Hello; 10 bytes  
 bye ocupa 4 bytes

2. Programa que permite al usuario introducir por teclado su nombre y su apellido y luego los visualiza por pantalla

Ejemplo 11.6: Ejemplos de lectura por teclado de cadenas

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     char nombre[20];
7     char apellido[30];
8     scanf("%s", nombre);
9     scanf("%s", apellido);
```

```

10
11     printf("Tu nombre es: %s \n", nombre);
12     printf("Tu apellido es: %s \n", apellido);
13     printf("Hola, %s %s\n", nombre, apellido);
14     return 0;
15 }
```

---

Al introducir por teclado en la ejecución:

Sara  
Garcia

Salida por pantalla en la ejecución:

Tu nombre es: Sara  
Tu apellido es: Garcia  
Hola, Sara Garcia

3. La salida por pantalla es:

```

Cadena 1: Juan
Longitud: 4 caracteres
Dimension total: 20 bytes
Cadena 2: Perez
Cadena 3:
Longitud: 0 caracteres
Cadena 1: JuanPerez
Longitud: 9 caracteres
1 y 2 iguales?: -1
Cadena 1: Perez
Longitud: 5 caracteres
1 y 2 iguales?: 0
```

4. Función que visualiza por pantalla en orden inverso una cadena dada como parámetro de la función

Ejemplo 11.7: Ejemplos de función con parámetro de tipo cadena

---

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 void inverso(char s[] )
6 {
7     int i;
8     for ( i = strlen(s)-1; i >=0 ; i-- )
9     {
10         printf("%c", s[i]);
11     }
12     printf("\n");
13     return;
14 }
15
16 int main()
17 {
18     char cad[] = "Hola";
19     printf("%s\n", cad);
20     inverso(cad);
21     return 0;
22 }
```

---

Salida por pantalla en la ejecución:

Hola  
aloH

5. La salida por pantalla es:

Valor de a: Ingenieria Industrial  
Valor de b: Ingenieria Industrial

## Ejercicios propuestos del capítulo de Cadenas

1. Construir un programa que pida introducir una cadena de caracteres por teclado y visualice por pantalla el número de caracteres que contiene dicha cadena.
2. Contar el número de veces que aparece una palabra en un texto.
3. Sustituir una palabra por otra en un texto, dado por teclado o en una cadena.
4. Dividir un texto mediante un separador y almacenar el resultado en un vector de cadenas alfanuméricas. Para realizar este ejercicio es recomendable suponer un tope máximo de elementos a separar en el texto.
5. Dada una palabra, insertarla en una cadena alfanumérica.
6. Dada una palabra, eliminarla de una cadena alfanumérica.
7. Cambiar la primera letra de las palabras en un texto a mayúscula.
8. Construir las siguientes funciones que trabajen con una cadena de caracteres como parámetro de la función:
  - a) Impresión en orden inverso por pantalla
  - b) Verificación de si todos los caracteres son dígitos
  - c) Verificación de si todos los caracteres son letras
  - d) Verificación de si todos los caracteres de la cadena son iguales
  - e) Calcular el número de caracteres alfabéticos
  - f) Calcular el número de caracteres correspondientes a dígitos decimales
  - g) Verificar si existe un carácter determinado (dado como parámetro) asignado a algunos de los caracteres
  - h) Indicar cuántas veces se ha asignado un carácter determinado (dado como parámetro) a los elementos de la cadena
  - i) Devolver el índice del primer elemento que coincide con un carácter determinado (dado como parámetro)
9. Construir las siguientes funciones que trabajen con una cadena de caracteres (equivalentes a las incluidas en la librería **string.h**):
  - a) Función que devuelve la longitud de la cadena (equivalente a `int strlen(s)`)
  - b) Función que copia el contenido de una cadena en otra (equivalente a `char* strcpy(s1,s2)`)
  - c) Función que devuelve el resultado de comparar dos cadenas (equivalente a `int strcmp(s1,s2)` )
  - d) Función que devuelve el resultado de concatenar dos cadenas (equivalente a `char* strcat(s1,s2)`)
10. Construir una función que convierta un valor numérico entero dado como parámetro en una cadena de caracteres
11. Construir una función que elimine los espacios en blanco de una cadena de caracteres. Por ejemplo, "Pedro Perez 123" quedaría como "PedroPerez123".
12. Construir una función que transforme una cadena en la correspondiente en mayúsculas. Por ejemplo, "Pedro Perez 123" en "PEDRO PEREZ 123".

# Capítulo 12

## Estructuras

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Definir el concepto de estructura (Conocimiento)
2. Escribir la declaración de tipos de estructuras necesaria para poder trabajar con ellas (Comprensión)
3. Interpretar el código fuente de un programa que trabaje con una estructura de datos (Comprensión)
4. Codificar una tarea sencilla convenientemente especificada, utilizando una estructura de datos (Aplicación)

### 12.1. Introducción

Según se incrementa el tamaño de los programas se hace necesaria una mayor estructuración de los datos que se manejan en el código. La manera de conseguirlo es mediante el empleo de tipos *compuestos* que son capaces de almacenar en una única variable los datos de varias (puede ser muchas) variables más simples. El uso de tipos compuestos que incorporen otros tipos compuestos puede dar lugar a diseños de datos tan complejos como sea necesario.

Los tipos compuestos pueden agrupar variables del mismo tipo. Los vectores y cadenas alfanuméricas son ejemplos de estos tipos compuestos cuyas variables están formadas por varias variables de tipos más simples y cuya utilidad es la capacidad de agrupar y usar una gran cantidad de datos mediante una única variable. Así por ejemplo, hasta ahora se han usado vectores de números enteros para agrupar varios números enteros (`int[]`) o para agrupar números en coma flotante (`float[]`), pero no se han mezclado. De hecho, los vectores no son capaces de unir en la misma variable otras variables más simples que sean de distinto tipo. Para llevar a cabo esta tarea se necesita un nuevo tipo de dato: la *estructura* o `struct`.

### 12.2. El tipo struct o estructura

El tipo `struct` de C es capaz de unir en un único tipo de variable otras variables más simples de distinto tipo entre ellas.

La declaración de un tipo `struct` implica la definición de los tipos de datos que lo componen. Así pues, no se puede declarar una variable simplemente como `struct`, sino que será necesario especificar cómo es el `struct` antes de declarar la variable.

La sintaxis de declaración de un `struct` es como sigue:

```
struct tipo_structura
{
    tipo1 identificador1;
    tipo2 identificador2;
    ....
    tipon identificadorn;
};
```

La declaración<sup>1</sup> consta de la palabra reservada **struct** seguida por el identificador correspondiente al nuevo tipo de dato y un bloque (dentro de las llaves) donde se declaran los campos o atributos del **struct** tal y como se declararían variables globales o locales. No existe ningún tipo de limitación a los tipos y variables que se pueden declarar dentro del **struct**, excepto que no puede haber un campo con identificador repetido. Una variable que se declare de un tipo **struct** contiene como variables más simples todos los campos o atributos declarados en el **struct**.

La declaración de variables de tipo **struct** se realiza igual que para cualquier otro tipo de dato. Naturalmente se pueden declarar punteros a estructuras o vectores de estructuras y, en general, cualquier combinación, modificación u operación que se pueda aplicar a otros tipos de datos también se puede realizar con los tipos estructura siempre y cuando tenga sentido. Por ejemplo, se puede aplicar la operación **sizeof** para calcular el tamaño en memoria de una estructura. El contenido de una variable **struct** A se puede asignar a otra B si son del mismo tipo: como resultado se copia la información en memoria de A a B. Como operación propia, el acceso a los atributos de una estructura se hace utilizando el operador punto (.) después del identificador de la variable de tipo estructura seguido del identificador del atributo.

En el ejemplo 12.1 se puede ver la declaración de un tipo estructura, de una variable tipo estructura y de un par de casos de uso de sus atributos.

Ejemplo 12.1: Declaración de una estructura

---

```

1
2 #include <string.h>
3 #include <stdio.h>
4
5 struct Estructura
6 {
7     int codigo;
8     char nombre[30];
9     double valor;
10};
11
12 int main()
13 {
14     struct Estructura st;
15
16     st.codigo = 7;
17     strcpy(st.nombre, "temperatura");
18     st.valor = 23.4;
19
20     printf("codigo: %d, nombre: %s, valor: %f \n",
21            st.codigo, st.nombre, st.valor);
22
23     st.valor += 20.0;
24
25     printf("codigo: %d, nombre: %s, valor: %f \n",
26            st.codigo, st.nombre, st.valor);
27
28     return 0;
29 }
```

---

Salida del programa:

```

codigo: 7, nombre: temperatura, valor: 23.400000
codigo: 7, nombre: temperatura, valor: 43.400000
```

### 12.3. Funciones y struct

Las funciones pueden tener parámetros de tipo **struct**. El mecanismo de funcionamiento es el de un parámetro formal por valor como para los tipos básicos. Esto significa que una estructura que se utilice como argumento de una función no se va a modificar dentro de la función puesto que la función recibe una copia de la estructura que se pase como argumento.

---

<sup>1</sup> La declaración de tipos **struct** se puede realizar de muy diversas maneras. Incluso hay una forma de declaración que se llama incompleta y que permite la declaración del contenido de la estructura más adelante, cuando sea estrictamente necesaria.

Cuando se utiliza como parámetro un puntero a **struct** se está utilizando el paso de un puntero o referencia a la estructura y de nuevo el paso del argumento (la dirección de memoria de una estructura) es análogo al caso de parámetros de tipo puntero a tipo básico. Por otro lado una función puede tener variables locales de tipo estructura.

Finalmente una función puede devolver un dato de tipo estructura. Se debe tener en cuenta que la estructura que recibe el resultado de la función copia toda la información de la estructura que se devuelve sobre sus propios campos.

En el ejemplo 12.2 se puede observar una función que ilustra todas estas opciones.

---

#### Ejemplo 12.2: Funciones y struct

---

```

1  #include <stdio.h>
2
3
4  struct Datos { int num; double valor; };
5
6  struct Datos ejemplo (struct Datos x1, struct Datos *x2)
7  {
8      struct Datos aux;
9
10     x1.num += 5;
11
12     aux.num = 2 * x1.num;
13     aux.valor = 2 * x1.valor;
14
15     *x2 = x1;
16
17     return aux;
18 }
19
20 void mostrar(const struct Datos* x)
21 {
22     printf("{ %i, %f }\n", x->num, x->valor);
23 }
24
25 int main()
26 {
27     struct Datos a = { 5, 13.13 };
28     struct Datos b, c;
29
30     c = ejemplo(a, &b);
31
32     mostrar(&a);
33     mostrar(&b);
34     mostrar(&c);
35
36     return 0;
37 }
```

---

Salida del programa:

```
{ 5, 13.130000 }
{ 10, 13.130000 }
{ 20, 26.260000 }
```

## 12.4. Arrays y struct

Se pueden declarar vectores cuyos elementos sean de un tipo **struct**. Por ejemplo:

```

struct alumno { int matricula; double nota; };
struct alumno a;
struct alumno grupo [100];
/* ... */
grupo[15].matricula = 10915;
```

```

grupo[15].nota = 7.50;
/* ... */
grupo[38] = a;

```

donde `a` es una variable sencilla de tipo `struct alumno`, pero `grupo` es un vector de 100 variables cada una de ellas de tipo `struct alumno`. El manejo de *arrays* de *structs* es análogo al manejo de arrays de cualquier tipo simple.

Sin embargo el uso de *arrays* dentro de *structs* si tiene una particularidad interesante. Como se ha visto en el tema de vectores y matrices, todos los vectores y matrices son parámetros por referencia independientemente de cómo se declare en parámetro en el prototipo de la función. Sin embargo, si el vector o matriz se declara **dentro** de una estructura, la forma de paso de la estructura domina y el vector o matriz si **se puede pasar por valor**. Nótese que si la estructura es muy grande esto puede ser un inconveniente y no una ventaja.

## 12.5. Punteros a estructuras

Pueden declararse variables puntero que apunten a datos de tipo `struct`. Por ejemplo:

```

struct alumno a;
struct alumno *p = &a;
/* ... */
p->matricula;
(*p).matricula; /* Equivalente al anterior */

```

Desde el punto de sintaxis la flecha (`->`) y el punto (`.`) son operaciones que se aplican a una variable de tipo `struct` que se coloca a la izquierda para obtener una variable que se corresponde con el campo cuyo identificador se coloca a la derecha.

## 12.6. Uso del tipo estructura

Existen algunos casos donde el uso de `struct` es especialmente adecuado. A continuación se van a estudiar algunos de ellos, junto con los ejemplos correspondientes. En este capítulo se mostrará el uso más básico: como una manera de agrupar información. En el capítulo de variables dinámicas se usará el tipo `struct` para implementar estructuras dinámicas.

### 12.6.1. Agrupación de datos que definen un objeto del problema

Existen muchos elementos de problemas reales cuya información se puede representar mediante la agrupación de varios datos característicos. En estos casos es frecuente tener que utilizar varias variables que representen estos elementos del problema o que se utilicen como datos de funciones utilizadas en el programa. En ambos casos el uso de una estructura simplifica el código resultante.

A modo de ejemplo (12.3), se va a considerar que un elemento característico de un problema es un vector de números en coma flotante con un número dado de componentes válidas y una cadena alfanumérica que identifica el tipo de magnitud física. Respecto de este elemento se va a realizar una operación suma y un programa que declara varios de estos elementos y prueba la suma.

Ejemplo 12.3: Estructura con datos auxiliares de un vector

---

```

1
2 #include <string.h>
3 #include <stdio.h>
4 #include <math.h>
5
6 struct VectorConEtiqueta {
7     char magnitud[30];
8     double valores[200];
9     unsigned tam;
10 };
11
12 int sumar(struct VectorConEtiqueta *res,
13             const struct VectorConEtiqueta *v1,
14             const struct VectorConEtiqueta *v2)

```

```

15 {
16     if ( strcmp((*v1).magnitud, (*v2).magnitud) == 0 ) {
17         unsigned i;
18         if ((*v1).tam < (*v2).tam) (*res).tam = (*v1).tam;
19         else (*res).tam = (*v2).tam;
20         for ( i = 0; i < (*res).tam; i++ ) {
21             (*res).valores[i] =
22                 (*v1).valores[i] + (*v2).valores[i];
23         }/*for i*/
24         strcpy((*res).magnitud, (*v1).magnitud);
25         return 1;
26     }/*if*/
27     else {
28         printf("Las magnitudes son diferentes\n");
29         printf("No se puede sumar\n");
30         return 0;
31     }
32 }/*sumar*/
33
34 int main()
35 {
36     struct VectorConEtiqueta vectores[10];
37     unsigned i,j; struct VectorConEtiqueta res;
38
39     for ( i = 0; i < 10; i++ ) {
40         strcpy(vectores[i].magnitud,"temperatura");
41         vectores[i].tam = i + 100;
42         for ( j = 0; j < vectores[i].tam; j++ ) {
43             vectores[i].valores[j] = i + j + 2.2;
44         }/* for j */
45     }/* for i */
46     sumar(&res, &(vectores[2]), &(vectores[4]));
47     printf("magnitud: %s, valores[3]: %f, tam: %d \n",
48           res.magnitud, res.valores[3], res.tam);
49     return 0;
50 }/*main*/

```

---

Salida del programa:

```
magnitud: temperatura, valores[3]: 16.400000, tam: 102
```

Aunque el programa resulte extenso, se destacan las ideas más importantes:

1. El uso de la estructura, `VectorConEtiqueta` simplifica la declaración de una función, `sumar`, que opera con todos sus datos. Notese que esta función realiza comprobaciones sobre los atributos `magnitud` y `tam`, de forma que, de no haberse declarado el `struct` obligaría a que la función tuviese 9 parámetros en vez de los 3 que tiene.
2. El uso de `VectorConEtiqueta` simplifica la declaración de los 20 elementos de este dato que se van a necesitar. De no usarse el `struct` habría que declarar 20 vectores de números, 20 cadenas alfanuméricas para las magnitudes y otros 20 números para el tamaño.

Se recomienda analizar el programa en profundidad hasta entender todo su contenido.

### 12.6.2. Programación orientada a objetos en C

Aunque estrictamente hablando C no es un lenguaje orientado a objetos sí es posible hacer programación orientada a objetos en C puro (sin C++, la extensión a objetos de C) precisamente mediante el uso de tipos basados en `struct`. Incluso es posible cierto grado de polimorfismo<sup>2</sup> mediante el uso de punteros a funciones.

El desarrollo de ejemplos que ilustren la programación orientada a objetos en C queda muy lejos del alcance de este texto dado que la programación orientada a objetos no es solo sintaxis sino, también y sobre todo, una manera

<sup>2</sup>de manera muy elemental, una rutina, método, módulo o componente de un programa escritos usando polimorfismo se comporta de manera distinta en tiempo de ejecución a pesar de tener una implementación fija.

de diseñar y estructurar el código. No obstante, para entender cómo se logra este enfoque se recomienda el estudio de librerías que emplean C orientado a objetos, por ejemplo: *Cairo Graphics*, *GTK*, *gsl*, etc. Todas ellas son ejemplos representativos de orientación a objetos en C.

## Ejercicios propuestos del capítulo de Estructuras

- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 12.4: Ejemplo de uso de una estructura

---

```

1  #include <stdio.h>
2  #include <string.h>
3
4
5  struct alumno
6  {
7      int matricula;
8      char nombre[40];
9      float nota;      /* Formato alternativo */
10 } ;                  /* la declaracion acaba con ; */
11
12 int main()
13 {
14     struct alumno a = { 19000, "Pedro Gomez", 7.86 };
15     printf("Nº de matricula: %d\n", a.matricula);
16     printf("Nombre: %s\n", a.nombre);
17     printf("Nota: %f\n", a.nota);
18     printf("Tamaño de a: %d\n", sizeof(a));
19     return 0;
20 }
```

---

- Construir un programa que declare y use dos variables **struct** cada una de las cuales represente un *punto* en el espacio bidimensional. Los miembros de la variable **struct** deben ser de tipo **float**. El programa debe asignar valores a las respectivas coordenadas de los puntos y calcular la distancia geométrica entre ellos.

- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 12.5: Estructuras y funciones

---

```

1
2  #include <stdio.h>
3  #include <math.h>
4
5  struct Punto
6  {
7      float x;
8      float y;
9  };
10
11 void muestra (struct Punto p)
12 {
13     printf("Se situa en x = %f", p.x);
14     printf(" e y = %f \n", p.y);
15 }
16
17 struct Punto asigna (float x, float y)
18 {
19     struct Punto aux;
20     aux.x = x;
21     aux.y = y;
22     return aux;
23 }
24
```

```

25 struct Punto suma (struct Punto p1, struct Punto p2)
26 {
27     p1.x += p2.x;
28     p1.y += p2.y;
29     return p1;
30 }
31
32 int main()
33 {
34     struct Punto a = {5.09, 2.36};
35     struct Punto b;
36     float d;
37
38     b.x = 7.22;
39     b.y = -3.89;
40
41     printf("A se situa en x = %f e y = %f \n", a.x, a.y);
42     printf("B se situa en x = %f e y = %f \n", b.x, b.y);
43
44     d = sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));
45     printf("La distancia entre A y B es %f\n", d);
46
47     return 0;
48 }
```

---

4. Construir las siguientes funciones que trabajan con datos de tipo punto (**struct**):

- Función **inicializa** que modifica, asignando valores iguales a 0, a los miembros del parámetro formal de tipo **\*punto**
  - Función **duplica** que modifica, duplicando los valores almacenados en los dos miembros del parámetro formal de tipo **\*punto**. La función debe devolver además un dato de tipo **punto** con el resultado de la operación.
5. Declarar un tipo **struct** denominado **persona** que permita almacenar el nombre de una persona (con un máximo de 50 caracteres) y tres datos de tipo numérico real (**float**). Declarar una variable **grupo** que pueda almacenar N elementos de tipo **persona**.

## Soluciones a los ejercicios propuestos del capítulo de Estructuras

1. La salida por pantalla es:

```
Nº de matricula: 19000
Nombre: Pedro Gomez
Nota: 7.860000
Tamaño de a: 48
```

2. Programa con estructuras que representan puntos en el espacio unidimensional

Ejemplo 12.6: Estructuras de puntos en el espacio 2D

```

1 #include <stdio.h>
2 #include <math.h>
3
4 struct punto
5 {
6     float x;
7     float y;
8 };
9
10 int main()
11 {
```

```

12     struct punto a = {7.58, 3.63};
13     struct punto b;
14     float d;
15
16     b.x = 2.06;
17     b.y = -4.15;
18     printf("A se situa en x = %f e y = %f \n", a.x, a.y);
19     printf("B se situa en x = %f e y = %f \n", b.x, b.y);
20     d = sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2));
21     printf("La distancia entre A y B es %f\n", d);
22     return 0;
23 }
```

---

Salida por pantalla en la ejecución:

```
A se situa en x = 7.580000 e y = 3.630000
B se situa en x = 2.060000 e y = -4.150000
La distancia entre A y B es 9.539330
```

3. La salida por pantalla es:

```
A se situa en x = 5.090000 e y = 2.360000
B se situa en x = 7.220000 e y = -3.890000
La distancia entre A y B es 6.602984
```

4. Programa con función `inicializa` y `duplica`:

Ejemplo 12.7: Funciones con punteros a estructuras

---

```

1
2 #include <stdio.h>
3
4 struct Punto
5 {
6     float x;
7     float y;
8 };
9
10 void inicializa (struct Punto *p)
11 {
12     (*p).x = 0;
13     (*p).y = 0;
14 }
15
16 struct Punto duplica (struct Punto *p)
17 {
18     (*p).x *= 2;
19     (*p).y *= 2;
20     return (*p);
21 }
22
23 void muestra (struct Punto p)
24 {
25     printf("Se situa en x = %f", p.x);
26     printf(" e y = %f \n", p.y);
27 }
28
29 int main()
30 {
31     struct Punto a = {7.58, 3.63};
32     struct Punto b;
33     printf("A: ");
```

```

34     muestra(a);
35     inicializa(&b);
36     printf("B: ");
37     muestra(b);
38     b = duplica(&a);
39     printf("A: ");
40     muestra(a);
41     printf("B: ");
42     muestra(b);
43     return 0;
44 }
```

---

Salida por pantalla en la ejecución:

```

A: Se situa en x = 7.580000 e y = 3.630000
B: Se situa en x = 0.000000 e y = 0.000000
A: Se situa en x = 15.160000 e y = 7.260000
B: Se situa en x = 15.160000 e y = 7.260000
```

5. Declaración de un array de **struct**:

Ejemplo 12.8: Array de struct

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 #define N 10
6
7 /* Declaracion del tipo struct */
8 struct persona
9 {
10     char nombre[50];
11     float dato1;
12     float dato2;
13     float dato3;
14 };
15
16 int main()
17 {
18     /* Declaracion de array de N personas */
19     struct persona grupo[N];
20     strcpy(grupo[0].nombre,"Antonio Lopez");
21     grupo[0].dato1 = 23.45;
22     grupo[0].dato2 = -4.8;
23     grupo[0].dato3 = 10.374;
24     strcpy(grupo[1].nombre,"David Ramirez");
25     grupo[1].dato1 = 12.54;
26     grupo[1].dato2 = -14.58;
27     grupo[1].dato3 = 0.134;
28     /* Continua... */
29     return 0;
30 }
```

---

## Otros ejercicios propuestos del capítulo de Estructuras

- Construir un programa que declare y use dos variables **struct** cada una de las cuales represente un punto en el espacio tridimensional. El identificador del nuevo tipo **struct** debe ser de tipo **punto3D**. Los miembros del tipo **struct** deben ser de tipo **float**. El programa debe asignar valores a las respectivas coordenadas de los puntos y calcular la distancia geométrica entre ellos.

2. Construir un programa que pida la fecha actual y la fecha de nacimiento de una persona y calcule y devuelva la edad de la persona descompuesta en años, meses y días. Para facilitar el cálculo se puede considerar que todos los meses del año tienen 31 días. Nota: Debe emplearse alguna estructura de tipo **struct**.

## Capítulo 13

# Archivos y Canales de Datos Estándar

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Definir el concepto de archivo y describir sus características fundamentales y los diferentes tipos de archivos (Conocimiento)
2. Comprender el concepto de canal de datos y describir sus usos más comunes (Conocimiento)
3. Entender y utilizar los conceptos de canal de entrada, salida y errores de un programa (Aplicación)
4. Describir las operaciones típicas y las rutinas predefinidas correspondientes que se emplean en C para la manipulación de archivos (Comprensión)
5. Interpretar el resultado de la ejecución de un programa que emplea archivos (Comprensión)
6. Determinar los formatos adecuados para almacenar los datos en un archivo en función de las necesidades de una aplicación (Aplicación)
7. Codificar una tarea sencilla convenientemente especificada que emplea archivos (Aplicación)

### 13.1. Introducción

Como se ha indicado anteriormente una *estructura de datos*, en general, es un conjunto de datos más simples. Las estructuras de datos pueden clasificarse según se realice su almacenamiento en:

1. *internas*, que son estructuras de datos que se almacenan en memoria durante la ejecución del programa. Por ejemplo, variables de tipo *array* (ó vector), cadenas alfanumérica (*char* [ ]) o *struct*.
2. *externas*, que son estructuras que se almacenan en dispositivos de almacenamiento masivo o memoria periférica como puede ser el disco duro del ordenador. La forma habitual de almacenar datos en un disco es mediante un *archivo* o *fichero*.

Un *archivo* o *fichero* es una estructura o colección secuencial de datos de tamaño variable que puede guardarse de forma permanente en un sistema de almacenamiento masivo. Las principales características de los archivos de disco como estructura de datos son las siguientes:

1. un archivo de disco es un almacén de datos *no volátil*,
2. *no* tiene un *tamaño predeterminado*: el tamaño de un archivo puede ir variando durante la ejecución de un programa según las necesidades. El tamaño está sólo limitado por el espacio de almacenamiento existente en disco y,
3. en la mayoría de los archivos los datos están almacenados en un *formato* determinado, que no tiene porqué coincidir con el formato empleado en otros archivos.

El uso de archivos de disco como almacén de datos tiene las siguientes ventajas:

1. la posibilidad de manejar *grandes* cantidades de datos (tanto de entrada como de salida),

2. los datos no se pierden al terminar de ejecutar un programa o al apagar el ordenador, es decir, pueden quedar almacenados *permanentemente* y
3. su empleo facilita la transmisión y el *intercambio* de datos entre programas, aplicaciones, ordenadores o sistemas diferentes. Esta ventaja afecta especialmente a los archivos de texto.

Aunque su empleo también tiene algunos inconvenientes, tanto en la construcción como durante la ejecución de un programa, así:

1. la *manipulación* de un archivo en un programa es algo más *complicada* que el de una estructura de datos interna y
2. las *operaciones* de entrada y salida (el acceso para escritura y lectura en los archivos en disco) son relativamente más *lentas*.

## 13.2. Estructura y tipos de archivos

Un archivo es una estructura de datos que consiste en una secuencia de *bytes* (estos archivos reciben habitualmente el nombre de archivos *binarios* u *octec streams*) o de datos de tipo **char** ó caracteres de texto (que toman el nombre de archivos *de texto*)<sup>1</sup>. Sobre esta secuencia simple de datos se añade cierta codificación propia de cada archivo que sirve para interpretar los datos elementales. Esta codificación recibe el nombre de *formato* y se emplea para organizar y dar significado a los datos. Así pues, los archivos se pueden clasificar en dos grandes grupos:

1. *Archivos de texto*: emplean caracteres para representar tanto información textual como datos numéricos. Son muy adecuados para el intercambio de información entre programas y aplicaciones.
2. *Archivos binarios*: los datos se almacenan con el mismo formato (codificación binaria) que en memoria. Son más eficientes desde el punto de vista del almacenamiento de los datos (ideales para grandes cantidades de información y bases de datos) ya que no es necesaria la traducción de formatos al realizar transferencias de información con el programa.

En cualquiera de los casos la estructura de los archivos utilizados en un programa sigue de forma general el esquema de la figura 13.1. Por ejemplo, en el caso de archivos de texto, cada uno de estos datos se interpreta como un carácter alfanumérico. En el caso de archivos binarios, los bits o *bytes* se interpretarán de acuerdo al formato empleado. Este formato debe especificar, entre otras cosas, cuál es la codificación y longitud en *bytes* de cada uno de los datos que se guardan en el archivo.



Figura 13.1: Esquema de la estructura de un archivo

Cuando se intenta acceder a algún dato más allá del final del archivo se genera un error o una señal, *eof* (*end of file*), que es la *señal de fin de archivo*. El acceso, tanto de entrada como de salida, a los datos del archivo se permite a través de una *localización* o *ventana* asociada a cada variable archivo. Esta localización recibe el nombre de puntero o cursor de la variable archivo. Es fundamental tener claro que este puntero o cursor indica la posición en la que se va a realizar el siguiente acceso de lectura o de escritura en el archivo en disco correspondiente. El acceso al contenido de los archivos de texto habitualmente es secuencial. Se pueden realizar saltos del puntero o cursor de lectura - escritura, pero sólo a posiciones conocidas o al inicio del archivo.

Los archivos de texto son ideales para intercambiar datos entre aplicaciones y se pueden crear, ver y modificar con cualquier editor de texto como, por ejemplo, el Bloc de notas o Notepad de los sistemas operativos de la familia Windows o cualquier editor de textos en Linux o Unix.

## 13.3. Archivos y canales de datos

Aunque los archivos son bloques de datos tienen como característica fundamental ser secuenciales. Esta característica los incluye dentro de un concepto abstracto que se usa para describir datos secuenciales independientemente del sitio donde se guarden o su procedencia. Este concepto es el canal, en inglés *stream*.

<sup>1</sup>Como un **char** ocupa un *byte*, la diferencia entre un tipo y otro es, simplemente, la forma de interpretar el contenido del archivo.

Un canal es la abstracción de cualquier flujo de datos secuencial. Un canal o flujo de datos produce una serie de datos, más concretamente *bytes*, de forma secuencial, es decir, ordenada, uno detrás de otro. Los *bytes* pueden proceder de un archivo, un puerto de comunicaciones u otro programa, este detalle no resulta relevante para poder usar el flujo de datos. Existen canales de datos que disponen de la información de toda la secuencia en cualquier momento, por ejemplo, los archivos. Pero otros canales, especialmente los que provienen de comunicaciones (puertos serie, comunicaciones por Internet) pueden no tener toda la información en todo momento. Normalmente las operaciones que acceden a los datos

La librería estándar de C implementa una serie de funciones para manejar archivos. El diseño de estas funciones utiliza el concepto de canal o flujo de datos. Por otro lado, los programas escritos en C utilizan 3 canales de datos estándares: el canal de entrada estándar o **stdin**, el de salida o **stdout** y el de errores o **stderr**. Estos canales tienen en C el tipo de otros archivos. De hecho, en C, la diferencia entre canal de datos y los archivos es difusa y a menudo resultan intercambiables. Si no se indica otra cosa el canal de entrada estándar es el teclado y la salida y el canal de errores son la pantalla.

## 13.4. Variables de tipo puntero a archivo en C

Para poder trabajar y manipular datos con archivos de disco dentro de un programa escrito en C es necesario utilizar una variable de un nuevo tipo de dato: el tipo *puntero a archivo*. Este tipo de dato se considera y maneja como una secuencia lineal de componentes. Como ya se detallará más adelante, mediante una variable de tipo puntero a archivo, que hace las veces de intermediario, se podrán crear o eliminar archivos en el disco, introducir (escribir) o sacar (leer) datos... durante la ejecución de un programa (figura 13.2).

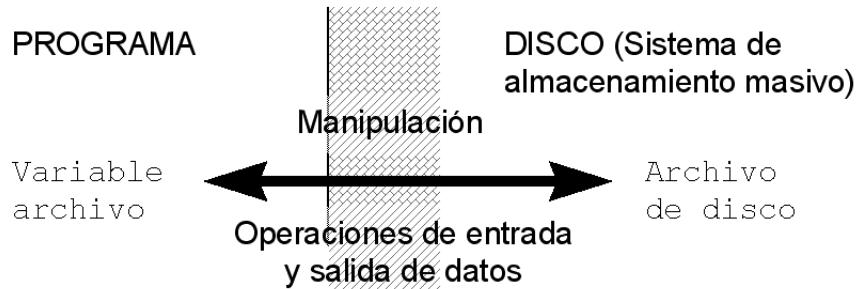


Figura 13.2: Utilización de variables puntero a archivo para la manipulación de archivos de disco

## 13.5. Declaración de variables puntero a archivo

Para trabajar con una estructura de datos de tipo archivo en un programa en C es necesario declarar previamente una variable de tipo archivo. Las variables de tipo archivo siempre se declaran como *punteros a archivo* y servirán para identificar en el programa fuente el archivo en disco con el que se va a trabajar.

```
FILE *f;
```

**FILE** es el identificador de un tipo de dato **struct** predefinido en el archivo de cabeceras de la biblioteca estándar de C **stdio.h**. La estructura **FILE** contiene información sobre el modo o clase de archivo, el *buffer* del archivo..., pero para poder realizar operaciones con archivos no es necesario conocer la declaración concreta de **FILE**. El manejo de los archivos siempre se realiza a través de las funciones de la librería estándar de C correspondiente.

Por otro lado, en el código fuente de un programa se pueden declarar varias variables puntero a archivo, cada una con su propio identificador, para trabajar con diferentes archivos en disco simultáneamente. Además, los programas en C que incluyan **stdio.h** también pueden utilizar los identificadores: **stdin**, **stdout** y **stderr** para usar respectivamente la entrada, salida y canal de errores estándares del programa. Todos estos identificadores son de tipo **FILE\*** como los archivos.

## 13.6. Asociación de variables y archivos

### 13.6.1. Apertura de archivos

Una vez declarada la variable puntero a archivo se le asocia el nombre de un archivo en disco a la vez que se le indica al sistema operativo que se va a empezar a trabajar con él. Esta operación recibe el nombre de *apertura* del archivo. El sistema operativo localiza el archivo en el dispositivo correspondiente, lo marca como archivo *en uso* y reserva un espacio (almacén temporal *buffer* del archivo) en la memoria para las posteriores operaciones de lectura y escritura de datos. La apertura del archivo se realiza mediante la función `fopen` que devuelve como valor de retorno un puntero al archivo abierto cuyo nombre se indica como primer parámetro de la llamada. El *modo* de uso se indica como segundo parámetro. El código de llamada a `fopen` tiene el siguiente aspecto:

```
f = fopen("nombre", "modo");
```

El parámetro **nombre** es una cadena que representa la ruta o el identificador del archivo externo asociado. Debe ser admisible por el sistema operativo correspondiente y no tiene porqué coincidir con el de un archivo ya existente en disco. Por ejemplo, si se trabaja en el sistema operativo DOS, el nombre del archivo puede tener un máximo de ocho caracteres y una extensión de tres caracteres. Tanto en DOS como en Windows en caso necesario puede (o debe) indicarse la unidad de disco (por ejemplo: A:, B:, C:) y el camino o vía de acceso (\codigos\, \datos\asignaturaC\,...). La secuencia \\ es traducida por el compilador de C por una única \ al generar la cadena de caracteres. Por ejemplo, si se desea leer el contenido de un archivo de texto **numeros.txt** situado en el directorio C:\MisDatos de un sistema Windows la llamada a la función sería:

```
f = fopen("C:\\MisDatos\\numeros.txt", "r");2
```

En el caso de que fuera un sistema tipo Unix o GNU-Linux, la llamada sería:

```
f = fopen("/home/MisDatos/numeros.txt", "r");3
```

En ambos sistemas las rutas con espacios, con letras no incluidas en el alfabeto inglés o símbolos especiales son posibles, pero pueden causar problemas imprevistos en cualquier momento. Por eso se recomienda usar siempre nombres sin espacios, símbolos especiales o letras que no pertenezcan al alfabeto inglés. El parámetro **modo** indica la forma de utilización del archivo según indica la tabla 13.1.

Cuadro 13.1: Algunos modos de uso de un archivo

Valor	Uso
r	<i>read only</i> - sólo lectura del archivo ya existente en disco. El cursor o puntero se sitúa al inicio del archivo.
w	<i>write</i> - sólo escritura desde el inicio del nuevo archivo. Si el archivo ya existe elimina su contenido y si no existía previamente crea uno nuevo.
a	<i>append</i> - sólo escritura/añadir datos al final del archivo ya existente. Si el archivo ya existe no elimina su contenido y si no existía previamente crea uno nuevo.
rb	sólo lectura (archivo binario)
wb	escritura desde el comienzo del archivo (binario)
ab	escritura de datos al final del archivo (binario)

Después de ejecutar con éxito la llamada a la función `fopen`, cualquier operación que se haga con la variable puntero a archivo, en la práctica se realizará en el archivo externo asociado. La llamada a `fopen` supone una reserva de memoria para la variable archivo que se libera al cerrar el archivo (ver más adelante). El puntero devuelto por la función `fopen` será NULL si, por alguna razón, no se ha conseguido abrir el archivo en la forma deseada (por ejemplo, al tratar de abrir un archivo que no existe para lectura). Es importante comprobar que no se ha producido ningún error en la operación de apertura del archivo ya que, si falla la ejecución de la llamada a `fopen`, no se pueden realizar el resto de operaciones sobre el archivo.

En el ejemplo 13.1 se muestra un programa que usa la función `fopen`.

<sup>2</sup>Windows también reconoce el carácter barra (\) como separador de carpetas en la ruta de un archivo, pero no distingue mayúsculas de minúsculas y algunas veces tiene problemas con los archivos sin extensión.

<sup>3</sup>Los sistemas tipo UNIX solo admiten la barra (/) como separador de carpetas y distingue mayúsculas de minúsculas en los nombres de los archivos. Es indiferente que los archivos tengan o no extensión.

## Ejemplo 13.1: Uso de fopen

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     FILE *f = fopen("datos.txt", "r");
7     if (f == NULL) {
8         printf("Error al abrir datos.txt\n");
9     }
10    else {
11        printf("Archivo abierto\n");
12        printf("Procesando datos.....\n");
13        printf(".....\n");
14        printf("Cerrando archivo...\n");
15        fclose(f);
16        printf("Archivo datos.txt cerrado.\n");
17    }
18    return 0;
19 }
```

---

Salida del programa:

```

Archivo abierto
Procesando datos.....
.....
Cerrando archivo...
Archivo datos.txt cerrado.
```

El nombre del archivo en disco también se puede asignar mediante una variable de tipo cadena alfanumérica como se presenta en el ejemplo 13.2.

## Ejemplo 13.2: Llamada a fopen con cadena alfanumérica

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     FILE *f; char s [30];
7     printf("Introduce el nombre del archivo: \n");
8     scanf("%s", s);
9     printf("Procesando archivo de nombre '%s' \n", s);
10    f = fopen(s, "r"); /* Equivalentes a la sentencia: */
11    if (f == NULL) { /* if (f = fopen(s, "r+"))== NULL) { */
12        printf("Error al abrir el archivo\n");
13    }
14    else {
15        printf("Procesando datos.....\n");
16        printf("Cerrando archivo...\n");
17        fclose(f);
18        printf("Archivo cerrado.\n");
19    }
20    return 0;
21 }
```

---

Salida del programa:

```

Introduce el nombre del archivo:
Procesando archivo de nombre 'datos.txt'
Procesando datos.....
Cerrando archivo...
Archivo cerrado.
```

### 13.6.2. Cierre de archivos

La función **fclose**, incluida dentro de la librería de C **<stdio.h>**, cierra el archivo asociado a una variable puntero a **FILE** (archivo) y libera los recursos asociados a él por el sistema operativo. Es decir, al ejecutar la función **fclose** se libera la variable asociada al archivo y el archivo en disco deja de estar *en uso*. Además devuelve 0 si el archivo se cerró con éxito y -1 (la constante **EOF** predefinida en C) en caso contrario.

Todo archivo abierto con **fopen** debe cerrarse con **fclose**. Salvo algunas excepciones, si durante la ejecución de una función se abre un archivo es altamente recomendable cerrar el archivo antes de terminar la ejecución de la función. Además puede ser conveniente verificar que el archivo se ha cerrado correctamente como muestra el ejemplo 13.3.

Ejemplo 13.3: Cierre de archivo con **fclose**

---

```

1  #include <stdio.h>
2
3
4  int main()
5  {
6      FILE *f;
7      int i;
8
9      f = fopen("datos.txt", "w");
10     if ( f == NULL) {
11         printf("Error al abrir datos.txt\n");
12     }
13     else {
14         printf("Archivo abierto\n");
15         printf("Escribiendo muchos datos.....\n");
16         printf("Cerrando archivo...\n");
17         i = fclose(f);          /* Equivalentes a: */
18         if ( i!=0 ) {           /* if (fclose(f)!=0) { */
19             printf("Error al cerrar archivo\n");
20         }
21         else {
22             printf("Archivo datos.txt cerrado.\n");
23         }
24     }
25     return 0;
26 }
```

---

Salida del programa:

```

Archivo abierto
Escribiendo muchos datos.....
Cerrando archivo...
Archivo datos.txt cerrado.
```

Una vez cerrado un archivo ya no se pueden realizar operaciones durante la ejecución del programa si antes no se vuelve a abrir otra vez. Tampoco se debe cerrar un archivo que previamente no se haya abierto.

En la mayoría de los casos, el sistema operativo cierra todos los archivos empleados al finalizar el programa, pero es una buena práctica acostumbrarse a cerrar de forma explícita los archivos, especialmente si existe la posibilidad de que puedan volver a ser abiertos por el mismo o por otro programa.

### 13.7. Uso de los canales estándares

Las operaciones de apertura y cierre de los canales de datos: **stdin**, **stdout** y **stderr** las añade directamente el compilador sin necesidad de ninguna sentencia especial en el código fuente del programa. Sin embargo si es posible utilizar algunas utilidades de los sistemas operativos para modificar la naturaleza de estos canales.

Las dos modificaciones más usuales son el direccionamiento de un canal a un archivo y la definición de la salida de un programa como la entrada de otro (hacer una tubería o *pipeline*). Ambas utilidades suelen formar parte de todos los sistemas operativos actuales (incluidos, naturalmente, Windows y GNU-Linux).

### 13.7.1. Direccionamiento a un archivo

El direccionamiento de la entrada de un programa a un archivo significa que el programa no va a obtener los datos que se pidan en la función `scanf` en el teclado sino en un archivo de texto. En todo es equivalente a que el usuario teclee el contenido del archivo en el momento de la ejecución del programa. Por supuesto al dirigir la entrada a un archivo la pulsación de cualquier tecla en el teclado es ignorada por el programa. Para realizar este cambio se usa el nombre del programa seguido de un símbolo “menor que” ( < ) y el nombre o ruta del archivo. Por ejemplo:

```
programa < datos.txt
```

El direccionamiento de la salida significa que todo aquello que se imprimiría en pantalla mediante `printf` se escribe en el archivo correspondiente. En muchos programas esto significa que dejan de verse los resultados esperados porque, al fin y al cabo, no se está escribiendo nada en pantalla. El direccionamiento de salida se indica mediante uno o dos símbolos “mayor que” ( > o >> ), cuando se usa un único símbolo mayor el archivo se crea o sobre-escribe, cuando se usan dos se añade la salida al final del archivo. El direccionamiento del canal de errores se hace con 2> o 2>> donde el *dos* indica el número de canal, dos para el canal de errores. Por ejemplo:

```
programa > datos.txt
programa 2> errores.txt
programa >> datos.txt 2> errores.txt
```

### 13.7.2. Tuberías

El direccionamiento del canal de un programa al canal de otro programa que se va a ejecutar a la vez recibe el nombre de tubería o *pipeline*. El nombre deriva de la posibilidad de enganchar sucesivamente los canales de varios programas formando una “tubería” de programas. Para indicar esta operación se usa el símbolo barra vertical ( | ), sale con ALT-GR + 1. Por ejemplo:

```
programa1 | programa2
programa1 | programa2 | programa3
```

## 13.8. Escritura y lectura de datos con formato en archivos de texto

Dependiendo de las necesidades existen varias funciones en el archivo de cabeceras `<stdio.h>` de la librería estándar de C que permiten la escritura de datos en archivos de texto y en archivos binarios. El contenido de este capítulo se centrará en las operaciones realizadas con archivos de texto con formato (`fprintf` y `fscanf`). Por ejemplo, en el caso de escritura en este tipo de archivos se suele emplear `fprintf` que permite escribir datos en un archivo con un funcionamiento prácticamente idéntico a `printf` (que visualiza datos por pantalla) aunque también pueden emplearse otras funciones.

### 13.8.1. El carácter eoln y la librería estándar de C

Es frecuente, aunque no obligatorio, que en archivos de texto el *formato* se establezca por líneas que contienen caracteres, cadenas o frases separadas por una marca que indica el fin de la línea: *eoln* (*end of line*). Por ejemplo, este tipo de formato se encuentra habitualmente en tablas de datos (archivos *csv* o *comma separated values*). Para el sistema operativo DOS la marca *eoln* es una secuencia de dos caracteres ASCII: el carácter de retorno de carro o CR (*Carriage Return*, carácter nº 13 del código ASCII) y el carácter de avance de línea o LF (*Line Feed*, ASCII nº 10). Mientras que los sistemas tipo UNIX o GNU-Linux utilizan un único carácter CR.

Para lograr una mejor portabilidad del código C, las funciones de la librería estándar de C en modo texto se implementan de forma que:

1. cuando leen un salto de línea configurado en cualquiera de las posibilidades indicadas anteriormente lo transforman en un único carácter LF (10 o '\n'),
2. al escribir en modo texto transforman cualquier carácter LF (10 o '\n') en la alternativa utilizada por el sistema operativo para representar el salto de línea.

Por ejemplo, la lectura en Windows de dos caracteres CR y LF desde un archivo en modo texto produce un único carácter '\n', mientras que la escritura en un archivo de texto de un único carácter '\n' escribe en el archivo la escritura de dos bytes: CR y LF.

### 13.8.2. Uso de la función fprintf

La función **fprintf**<sup>4</sup> comparte con **printf** la misma forma de escribir los datos y la sintaxis de los especificadores de formato tal y como se describieron en el capítulo de datos simples y en el de cadenas alfanuméricas. La diferencia consiste en donde escriben: **printf** lo hace en el canal de salida estándar (la pantalla), **fprintf** lo hace en un archivo (**FILE\***) que se pasa como primer argumento. Para conocer en detalle la sintaxis de los especificadores de formato véase el apartado correspondiente en la última parte del texto. No obstante aquí indicamos algunos detalles interesantes.

- Debe tenerse en cuenta que la cadena que indica el formato puede ser una variable. Normalmente las llamadas a **printf** tienen como argumento una constante literal, pero esto no es obligatorio.
- La implementación de la función **fprintf** de la librería estándar de C en cada sistema hace la conversión entre el carácter '\n' que es el carácter número 10 en la tabla ASCII al formato del salto de línea nativo en la plataforma correspondiente. Por ejemplo, en Windows se escriben dos caracteres (13 y 10) y en gnu-linux se escribe un único carácter 13.
- Cuando un argumento no se corresponde con el tipo indicado por el especificador de formato el resultado es difícil de predecir. En general, lo que se muestra es el resultado de una conversión forzada entre tipos, pero es difícil determinar la conversión que se está realizado y por tanto el resultado final. Muchos errores en programas se pueden evitar eligiendo especificadores de formato que coincidan estrictamente con el tipo de la variable que se va a mostrar. La única excepción a esta regla es la utilización mezclada de especificadores de formato y variables de tipo **char** y entero, esta mezcla sirve para encontrar el número de una letra en la tabla ASCII o la letra que se corresponde con un número en la misma tabla.

Como se ha indicado el uso de **fprintf** es simplemente añadir la variable de tipo **FILE\*** al principio, por ejemplo, suponiendo que **f** es una variable tipo **FILE\*** que apunta a un archivo ya abierto:

```
fprintf( f, "Hola, soy un numero%d\n", 12);
```

escribe el archivo el número 12 seguido de un salto de línea.

En un archivo de texto se pueden almacenar, además de cadenas de caracteres, datos pertenecientes a otros tipos. En la tabla 13.2 se recopilan los especificadores de formatos empleados con **fprintf** para diversos tipos de datos.

Cuadro 13.2: Formatos correspondiente a diferentes tipos de dato para **fprintf**

Formato	Tipo de datos
%d	int
%f	float y double
%c	char
%s	char[] (cadena de caracteres)

En el ejemplo 13.4 se muestra un programa de escritura de datos numéricos. El programa genera un nuevo archivo llamado *numeros.txt* con una cantidad aleatoria (entre 5 y 15) de números aleatorios enteros (con valores entre 0 y 99). Además visualiza por pantalla los valores almacenados en el archivo de texto puesto que las líneas 14 y 15 escriben lo mismo.

Ejemplo 13.4: Escribe un archivo con números aleatorios

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 int main()
7 {
8     FILE *f; unsigned short n, i, k;
9     srand(time(0));
10    n = 5 + rand() % 10;
11    if ( (f = fopen("numeros.txt", "w")) != NULL ) {
12        for (i=0; i<n; i++) {

```

<sup>4</sup>Otras funciones que escriben en archivo son **fputc**, **fputs** o **fwrite** (para archivos binarios), pero se han dejado fuera del alcance de este texto por brevedad

```

13         k = rand() % 100;
14         fprintf(f, "%d ", k);
15         printf("%d ", k);
16     }
17     printf("\n Archivo numeros.txt creado \n");
18     fclose(f);
19 }
20 else {
21     printf("Error al abrir el archivo.\n");
22 }
23 return 0;
24 }
```

---

Salida por pantalla en la ejecución:

```

78 40 4 75 38 28 33
Archivo numeros.txt creado
```

En el ejemplo anterior es importante destacar que los datos numéricos enteros se almacenan en el archivo convertidos en caracteres de texto y se separan entre sí por un espacio en blanco para facilitar su posterior lectura. El archivo generado es perfectamente legible por cualquier editor de texto.

### 13.8.3. Uso de la función fscanf

La función **fscanf**<sup>5</sup> comparte con **scanf** la forma de leer los datos y la sintaxis de los especificadores de formato tal y como se describieron en el capítulo de datos simples y cadenas. La diferencia consiste en donde leen: **scanf** lo hace en el canal de entrada estándar (el teclado), **fscanf** lo hace en un archivo (**FILE\***) que se pasa como primer argumento.

La función **scanf** es un ejemplo frecuente de cómo se usan parámetros de tipo puntero en muchas funciones de la librería estándar (y de otras librerías). Dado que en C el paso de parámetros siempre es por valor, no es posible cambiar el valor de un argumento dentro de una función... excepto que pasemos como argumento el puntero de una variable. En este caso, la variable se puede modificar porque la variable no es el argumento, el argumento es su puntero y ese no cambia.

Por tanto para conocer en detalle cómo funciona la función **scanf** es necesario conocer cómo se produce el paso de variables a una función mediante su puntero correspondiente. Esta es la base para entender **scanf**. No obstante, hay dos puntos que tenemos que tener muy claros en el uso de **fscanf** y que no aparecen en el paso de otros argumentos de tipo puntero.

1. La función **fscanf** no puede comprobar si hemos pasado el puntero a una variable válida. La función recibe un puntero que debe apuntar a una variable cuyo tipo debe coincidir completamente con el tipo indicado por el especificador de formato. Esta obligación corre a cuenta del programador que utiliza la función **fscanf**, su incumplimiento puede suponer errores muy graves de funcionamiento del programa. Insistimos que esta obligación tiene dos partes: la variable debe existir y la variable tiene que ser del tipo indicado por el especificador de formato<sup>6</sup>.
2. Las variables de tipo vector, entre ellas especialmente las cadenas alfanuméricas, son convertibles directamente a puntero (sin poner **&**), pero cuando se hace la conversión a puntero deja de conocerse su tamaño. Es un error muy frecuente no asegurar que la variable de tipo cadena alfanumérica es capaz de almacenar la información que va recibir. La capacidad de la cadena debe ser mayor que el número de caracteres leídos. Es una muy buena práctica utilizar (**obligatoria**<sup>7</sup>), para la lectura de cadena alfanuméricas, el modificador que afecta al número de caracteres leídos tal y como se ve en el siguiente ejemplo:

```

char aux[100];
scanf("%99s", aux);
```

Observe que el número que aparece entre el carácter **%** y el carácter **s** es 99, uno menos que la capacidad de la cadena. La razón es que **fscanf** debe almacenar también el carácter nulo y, por tanto, solo puede leer del archivo 99 caracteres no 100.

<sup>5</sup>Otras funciones que escriben en archivo son **fgetc** o **fread** (para archivos binarios), pero se han dejado fuera del alcance de este texto por brevedad

<sup>6</sup>Algunos compiladores dan avisos si el programador no tiene en cuenta esta limitación

<sup>7</sup>Entendemos que es tan buena práctica que la consideramos obligatoria para los alumnos de la asignatura de la ETSII - UPM

También es importante conocer los siguientes detalles relacionados con la entrada de datos:

- La implementación de la función **fscanf** de la librería estándar de C en cada sistema hace la conversión desde el salto de línea nativo en la plataforma correspondiente al carácter '**\n**' que es el carácter número 10 en la tabla ASCII. Por ejemplo, en Windows se leen dos caracteres (13 y 10) y se transforman en una única letra '**\n**' y en gnu-linux se lee un único carácter 13, pero se obtiene un carácter 10.
- Cuando se leen varios datos seguidos se debe tener en cuenta que:
  1. Los números se terminan de leer cuando se encuentra un carácter que no pueda formar parte del número, por ejemplo, una letra o un signo de puntuación distinto del punto decimal.
  2. En los números en coma flotante la parte entero y fraccionaria se separa mediante un **punto no una coma**.
  3. Las cadenas alfanuméricas se terminan de leer cuando se encuentra un espacio en blanco de cualquier tipo (barra espaciadora, tabulador, salto de línea).
  4. Si se escribe un espacio en blanco (barra espaciadora, tabulador, salto de línea en el formato que se pasa como primer argumento al **fscanf**) se saltan todos los espacios en blanco que aparezcan en la entrada hasta encontrar un carácter distinto de espacio en blanco. Solo es necesario poner un único espacio en blanco en el formato para saltar cualquier número de espacios en la entrada, incluido ningún espacio.
  5. Si se escribe algún carácter distinto de un especificador de formato o un espacio en blanco, **fscanf** busca que ese mismo carácter aparezca en la entrada, para a continuación descartarlo, pero si no lo encuentra no continua procesando la entrada.
- El retorno de la función indica el número de datos (no de caracteres) leídos correctamente, es decir, se han leído de la entrada y se han podido convertir al tipo indicado por el especificador. Puede ocurrir que se produzcan errores cuando se intentan leer varios datos y en este caso el retorno indicará el número de datos que se han leído correctamente. Cuando se usa **fscanf** se puede llevar al final de archivo y se retorna un valor EOF. Es posible provocar el final de archivo de la entrada por teclado, se pueden usar las combinaciones de teclas **control + z** o **control + d** para lograrlo, pero no es frecuente hacer esto. Normalmente el valor del retorno se comprueba frente al número de datos que se desea leer, si coinciden significa que no se han producido errores.

En la tabla 13.3 se recopilan los especificadores de formatos empleados con **fscanf** para diversos tipos de datos. Es importante notar que la diferencia más importante respecto de **fprintf** es el tratamiento de los datos de tipo coma flotante.

Cuadro 13.3: Formatos correspondiente a diferentes tipos de dato para scanf

Formato	Tipo de datos
%d	int
%f	float
%lf	double
%c	char
%99s	char [100] (cadena de caracteres). Para cualquier otro tamaño de cadena reste 1 y escriba el resultado entre el % y la s

Como se ha indicado el uso de **fscanf** respecto de **scanf** es simplemente añadir la variable de tipo **FILE\*** al principio, por ejemplo, suponiendo que **f** es una variable tipo **FILE\*** que apunta a un archivo ya abierto y **num** es una variable **int** el siguiente ejemplo lee del archivo un número entero.

```
fscanf( f, "%d", &num);
```

#### 13.8.4. Lectura de datos de un archivo de texto con fscanf

Es importante considerar que cada vez que se realiza una operación de lectura de un archivo el cursor, es decir, la localización del punto de lectura o escritura sobre el archivo, se mueve hasta colocarse en el carácter que aparece después del dato leído. Este comportamiento permite llamadas sucesivas a **fscanf** donde cada de ellas lee nueva información.

Cada vez que se hace una operación de lectura es conveniente comprobar si ésta se ha llevado a cabo con éxito. Los motivos para que un proceso de lectura no dé como resultado el dato al que apunta el puntero asociado pueden

ser dos: se ha producido un error en la lectura o se ha llegado al final del archivo. Las funciones de lectura cuando detectan alguna de estas situaciones generalmente devuelven la macro `EOF` o algún otro código de error<sup>8</sup>.

El proceso de lectura de un archivo en C puede ser muy complejo. Normalmente se implementa mediante un bucle que se repite hasta que se alcanza el fin de archivo. Es importante notar que solo el intento de leer más allá del final del archivo provoca el indicador `EOF`. Esto hace necesario comprobar el resultado de cada una de las funciones de lectura. Si, además, no se puede suponer que el formato del archivo es el correcto, también es necesario comprobar si se ha producido algún error en el proceso de lectura. A continuación se muestra el aspecto de un bucle `while` que se puede emplear para leer un archivo y cuyo estilo resulta sencillo y aplicable a muchos casos.

```
int fin = fscanf(f, "formato", &x1, ...);
while ( fin != EOF ) {
    /* Proceso de los datos */

    /* Nueva lectura */
    fin = fscanf(f, "formato", &x1, ...);
}
```

También se puede emplear el valor devuelto por la función `fscanf` (el número de datos leídos y almacenados en los argumentos correspondientes) para detectar cuando no quedan más datos por leer. En el ejemplo 13.5 se muestra un programa que usa esta propiedad de la función `fscanf` para leer carácter a carácter el contenido del archivo hasta que se llega al final del mismo.

Ejemplo 13.5: Lectura de archivo

---

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     FILE *f; char c; int i = 0, aux;
7
8     if ( (f = fopen("datos.txt", "r")) != NULL ) {
9         do {
10             aux = fscanf(f, "%c", &c);
11             if ( aux == 1 ) {
12                 printf("%c", c);
13                 i++;
14             }
15         } while (aux==1);
16         fclose(f);
17         printf("El numero de caracteres es %i.\n", i);
18     }
19     else {
20         printf("Error al abrir el archivo.\n");
21     }
22     return 0;
23 }/*main*/
```

---

Salida por pantalla en la ejecución:

```
Primer texto de ejemplo
N: 1234 fg af
Eneros: 002345; 2345; +2345;2345 ;
N: 4 fg zf
Reales: 0.333333; 0.33333333333333; +0.33;0.3333333333 ;
N 2 fg zf
Cadenas: Hola Mundo; Hola Mun;Hola Mun ;
N: 2 3a ff
El numero de caracteres es 216.
```

<sup>8</sup>Se puede utilizar la función estándar `ferror` para detectar si ha existido algún error en la ejecución de la lectura del archivo. La función `ferror` devuelve cero si no ha habido error y un valor distinto de cero si ha habido error.

El contenido del archivo datos.txt es:

```
Primer texto de ejemplo
N: 1234 fg af
Eteros: 002345; 2345; +2345;2345 ;
N: 4 fg zf
Reales: 0.333333; 0.33333333333333; +0.33;0.3333333333 ;
N 2 fg zf
Cadenas: Hola Mundo; Hola Mun;Hola Mun ;
N: 2 3a ff
```

En el ejemplo 13.6 se muestra otra forma de lectura de archivo: se abre el archivo **datos.txt** generado en un ejemplo anterior, se va leyendo el contenido del mismo línea a línea mediante la función **fscanf**, con un especificador de formato que obtiene un número entero y dos cadenas. Todo ello mientras que se puedan leer los datos correspondientes. En consecuencia, al alcanzar la línea donde la '**N**' ya no aparece seguida de '**:**', el bucle se interrumpe y se acaba la lectura, solo se leen las dos primeras líneas tal y como se puede apreciar en la salida por pantalla.

Ejemplo 13.6: Lee un archivo escrito previamente

---

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     FILE *f;
7     char aux1[3], aux2[3];
8     int i;
9
10    if ( (f = fopen("datos.txt","r")) != NULL )
11    {
12        /* Lee una N, el caracter ':', un entero,
13         * dos cadenas de 2 caracteres maximo */
14        while ( 3 == fscanf(f, " N: %i %2s %2s", &i, aux1, aux2 ) )
15        {
16            printf("%i %s %s\n", i, aux1, aux2);
17        }
18        fclose(f);
19        printf("Archivo de texto datos.txt cerrado.\n");
20    }
21    else
22    {
23        printf("Error al abrir el archivo.\n");
24    }
25    printf("Fin del programa\n");
26    return 0;
27 }
```

---

La salida por pantalla al ejecutar el programa es:

```
Archivo de texto datos.txt cerrado.
Fin del programa
```

### 13.8.5. Uso de la función fgets

Para leer un archivo de texto línea a línea una buena opción es usar la función **fgets**. Su uso es parecido a la función **gets**, pero presenta algunas diferencias importantes.

La función **fgets** aparece declarada en *stdio.h* con el siguiente prototipo simplificado:

```
char *fgets(char * cad, int n, FILE * stream);
```

Donde **cad** es la cadena alfanumérica donde se va a almacenar los caracteres de la línea, **n** es el tamaño de la cadena y **stream** es el archivo. El retorno de la función es la misma cadena (**cad**) o **NULL** si no se puede leer del archivo. El carácter salto de línea se almacena en la cadena y, por supuesto, también el carácter nulo para terminar la cadena. Para usar **fgets** normalmente se usa un bucle estructura como aparece a continuación:

```

FILE *f; char line[1024], *aux;
int line_number = 0;
/* Abrir archivo f */
aux = fgets(line, 1024, f);
while ( aux != NULL ) {
    ++line_number;
    /* Proceso de la linea */

    /* Nueva lectura linea */
    aux = fgets(line, 1024, f);
}
printf("Se han leido %i lineas\n", line_number);

```

Una de las principales ventajas del bucle anterior frente a otras formas de leer un archivo es que se puede identificar el número de línea donde se produce un error. Observe que se usa un tamaño de línea suficientemente grande para almacenar líneas muy grandes.

La cadena que se obtiene al leer una línea se puede procesar de muy distintas maneras. La más básica es carácter a carácter, pero se pueden hacer un procesamiento más avanzado mediante otras funciones como `sscanf` o `strtok`<sup>9</sup>.

Ejemplo 13.7: Lee un archivo línea a línea

---

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     FILE *f; char line[1024], *aux; int line_number = 0;
7
8     if ( (f = fopen("datos.txt","r")) != NULL )
9     {
10         aux = fgets(line, 1024, f);
11         while ( aux != NULL ) {
12             ++line_number;
13             /* Proceso de la linea */
14             if ( line[0] == 'N' && line[1] == ':' && line[2] == ' ' )
15             {
16                 printf("N: %i\n", line[3] - '0');
17             }
18             else
19             {
20                 printf("Error de formato en linea %i \n", line_number);
21             }
22             /* Nueva lectura linea */
23             aux = fgets(line, 1024, f);
24         }
25
26         fclose(f);
27         printf("Se han leido %i lineas\n", line_number);
28         printf("Archivo de texto datos.txt cerrado.\n");
29     }
30     else
31     {
32         printf("Error al abrir el archivo.\n");
33     }
34     printf("Fin del programa\n");
35     return 0;
36 }

```

---

La salida por pantalla al ejecutar el programa es:

Error de formato en linea 1

<sup>9</sup>Los detalles de las funciones `sscanf` y `strtok` se pueden encontrar en la parte de material adicional

```

N: 1
Error de formato en linea 3
N: 4
Error de formato en linea 5
Error de formato en linea 6
Error de formato en linea 7
N: 2
Se han leido 8 lineas
Archivo de texto datos.txt cerrado.
Fin del programa

```

En el ejemplo 13.7 se muestra un programa que lee de nuevo el archivo *datos.txt*. En esta lectura se ha considerado correcta una línea si comienza por “N: “, compruebe que líneas cumplen y cuales no. Por otro lado solo se ha extraído como información la primera cifra del número.

### 13.8.6. Datos con formato en archivos

Para facilitar la escritura y posterior lectura de los datos incluidos en un archivo de texto es conveniente hacerlo con un formato que facilite ambos tipos de operaciones y que evite los errores durante la ejecución. Los formatos más sencillos de leer mediante programas en C son, entre otros:

- Columnas formadas por números separados por blancos (tabuladores, espacios) o por algún signo de puntuación (por ejemplo: ‘;’).
- Columnas formadas por cadenas alfanuméricas acabadas en blancos.
- Columnas de números y cadenas preferente todas acabadas en blancos.
- Columnas de ancho fijo, tanto para números como para cadenas. Si un número o cadena ocupa menos caracteres de los especificados resulta más sencillo si se rellena con espacios a la izquierda.

En el ejemplo 13.8 se muestra un programa que introduce datos de distintos tipos (entero, carácter, real y cadena de caracteres) en un archivo de texto con un formato determinado: los agrupa en diferentes líneas y los separa entre sí por caracteres de coma. Observe que la cadena se deja al final de la línea para que el proceso de lectura sea más sencillo (la lectura de una cadena no se puede acabar en coma con especificador %s).

Ejemplo 13.8: Escribe datos de distintos tipos

---

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     FILE *f; char nombre_archivo[20] = "varios.txt";
7     char s[20] = "final"; char c ='A';
8     double x = 24.75; int n = 1;
9
10    printf("Inicio del proceso de escritura.\n");
11    if ( (f = fopen(nombre_archivo,"w")) != NULL ) {
12        while ( n < 20 ) {
13            fprintf(f, "%i,%c,%6.3f,%s\n",n,c,x,s);
14            printf("OK: %i,%c,%6.3f,%s\n",n,c,x,s);
15            n += 5;
16            c++;
17            x = x / n;
18        }
19        fclose(f);
20        printf("Fin del proceso de escritura.\n");
21    }
22    else {
23        printf("Error al abrir el archivo.\n");
24    }
25    printf("Fin del programa.\n");

```

---

```

26     return 0;
27 }/*main*/

```

---

Salida por pantalla al ejecutar el programa:

```

Inicio del proceso de escritura.
OK: 1,A,24.750,final
OK: 6,B, 4.125,final
OK: 11,C, 0.375,final
OK: 16,D, 0.023,final
Fin del proceso de escritura.
Fin del programa.

```

El contenido del archivo de texto generado por el programa ejemplo es:

```

1,A,24.750,final
6,B, 4.125,final
11,C, 0.375,final
16,D, 0.023,final

```

En el ejemplo 13.9 se muestra un programa que lee los datos de distintos tipos (entero, carácter, real y cadena de caracteres) introducidos previamente en un archivo de texto con el ejemplo anterior.

Ejemplo 13.9: Lee datos de distintos tipos

---

```

1 #include <stdio.h>
2
3 int main()
4 {
5     FILE *f; char nombre_archivo[20]= "varios.txt";
6     char s [20]; char c; float x; int n;
7
8     if ( ( f = fopen(nombre_archivo,"r")) != NULL )
9     {
10         while ( fscanf(f,"%i,%c,%f,%s",&n,&c,&x,s) != EOF )
11         {
12             printf("OK: %i,%c,%f,%s\n",n,c,x,s);
13         }
14         printf("Fin del proceso de lectura.\n");
15         fclose(f);
16     }
17     else
18     {
19         printf("Error al abrir el archivo.\n");
20     }
21     printf("Fin del programa.\n");
22     return 0;
23 }

```

---

Salida por pantalla al ejecutar el programa:

```

OK: 1,A,24.750000,final
OK: 6,B,4.125000,final
OK: 11,C,0.375000,final
OK: 16,D,0.023000,final
Fin del proceso de lectura.
Fin del programa.

```

## 13.9. Otras operaciones

En las librerías estándar de C existen muchas otras funciones que permiten implementar otras tantas operaciones con archivos. También existen funciones para acceder a un archivo de modo directo o aleatorio, es decir, no secuencial.

Este acceso directo permite leer y escribir datos en cualquier orden conociendo la posición correspondiente del dato en la secuencia. Son las funciones: `fseek`, `ftell` o `rewind`.

En la tabla 13.4 se muestran otras rutinas que permiten realizar otras operaciones con archivos.

Cuadro 13.4: Otras rutinas para trabajar con archivos

Función	Operación
<code>remove("nombre.ext")</code>	Borra el archivo en disco. Devuelve 0 si la operación se ha llevado a cabo con éxito y un valor distinto de 0 en caso contrario
<code>rename("nombreAntiguo", "nombreNuevo")</code>	Modifica el nombre de un archivo en disco. Devuelve 0 si la operación se ha llevado a cabo con éxito y un valor distinto de 0 en caso contrario (y el archivo conserva su nombre original)

## Ejercicios propuestos del capítulo de Archivos

1. ¿Cuál es número máximo de datos o registros que puede almacenarse en un archivo?
2. ¿Cómo puede conocerse el número total de caracteres que contiene un archivo de texto? ¿y el número de caracteres exceptuando los caracteres de las marcas de fin de línea?
3. Escribir un programa que cuente el número de caracteres de un archivo de texto.
4. Escribir un programa que copie el contenido de un archivo de texto (origen) en otro archivo (destino).
5. Escribir un programa que elimine el archivo "nombre.txt" del disco
6. Escribir un programa que renombre un archivo en disco.
7. Completar el programa del ejemplo 13.7 para que, al leer cada línea, sea capaz de extraer el número y las dos cadenas de caracteres a continuación. Nota: Este ejercicio tiene una solución trivial usando `sscanf` y otra solución mucho más compleja si se tiene que procesar carácter a carácter, intente hacerlo de las dos formas.
8. Escribir un programa que permita realizar las siguientes operaciones con un archivo que contenga una secuencia de números enteros (o de cualquier otro tipo de dato):
  - a) añadir nuevos datos al archivo
  - b) modificar datos ya almacenados
  - c) eliminar datos del archivo
  - d) listar en pantalla todos los datos del archivo.

## Soluciones a los ejercicios propuestos del capítulo de Archivos

1. Un número indefinido con la limitación del espacio libre disponible en el sistema de almacenamiento de datos correspondiente.
2. Puede calcularse a partir del tamaño que ocupa el archivo en el directorio del disco correspondiente o con un programa que cuente el número de caracteres leídos del archivo desde el principio hasta el final.
3. Programa que cuenta el número de caracteres de un archivo de texto. Nota: Dado el comportamiento de la función `fscanf` se ha considerado que el salto de línea es un único carácter en cualquier plataforma. Se proponen dos alternativas:

Ejemplo 13.10: Cuenta caracteres en un archivo de texto

---

```

1
2 #include <stdio.h>
3
```

```

4 int main()
5 {
6     FILE *f; char c; long i = 0;
7     if ( (f = fopen("datos.txt","r")) != NULL ) {
8         while ( fscanf(f,"%c",&c) == 1 ) {
9             printf("%c",c);
10            ++i;
11        }
12        fclose(f);
13        printf("\nEl numero de caracteres es %ld.\n",i);
14    }
15    else {
16        printf("Error al abrir el archivo.\n");
17    }
18    return 0;
19 }
```

---

Salida por pantalla del programa:

```

Primer texto de ejemplo
N: 1234 fg af
Enteros: 002345; 2345; +2345;2345 ;
N: 4 fg zf
Reales: 0.333333; 0.33333333333333; +0.33;0.3333333333 ;
N 2 fg zf
Cadenas: Hola Mundo; Hola Mun;Hola Mun ;
N: 2 3a ff
```

El numero de caracteres es 216.

#### 4. Programa que copia el contenido de un archivo de texto en otro.

Ejemplo 13.11: Copia un archivo de texto en otro

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     FILE *f, *g; char c;
7     printf("Copia un archivo origen en otro destino.\n");
8     if ( (f = fopen("datos.txt","r")) == NULL ) {
9         printf("Error al abrir el archivo origen.\n");
10    }
11    else {
12        if ( (g = fopen("copia.txt","w")) == NULL ) {
13            printf("Error al abrir el archivo destino.\n");
14        }
15        else {
16            while ( fscanf(f, "%c", &c) != EOF ) {
17                fprintf(g,"%c",c);
18                printf("%c",c);
19            }
20            fclose(g);
21            printf("\nCopia finalizada con exito.\n");
22        }
23        fclose(f);
24    }
25    printf("Fin del programa.\n");
26    return 0;
27 }
```

---

Salida por pantalla del programa:

```
Copia un archivo origen en otro destino.
Primer texto de ejemplo
N: 1234 fg af
Enteros: 002345; 2345; +2345;2345 ;
N: 4 fg zf
Reales: 0.333333; 0.33333333333333; +0.33;0.3333333333 ;
N 2 fg zf
Cadenas: Hola Mundo; Hola Mun;Hola Mun ;
N: 2 3a ff

Copia finalizada con exito.
Fin del programa.
```

5. Escribir un programa que elimine el archivo "nombre.txt" del disco

Ejemplo 13.12: Borra un archivo de disco

---

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     char nombreArchivo[] = "nombre.txt";
7     printf("Archivo %s ", nombreArchivo);
8     if (remove(nombreArchivo) == 0 ) {
9         printf("ha sido borrado\n");
10    }
11    else {
12        printf("no ha podido ser borrado\n");
13    }
14    return 0;
15 }
```

---

6. Escribir un programa que renombre un archivo en disco.

Ejemplo 13.13: Renombra un archivo de disco

---

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     char antiguo[] = "antiguo.txt";
7     char nuevo[] = "nuevo.txt";
8     printf("Archivo %s ", antiguo);
9     if (rename(antiguo, nuevo) == 0) {
10         printf("renombrado como %s\n", nuevo);
11     }
12     else {
13         printf("no ha podido ser renombrado\n");
14     }
15     return 0;
16 }
```

---

# Capítulo 14

## Estructuras de datos dinámicas

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

1. Describir las características de las variables dinámicas (Conocimiento).
2. Conocer la forma de gestionar variables dinámicas (Conocimiento).
3. Interpretar un programa que use variables dinámicas (Comprendición).
4. Diseñar los tipos de datos necesarios para utilizar una estructura de datos dinámica (Aplicación).
5. Codificar programas que realicen tareas sencillas empleando estructuras de datos dinámicas (Aplicación).

### 14.1. Variables estáticas y dinámicas

Hasta ahora, en la teoría y ejemplos tratados en este texto, las variables que se han usado han sido *estáticas*. Las variables estáticas tienen una serie de características comunes:

1. Todas las variables estáticas que utiliza un programa o una rutina se declaran explícitamente mediante la correspondiente sentencia declarativa.
2. Las variables estáticas simples, las compuestas y las estructuras de datos estáticas definen su tipo y tamaño durante la compilación, y en tiempo de ejecución se reserva el espacio anteriormente definido en memoria en el segmento de datos (variables globales) o en el segmento de pila o *stack* (variables locales).
3. Cada variable estática tiene un único nombre o identificador dentro de su ámbito.

Cuando se trabaja con variables estáticas estructuradas o estructuras de datos estáticas su empleo es muy sencillo pero implica una serie de inconvenientes:

1. Proporcionan una estructura rígida que apenas puede alterarse durante la ejecución del programa.
2. El espacio en memoria correspondiente tanto al segmento de datos como al segmento de pila es muy limitado.
3. No aprovechan de forma óptima la memoria disponible. Hay que reservar espacio en memoria para toda la estructura durante toda la ejecución independientemente de los requerimientos reales de los programas. En algunos casos se desperdiciará parte del espacio en memoria reservado y en otros no podrán almacenarse más datos de los previstos en un principio.

Se puede interpretar que una variable estática es una referencia fija de un dato en memoria. Es decir, una variable estática sirve para encontrar el mismo dato en memoria durante todo su ciclo de vida. Por tanto, cuando se usa una variable estática se realizan las siguientes operaciones a más bajo nivel:

1. Se reserva espacio en memoria para almacenar un dato correspondiente al tipo de la variable.
2. Durante todo el ciclo de vida la variable hace referencia al mismo espacio en memoria (al reservado al inicio del ámbito de la variable).
3. Cuando se resuelve el ámbito de la variable se libera la memoria asociada a la variable.

Por otro lado, las variables *dinámicas* no tienen identificador ni un ciclo de vida predeterminado como el descrito para las estáticas. Por tanto, cabe preguntarse cómo se hace referencia a una variable dinámica, es decir, cómo se encuentran en la memoria del ordenador. La respuesta son los punteros. Los punteros serán la referencia flexible de las variables dinámicas. Se destaca que la referencia dada por los punteros es flexible dada su condición de variables, es decir, un puntero puede cambiar la variable a la que hace referencia.

Así pues, a diferencia de las variables estáticas, las variables dinámicas:

1. No se declaran en ningún lugar del programa o de las rutinas.
2. Para almacenar una variable dinámica se reserva espacio en memoria en algún momento a determinar por el programador durante la ejecución del programa.
3. Asimismo puede también liberarse para otros usos ese espacio de memoria reservado para una variable dinámica durante la ejecución y en el momento que determine el programador. Es decir, el uso de la variable dinámica no implica reservar espacio en memoria durante toda la ejecución del programa.
4. Por consiguiente, el tamaño y, por lo tanto, el espacio reservado en memoria para las estructuras de datos dinámicas (formadas por un conjunto de variables dinámicas) puede variar durante la ejecución del programa según las necesidades del mismo.
5. Una misma variable dinámica puede tener varias referencias. Es decir, se puede encontrar el valor de una variable dinámica de formas distintas pero que son equivalentes.
6. La parte de la memoria utilizada para almacenar las variables dinámicas durante la ejecución de un programa es distinta del lugar de almacenamiento de las variables estáticas y se denomina *segmento de montículo*, memoria libre o *heap*. En general, esto tiene como consecuencia tener más espacio que el espacio máximo reservado para las variables estáticas, prácticamente hasta el espacio máximo libre disponible en memoria.

## 14.2. Punteros y variables dinámicas

Como se ha comentado anteriormente para poder trabajar con variables dinámicas es necesario almacenar su dirección de memoria en una variable puntero. Un apuntador, puntero o referencia representa una dirección de memoria. La declaración de una variable puntero sigue la sintaxis: `tipoDato *identificadorPuntero;` Por otro lado el operador *dirección de* (`&`) permite determinar la dirección de una variable mientras que el operador *indirección* (`*`) permite acceder a la variable a la que apunta una variable puntero. `NULL` representa una dirección de memoria nula. Es habitual el uso de los operadores de relación de igualdad (`==`) y desigualdad (`!=`) entre punteros. Sólo se permiten asignaciones directas entre punteros que apuntan al mismo tipo de dato pero se permiten otras asignaciones con conversión explícita (*cast*). En este sentido existe un tipo puntero indefinido (se denomina puntero a `void` o `void *`) que suele emplearse como comodín.

## 14.3. La función malloc

La función `malloc`, declarada como:

```
void* malloc(unsigned int)
```

está incluida en la librería `<stdlib.h>`. Al ejecutarse, reserva espacio en memoria para una nueva variable dinámica en el montículo que ocupa (en bytes) lo indicado en el argumento. Además devuelve un puntero a la dirección de memoria del primer byte del espacio reservado y `NULL` en caso de error. Sirve para cualquier tipo de variable dinámica por lo que es recomendable su empleo combinado con el operador *cast* correspondiente. Cuando este *cast* no se indica el compilador puede producir un aviso. Ejemplos de uso de la función:

```
int* p = (int*) malloc(sizeof(int));
char* s = (char*) malloc(5);
```

## 14.4. La función free

La liberación del espacio reservado previamente para una variable dinámica se lleva a cabo mediante la función **free**, declarada como:

```
void free(void *p);
```

que está incluida en la librería `<stdlib.h>`. La ejecución de la función libera el espacio en memoria reservado por una llamada previa a **malloc**. El puntero argumento debe tener el mismo valor que devuelve la función **malloc**. Una llamada a la función **free** libera toda la memoria reservada previamente por una llamada a **malloc**. Su uso evita las fugas de memoria (*memory leaks*). Por ejemplo:

```
int* p = (int*) malloc(sizeof(int));
/* Se utiliza p ... */
free(p);
```

## 14.5. Puntero a puntero e indirección múltiple

Una variable de tipo puntero puede apuntar a una variable dinámica de tipo puntero:

```
identTipo **p;
```

Por ejemplo:

```
int **p;
p = (int**) malloc(sizeof(int*));
*p = (int*) malloc(sizeof(int));
**p = 193;
```

## 14.6. Punteros y variables dinámicas simples

El uso de punteros resulta imprescindible en la gestión flexible de memoria que se realiza mediante variables dinámicas. A este efecto hay una serie de operaciones con punteros que afectan a las variables de una manera que es muy razonable, pero no intuitiva a primera vista. En esta sección se van a mostrar programas y diagramas que muestran el comportamiento de los punteros y las variables dinámicas en casos significativos.

### 14.6.1. Creación y liberación de una variable dinámica

La primera operación básica en la gestión de memoria dinámica es la creación de variables dinámicas. En C la creación de variables dinámicas se hace mediante la función **malloc** (de *memory allocation*, reserva de memoria, literalmente asignación de memoria). Es obligatorio utilizar la función **malloc** (u otras alternativas) para reservar memoria para las variables dinámicas pero, además, resulta igual de obligatorio el uso de un puntero, normalmente variable estática, como referencia de la nueva variable.

En el ejemplo 14.1 se puede observar un programa con reserva de una variable dinámica. La función **malloc** reserva el número de bytes indicado por el argumento y devuelve un puntero tipo **void\*** como resultado. En general, **malloc** se suele llamar utilizando como argumento una expresión que incluye el resultado del operador **sizeof** sobre un tipo de dato multiplicado por un número entero para reservar espacio para más de una variable. Normalmente el puntero resultante no resulta útil como **void\***, así que se fuerza su conversión a un puntero del tipo que se ha utilizado para calcular el tamaño del espacio reservado en memoria. En el ejemplo se ha forzado su conversión a **int\***.

Ejemplo 14.1: Reserva de memoria con malloc

---

```

1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main()
6 {
7     int *p = (int*)malloc(sizeof(int));
8 }
```

```

9     *p = 50;
10
11    printf(" puntero = %p variable dinamica = %d \n", p, *p);
12
13    free(p);
14
15    return 0;
16 }
```

---

Salida del programa:

```
puntero = 0x97ba008 variable dinamica = 50
```

Tal y como se ha indicado en las características de las variables dinámicas el espacio reservado en memoria por `malloc` (para una variable dinámica determinada) no tiene un identificador. La manera de asignar valores a la variable o usar su valor en expresiones es utilizar la dirección de memoria devuelta por `malloc` y almacenarla en una variable puntero. A través de ese puntero y aplicando el operador *indirección* se obtiene la variable dinámica.

El proceso de reserva de memoria tiene las siguientes fases:

1. Al arrancar el programa se reserva espacio para las variables estáticas declaradas en `main`.
2. Al llamar a `malloc`, en la línea 7, se reserva memoria para la nueva variable dinámica. En la figura 14.1 se muestran ambas variables todavía sin relación entre ellas, pero ambas ya en memoria.

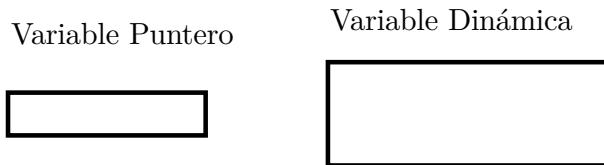


Figura 14.1: Un puntero y una variable dinámica en memoria

3. El puntero devuelto por `malloc` se guarda para usarse como referencia en la variable puntero estática. El resultado final se muestra en la figura 14.2 donde la flecha es una forma de representar gráficamente el apuntamiento, es decir, el puntero *apunta* a la variable, la dirección de la variable dinámica se guarda como valor de la variable puntero.

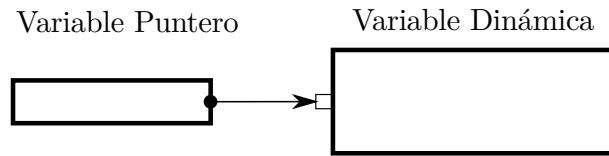


Figura 14.2: Puntero apuntando a una variable dinámica

Al final de este proceso el valor del puntero queda inicializado a una dirección de memoria válida, en el sentido de ser el lugar donde se encuentra una variable dinámica. Si `malloc` no tuviese éxito al reservar memoria devuelve el valor de `NULL`. De esta manera indica que la nueva variable no se ha podido crear.

Nótese también que las variables dinámicas no están sujetas al ciclo de vida de las variables estáticas cuya reserva y liberación se rigen por el ámbito de su identificador. Las variables dinámicas se pueden usar siempre que se disponga de una variable puntero con la dirección de memoria que ocupan y hasta que se liberen.

Así pues, para completar el ciclo de vida de una variable dinámica es necesario liberar el espacio reservado. Esta operación se realiza con una llamada a la función `free`. La función `free` tiene como parámetro una variable puntero: una dirección de memoria. Realmente `free` no actúa sobre el puntero sino sobre la variable a la que apunta, liberando una reserva realizada previamente.

Las funciones `malloc` y `free` están declaradas en `<stdlib.h>` como:

```
void* malloc(size_t size);
void free(void *ptr);
```

Es importante observar que el resultado de `malloc` y el parámetro de `free` son punteros “genéricos” (apuntan a `void`). Estas funciones se pueden utilizar para cualquier tipo de variable dinámica. El tipo `size_t` es el tipo de resultado del operador `sizeof`, de tipo entero sin signo, pero cuya definición depende del compilador y se realiza en `<stddef.h>` y otros archivos de cabecera (normalmente no es necesario incluirlo directamente).

### 14.6.2. Manipulación de la variable dinámica

Tal y como se ha visto de forma general el operador indirección sirve para devolver la variable a la que apunta un puntero. Para variables dinámicas es imprescindible el uso de este operador puesto que es el único mecanismo para acceder al valor de la variable dinámica. Naturalmente el resultado del operador indirección, tomado como una variable, se puede utilizar en el lado izquierdo de una asignación o en otros lugares donde se espere una variable y también como parte de una expresión, por ejemplo: en una suma o un producto, donde se usa su valor para calcular un resultado.

En el ejemplo 14.1, en la línea 9, se muestra un ejemplo del operador indirección, en este caso, usado como una variable para una asignación cuya consecuencia es guardar en la variable dinámica el valor 50. El estado de la memoria después de esta sentencia se muestra en la figura 14.3.

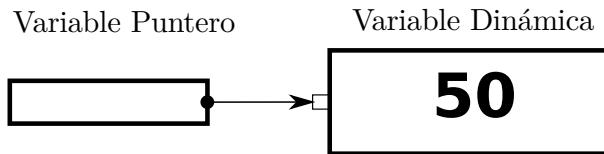


Figura 14.3: Puntero apuntando a una variable dinámica

### 14.6.3. Múltiples punteros y variables dinámicas

Normalmente existe una correspondencia uno a uno entre punteros y variables dinámicas. Cada puntero es capaz de guardar una única referencia a una variable dinámica y una variable dinámica necesita al menos un puntero para resultar útil como dato.

Esta limitación y forma de funcionamiento se muestra en el ejemplo 14.2, donde se puede observar cómo se crean dos variables dinámicas, una por cada llamada a `malloc` y cómo son necesarias las variables puntero `p` y `q` para guardar referencias a estas variables.

Ejemplo 14.2: Múltiples punteros y variables dinámicas

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     int *p; int *q;
8
9     p = (int*)malloc(sizeof(int));
10    q = (int*)malloc(sizeof(int));
11
12    *p = 30;
13    *q = 40;
14
15    printf(" puntero = %p variable dinamica = %d \n", p, *p);
16    printf(" puntero = %p variable dinamica = %d \n", q, *q);
17
18    *q = *p;
19
20    printf(" puntero = %p variable dinamica = %d \n", p, *p);
21    printf(" puntero = %p variable dinamica = %d \n", q, *q);
22
23    free(p); free(q);
24
25    return 0;
26 }
  
```

---

Salida del programa:

```
puntero = 0x99cb008 variable dinamica = 30
puntero = 0x99cb018 variable dinamica = 40
puntero = 0x99cb008 variable dinamica = 30
puntero = 0x99cb018 variable dinamica = 30
```

Al final del proceso de reserva, después de la línea 9, la situación del uso de memoria se muestra en la figura 14.4, donde se pueden ver las dos variables puntero apuntando cada una a una variable dinámica.

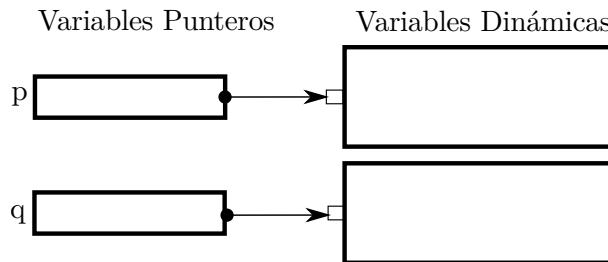


Figura 14.4: Dos punteros apuntando a dos variables dinámicas

Por otro lado, en el ejemplo 14.2, también se muestran operaciones de asignación con las variables dinámicas. Lo más importante es observar que siempre se han acompañado los identificadores de las variables puntero con el operador de indirección y que el efecto siempre se aplica a la variable dinámica. El valor de la variable puntero, que se representa por la flecha de apuntamiento, no cambia. El resultado en memoria después de las primeras asignaciones de las variables dinámicas, después de la línea 12, se muestra en la figura 14.5.

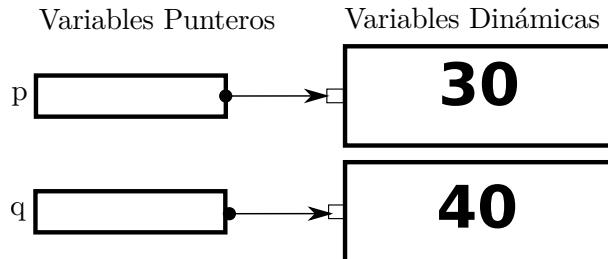


Figura 14.5: Dos punteros apuntando a dos variables dinámicas con valores

En la línea 17 del mismo ejemplo, se asigna una variable dinámica a otra. Igual que antes esto no supone ningún cambio en las variables puntero, lo que se puede comprobar a través de la salida por pantalla. Naturalmente el valor de la variable dinámica sí que ha cambiado. El resultado en memoria se muestra en la figura 14.6.

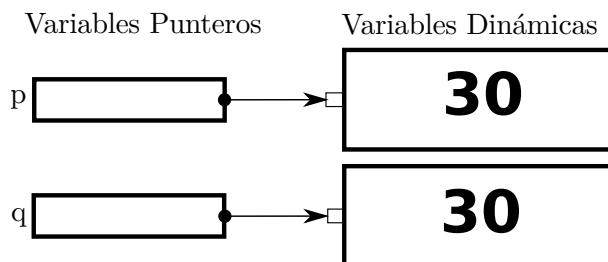


Figura 14.6: Dos variables punteros apuntando a dos variables dinámicas con el mismo valor

#### 14.6.4. Asignación entre punteros

La operación más crítica en gestión de variables dinámicas es la asignación entre variables puntero. Como consecuencia de una asignación cambia la flecha que se ha representado en las figuras anteriores. Esta flecha representa el valor

de la variable puntero, su apuntamiento. Esta operación puede tener como consecuencia errores o efectos inesperados si no se realiza con precaución y con una idea muy precisa de las consecuencias de la operación.

Por otro lado, la asignación entre punteros permite que dos punteros apunten a la misma variable dinámica. Ésta es una posibilidad interesante y de utilidad en muchos problemas de programación, pero de nuevo debe ser empleada con precaución. En concreto, debe conocerse cuando se produce esta circunstancia para evitar efectos imprevistos en la ejecución del código que se desarrolle.

En el ejemplo 14.3 se muestran distintas asignaciones entre punteros.

Ejemplo 14.3: Asignaciones entre punteros

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7
8     int *p; int *q;
9
10    /* Una variable dos punteros */
11    p = (int*)malloc(sizeof(int));
12    q = p;
13
14    *p = 30;
15    *q = 40;
16
17    printf(" puntero = %p variable dinamica = %d \n", p, *p);
18    printf(" puntero = %p variable dinamica = %d \n", q, *q);
19
20    /* Dos variables dos punteros */
21    q = (int*)malloc(sizeof(int));
22    *q = 95;
23
24    printf(" puntero = %p variable dinamica = %d \n", p, *p);
25    printf(" puntero = %p variable dinamica = %d \n", q, *q);
26
27    /* Una variables dos punteros */
28    q = p; /* Provoca variable inaccesible y fuga de memoria */
29    *q = 40;
30
31    printf(" puntero = %p variable dinamica = %d \n", p, *p);
32    printf(" puntero = %p variable dinamica = %d \n", q, *q);
33
34    free(p);
35
36    return 0;
37 }
```

---

Salida del programa:

```

puntero = 0x9a5a008 variable dinamica = 40
puntero = 0x9a5a008 variable dinamica = 40
puntero = 0x9a5a008 variable dinamica = 40
puntero = 0x9a5a018 variable dinamica = 95
puntero = 0x9a5a008 variable dinamica = 40
puntero = 0x9a5a008 variable dinamica = 40
```

En primer lugar se puede observar un caso con dos punteros y una única variable dinámica. Ambos punteros apuntan a la misma variable como consecuencia de la asignación. Como consecuencia `*p` y `*q` representan la misma variable. En la figura 14.7 se muestra un diagrama con la situación en memoria después de ejecutar la asignación.

A continuación se repite la situación descrita en el apartado anterior. Al realizar una nueva llamada a `malloc`, cada puntero apunta a una variable dinámica diferente. La situación en memoria es igual que la que se muestra en la

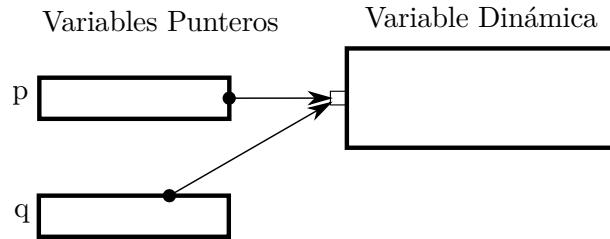


Figura 14.7: Dos punteros apuntando a una variable dinámica

figura 14.4. Respecto de la situación anterior se puede comprobar que la asignación del valor que devuelve `malloc` a `q` supone un cambio en el apuntamiento de la variable puntero.

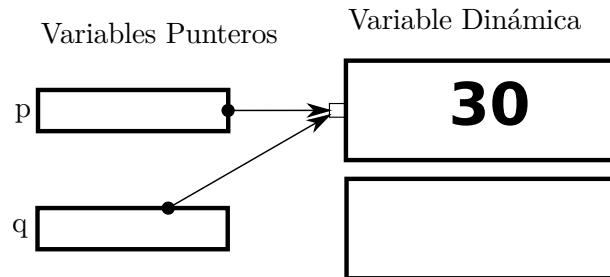


Figura 14.8: Variable dinámica inaccesible y fuga de memoria

A partir de esa situación se realiza una nueva asignación a la variable puntero `q`. Esta asignación vuelve a cambiar su valor y por tanto la variable a la que apunta. La variable puntero vuelve a apuntar a la primera variable que se creó y pierde la variable cuya referencia estaba guardando. Este es un ejemplo de mal uso de una asignación entre punteros. Tiene dos consecuencias negativas relacionadas entre sí:

1. Se pierde la referencia a la variable dinámica creada en segundo lugar, de manera que queda inaccesible. No se puede usar o modificar su valor de ninguna manera.
2. En consecuencia no se puede pedir la liberación de la memoria reservada para dicha variable. Esto se conoce como fuga de memoria (*memory leak*) y su existencia se considera un defecto grave en la calidad del código desarrollado incluso si el programa funciona correctamente. Uno de los síntomas de los programas con fugas de memoria es su incapacidad de ejecutarse de manera sostenida. Dependiendo de la gravedad de la pérdida de memoria pueden funcionar durante una hora, quizás más, pero en un plazo medio, quizás unos días, acaban fallando.

La situación final en memoria del ejemplo 14.3 se muestra en el diagrama de la figura 14.8. A consecuencia de la asignación entre punteros, no existe ningún puntero que guarde una referencia de la variable dinámica creada en segundo lugar.

## 14.7. Variables dinámicas de tipo vector

Para crear y usar variables dinámicas de tipo vector se aprovechan algunas características de los *arrays* en C y su uso a través de punteros. Estas son:

1. Un *array* en C es una estructura de datos en memoria que coloca variables de mismo tipo de forma consecutiva, una detrás de otra.
2. La reserva de memoria asociada a una variable memoria se realiza en un bloque compacto en memoria. Si dicho bloque se divide en bloques más pequeños, éstos resultan consecutivos.
3. Un puntero que apunta al primer elemento de un *array* y apunta al tipo de los elementos de *array* puede servir para obtener mediante el operador *indexación* `[]` cualquier otro elemento del *array* facilitando el índice que se precise en cada caso.

Estas características hacen posible que se pueda crear un *array* dinámico con una *única* variable dinámica y, a su vez, se pueda manipular de forma idéntica a un *array* estático si se convierte al tipo de los elementos del *array* la dirección de memoria suministrada por `malloc` a una variable puntero. El puntero así obtenido apunta al primer elemento del *array* y tiene el tipo necesario para poder trabajar correctamente con él.

En el ejemplo 14.4 se muestran la creación y uso de un vector dinámico.

---

Ejemplo 14.4: Creación y uso de un vector dinámico

---

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main()
6  {
7      unsigned tam = 10, i;
8      int *vector = (int*) malloc(tam*sizeof(int));
9
10     /* rellena el vector dinamico */
11     for ( i = 0; i < tam; ++i ) {
12         vector[i] = 3*i;
13     }
14
15     /* muestra el vector dinamico */
16     for ( i = 0; i < tam; ++i ) {
17         printf("%i ", vector[i]);
18     }
19
20     free(vector);
21     return 0;
22 }
```

---

Salida del programa:

0 3 6 9 12 15 18 21 24 27

Obsérvese que el vector dinámico se crea mediante una *única* llamada a `malloc` y se libera mediante una *única* llamada a `free`. Obsérvese también que en ningún sitio es necesario el uso del operador *indirección*, el puntero `vector` se comporta igual que si fuese un *array* estático. De hecho, las componentes del vector dinámico se determinan sencillamente mediante `vector[i]`.

## 14.8. Matrices dinámicas

El caso de matrices dinámicas, con número de filas y columnas variables, es más complejo que el de los vectores dinámicos como consecuencia de los siguientes puntos:

1. El *array* multidimensional (matriz) en C es una estructura de datos que coloca en memoria las filas una a continuación de otra. A su vez cada fila se compone de variables consecutivas del tipo de los elementos de la matriz.
2. El operador indexación de una matriz estática se basa en que se conoce el tamaño de una fila (el número de columnas), por tanto el primero de los operadores de indexación que se aplica a una variable de tipo matriz localiza la fila de la matriz mediante el desplazamiento correspondiente, mientras que el segundo operador localiza la columna dentro de la fila.
3. No se puede declarar una variable puntero cuyo operador de indexación implique un desplazamiento variable equivalente al tamaño de la fila.<sup>1</sup>

En consecuencia no es posible crear una *única* variable dinámica cuyo puntero de referencia se comporte como una matriz estática, es decir, se pueda hacer `matriz[i][j]`. Existen varias soluciones alternativas para resolver o abordar este problema, a continuación se va a explicar una de ellas y se puede encontrar en el capítulo de material adicional otras dos soluciones a este problema.

---

<sup>1</sup>Sí existe una forma no trivial de realizar esta declaración, pero no bajo el estándar ANSI (C89), sino en C99 y que, además, no compila en los compiladores de Microsoft.

### 14.8.1. Matriz dinámica como vector de punteros

Esta solución se basa en crear un número de variables dinámicas igual al número de filas de la matriz y guardar la referencia a cada una de esas variables en un vector (dinámico) de punteros.

Supuesta una matriz dinámica, `matriz`, construida de esta forma, la subexpresión, `matriz[i]` devuelve el puntero que apunta a la fila `i`. Efectivamente, al ser `matriz` un vector, la aplicación del operador indexación devuelve el elemento correspondiente, dado que el vector es de punteros, `matriz[i]` es un puntero. Ahora bien, ya sabemos que punteros y vectores son intercambiables de manera que al puntero obtenido le podemos aplicar otro operador indexación: `matriz[i][j]` y de esta manera obtenemos el elemento de la matriz.

En el ejemplo 14.5 obsérvese que:

1. La declaración de `matriz` es de tipo puntero a puntero: un puntero es por los elementos del vector y el otro es para recoger la referencia del vector dinámico.
2. Las dos llamadas a `malloc`, una para el vector de punteros y otro para las filas.
3. La aplicación de los operadores de indexación a la `matriz`.

Ejemplo 14.5: Matriz dinámica

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NF 3      /* Numero de filas */
6 #define NC 3      /* Numero de columnas */
7
8 int main()
9 {
10     int **m;
11     int i, j;
12     float suma = 0.0;
13     m = (int **) malloc (NF*sizeof(int *));
14     for (i = 0; i<NF; i++) {
15         m[i] = (int *) malloc (NC*sizeof(int));
16     }
17     for (i = 0; i<NF; i++) {
18         for (j = 0; j<NC; j++) {
19             m[i][j] = i+j;
20         }
21     }
22     for (i = 0; i<NF; i++) {
23         for (j = 0; j<NC; j++) {
24             printf(" %d", m[i][j]);
25             suma += m[i][j];
26         }
27         printf("\n");
28     }
29     printf("Media: %f\n", suma/(NF*NC));
30     for (i = 0; i<NF; i++) {
31         free(m[i]);
32     }
33     free(m);
34     return 0;
35 }
```

---

Salida por pantalla en la ejecución:

```

0 1 2
1 2 3
2 3 4
Media: 2.000000
```

## 14.9. Variables dinámicas de tipo struct

### 14.9.1. Uso del tipo estructura para variables dinámicas

Una ventaja adicional a la agrupación de datos en un único tipo es la simplificación de la gestión de memoria cuando se usan variables dinámicas. La reserva y liberación de memoria se puede hacer de una sola vez, de manera que el código resulta más compacto y sencillo.

En el ejemplo 14.6, se ha desarrollado exactamente el mismo programa que en el ejemplo 12.1, pero utilizando una variable dinámica del tipo **Estructura** y su puntero en vez de una variable estática.

Ejemplo 14.6: Variable dinámica de tipo estructura

---

```

1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 struct Estructura
6 {
7     int codigo;
8     char nombre[30];
9     double valor;
10 };
11
12 int main()
13 {
14     struct Estructura *p_st;
15     p_st = (struct Estructura*) malloc(sizeof(struct Estructura));
16
17     p_st->codigo = 7;
18     strcpy(p_st->nombre, "temperatura");
19     p_st->valor = 23.4;
20
21     printf("codigo: %d, nombre: %s, valor: %f \n",
22            p_st->codigo, p_st->nombre, p_st->valor);
23
24     p_st->valor += 20.0;
25
26     printf("codigo: %d, nombre: %s, valor: %f \n",
27            p_st->codigo, p_st->nombre, p_st->valor);
28
29     free(p_st);
30
31     return 0;
32 }
```

---

Salida del programa:

```

codigo: 7, nombre: temperatura, valor: 23.400000
codigo: 7, nombre: temperatura, valor: 43.400000
```

Es conveniente resaltar en este ejemplo algunos aspectos característicos del uso de los tipos **struct** con variables dinámicas:

1. Naturalmente la reserva de memoria se realiza de forma análoga a cualquier otra variable, incluida la facilidad de usar el operador **sizeof** que calcula el tamaño en memoria de la variable<sup>2</sup>.
2. La notación habitual del operador indirección es mucho más simple y razonable. En vez de utilizar, **(\*puntero).atributo**, tal y como se ha hecho en el ejemplo 12.3, en la implementación de la función **sumar** se puede utilizar directamente **puntero->atributo** siendo esta notación mucho más clara.
3. Nótese que la reserva de memoria de vectores estáticos con longitud conocida se realiza sin más. En este ejemplo, el espacio de 30 caracteres de la cadena se reserva directamente porque forma parte del tamaño de la estructura. Esto no es así si se desea usar vectores dinámicos. En ese caso se utiliza un puntero entre los atributos de la estructura y luego hay que realizar la reserva de memoria para variables dinámicas asociadas a ese puntero.

<sup>2</sup>el tamaño de la estructura en memoria no suele ser la suma del tamaño de sus campos, por eso, es obligatorio usar **sizeof**.

### 14.9.2. Estructuras de datos dinámicas

Existe una gran variedad de estructuras dinámicas que se pueden implementar en C, las más importantes son las siguientes:

**Listas:** Una lista es una estructura de datos lineal y ordenada, es decir, una secuencia, a cada dato le sigue otro del mismo tipo desde el primero hasta el último. En C se pueden implementar usando un **struct** que incorpora un puntero a la misma estructura y que sirve para apuntar al siguiente elemento. Existe algunos tipos especiales de listas, por ejemplo: listas dobles (con otro puntero que apunta al elemento anterior en la lista) o listas circulares (el puntero del último elemento apunta al primer elemento de la lista).

**Árboles:** Un árbol es una estructura jerárquica de datos donde cada elemento o nodo puede tener varios nodos descendientes y un ascendiente. Existe un nodo principal o raíz que no tiene ascendientes. Los nodos sin descendientes se suelen denominar hojas. Existen tipos especiales de árboles como los arboles binarios (máximo de 2 descendientes por nodo) o arboles AVL (además cumplen ciertas propiedades). Los árboles son adecuados cuando se necesita realizar búsquedas en la estructura de datos dinámica. En C se implementan con un **struct** donde se definen tantos punteros al mismo **struct** como descendientes se permitan.

**Tablas de Hash:** un tabla de *hash* es una estructura dinámica similar a un vector, pero mucho más compleja. Sistemáticamente se basa en usar el resultado de una función *hash* como índice para los datos guardados en la tabla. Es muy recomendable su uso en aplicaciones con muchas búsquedas o con búsquedas en estructuras muy grandes porque el tiempo de búsqueda es constante independientemente del número de elementos dentro de la tabla (aproximadamente es el tiempo de aplicar la función *hash*).

Como ejemplo ilustrativo del manejo de estructuras dinámicas se va a desarrollar un ejemplo de estructura dinámica en C, una cola. Es una estructura dinámica presente en muchos problemas de programación, un ejemplo muy significativo es la cola de mensajes de Windows (*Windows Message Queue*), pero existen otras muchas. Sus características principales son:

**Colas:** una cola es una estructura dinámica de datos que se caracteriza porque tiene tres operaciones básicas:

1. insertar un nuevo elemento al final de la estructura,
2. eliminar un elemento del principio de la estructura y
3. conocer si la estructura tiene algún elemento.

Un nombre alternativo a cola es FIFO, del inglés *First In, First Out* que describe el orden de entrada y salida a la estructura. La implementación utiliza como elemento base un **struct** con un puntero al siguiente elemento de la cola. Por otro lado, es conveniente guardar el principio y final de la cola para simplificar el código fuente de la implementación de las operaciones necesarias. Su comportamiento es análogo a una fila de turno de atención en un comercio.

A continuación desarrollaremos un programa de ejemplo de una cola con alguna funcionalidad adicional que no forma parte de la definición de esta estructura. La definición de una pila se puede encontrar en la parte de material adicional.

### 14.9.3. Implementación de una cola

El ejemplo 14.7 muestra el código fuente de un programa en C que implementa y usa una cola. Los detalles a tener en cuenta respecto de esta implementación son:

1. Observe la declaración de datos. Es importante que la declaración de la estructura **Elemento** incluye un puntero a si misma. Este puntero es la base para conectar los datos de la estructura dinámica, cada vez que se crea un elemento tenemos un nuevo puntero disponible para almacenar otro elemento de la estructura dinámica. En estructuras más complejas no se usa un único puntero sino varios (por ejemplo, para árboles binarios o para listas doblemente enlazadas se usan dos punteros).
2. Aunque esta cola está formada por datos de tipo **double** se puede escribir una cola que almacene cualquier dato simplemente cambiando la declaración de **dato**. Es más, incluso es posible hacer una cola de cualquier tipo de variable haciendo que **dato** sea un puntero genérico (**void\***).
3. Para simplificar las declaraciones de las funciones que realizan operaciones sobre la cola, ésta se ha definido como otra estructura con dos punteros que apuntan al primer y último de sus elementos.

4. Observe que el parámetro puntero a cola se declara con **const** cuando la función que opera sobre la cola no la modifica.
5. Para simplificar el acceso a los atributos de las estructura siempre se ha empleado el operador **->**.
6. Para estudiar el ejemplo se recomienda dibujar un diagrama donde se muestren las operaciones que se realizan en cada una de las sentencias numeradas. La solución de implementación de estos pasos no es única, pero hay algunos aspectos que deben respetar cualquier solución a este problema, entre ellos:
  - a) En general, cuando sea necesaria una nueva variable dinámica tendremos que llamar a **malloc** y cuando necesitemos liberarla llamaremos a **free**.
  - b) Un puntero no se puede reasignar o liberar si se pierde la referencia de una variable dinámica. Por ejemplo, esto tiene como consecuencia que para eliminar el primer elemento de la pila sean necesarios los tres primeros pasos programados en **pop**. Si se comete este error se puede provocar:
    - que la implementación directamente **no funcione** o
    - que se produzca una **fuga de memoria** (*memory leak*). Este error es grave, especialmente en programas que funcionen de forma permanente (acaban consumiendo toda la memoria del ordenador).
  - c) Nunca se debe acceder a un puntero (llamar al operador **->**) si no se comprueba previamente que existe la variable dinámica a la que apunta.
  - d) Cuando se inicializa una estructura con un atributo puntero, éste se debe inicializar a **NULL** para marcar que no existe su variable dinámica. Cuando un puntero apunta a **NULL** es correcto asignarle una nueva variable dinámica.
7. Finalmente, para comprobar que la implementación de estas funciones es correcta se debe suponer que la cola está bien constituida y seguir los pasos de cada función. Si al final de cada una, la cola queda igualmente bien constituida, la implementación es correcta. Se deben realizar estas comprobaciones para colas vacías, con un único elemento y con un número arbitrario de elementos. En una cola bien constituida:
  - a) El puntero **sig** de cada elemento apunta al siguiente y el puntero **sig** del último elemento apunta a **NULL**.
  - b) Los punteros **prim** y **ult** de la estructura de la cola siempre apuntan, respectivamente, al primero y último de sus elementos.
  - c) Si la estructura está vacía, al menos el puntero **prim** apunta a **NULL**.

En el ejemplo podemos ver las tres funciones que se pueden aplicar a una cola. Estas funciones se han llamado: **is\_empty**, **pop** y **push** porque reciben nombres similares a estos en algunas librerías de estructuras de datos dinámicas. Además se han añadido las funciones: **create\_queue** y **destroy\_queue**, estas funciones sirven para forzar a que la cola se inicialice y destruya correctamente. En este sentido se debe observar que la cola solo se utiliza a través de un puntero y de las funciones definidas, de una manera muy similar a cómo se usan los archivos a través de **FILE\***. Destacamos también que la función **pop** comprueba si la cola está vacía y devuelve **NaN** (*Not a Number*) en ese caso.

Ejemplo 14.7: Implementación de una cola de números en coma flotante

---

```

1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 struct Elemento
6 {
7     double dato;
8     struct Elemento *sig;
9 };
10
11 struct FIFO
12 {
13     struct Elemento *prim;
14     struct Elemento *ult;
15 };
16
17 struct FIFO* create_queue()
18 {

```

```

19     struct FIFO* f = malloc(sizeof(struct FIFO));
20     f->prim = NULL; /* 1 */
21     return f;
22 }
23
24 int is_empty(const struct FIFO *f)
25 {
26     return f->prim == NULL;
27 }
28
29 double pop(struct FIFO *f)
30 {
31     struct Elemento *aux;
32     double result;
33
34     if ( is_empty( f ) )
35     {
36         return 0.0/0.0; /*easy way to produce NaN*/
37     }
38     else
39     {
40         aux = f->prim; /* 1 */
41         result = aux->dato; /* 2 */
42         f->prim = aux->sig; /* 3 */
43         free ( aux ); /* 4 */
44         return result; /* 5 */
45     }
46 }
47
48 void destroy_queue(struct FIFO* f)
49 {
50     while ( ! is_empty( f ) )
51     {
52         pop( f );
53     }
54     free ( f );
55 }
56
57 void push(struct FIFO *f, double x)
58 {
59     struct Elemento *aux;
60
61     aux = malloc(sizeof(struct Elemento)); /*1*/
62     aux->dato = x; /*2*/
63     aux->sig = NULL; /*3*/
64
65     if ( f->prim != NULL )
66     {
67         f->ult->sig = aux; /*4*/
68         f->ult = aux; /*5*/
69     }
70     else
71     {
72         f->prim = aux; /*6*/
73         f->ult = aux; /*7*/
74     }
75 }
76
77 int main()
78 {
79     int i;
80     struct FIFO *fila;
81

```

```

82     fila = create_queue();
83
84     push(fila, 4.2);
85     push(fila, 3.7);
86     push(fila, 8.2);
87
88     printf("El primero es %f\n", pop(fila));
89
90     push(fila, 7.0);
91
92     for ( i = 0; i < 3; ++i )
93     {
94         printf("%f\n", pop(fila));
95     }
96
97     printf("vacia: %i\n", is_empty(fila));
98
99     printf("queda alguno?: %f\n", pop(fila));
100
101    destroy_queue( fila );
102
103    return 0;
104 }
```

---

Salida del programa:

```

El primero es 4.200000
3.700000
8.200000
7.000000
vacia: 1
queda alguno?: -nan
```

#### 14.9.4. Ampliación de la implementación de una cola

Para complementar el ejemplo anterior se va a introducir una operación que no forma parte de las operaciones teóricas que se pueden realizar sobre una cola, pero que puede ser interesante para:

- depurar una cola,
- realizar operaciones no destructivas sobre los elementos de la cola y
- como ejemplo académico del uso de los punteros en estructuras dinámicas.

Se trata de hacer un recorrido sobre los elementos de una cola y hacer una operación sobre cada uno de ellos. Para realizar esta tarea se debe utilizar un puntero auxiliar que debe tomar valores desde la dirección del primer elemento de la cola hasta el último y finalmente `NULL` cuando se alcanza el final de la cola (y por tanto el final de la operación requerida). En el ejemplo 14.8 se muestra una versión modificada del ejemplo anterior donde el recorrido de la cola se ha introducido en la implementación de `main` para calcular la media de los elementos de cola. Se pueden hacer muchas operaciones análogas a este ejemplo realizando otras manipulaciones de cada uno de los elementos. En cualquiera de ellas es crucial:

- Asignar al puntero auxiliar la dirección del primer elemento de la cola.
- Avanzar a la posición siguiente reasignando el valor del puntero auxiliar.
- Comprobar que se alcanza el final de la cola (el valor de `aux` es `NULL`).
- Nótese también que reasignar el valor del puntero auxiliar no es problema porque todas las variables dinámicas de la cola tienen otro puntero que las apunta, de hecho el puntero `sig` del elemento anterior.

## Ejemplo 14.8: Implementación de una cola con recorrido

---

```

1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 struct Elemento
6 {
7     double dato;
8     struct Elemento *sig;
9 };
10
11 struct FIFO
12 {
13     struct Elemento *prim;
14     struct Elemento *ult;
15 };
16
17 struct FIFO* create_queue()
18 {
19     struct FIFO* f = malloc(sizeof(struct FIFO));
20     f->prim = NULL; /* A */
21     return f;
22 }
23
24 int is_empty(const struct FIFO *f)
25 {
26     return f->prim == NULL;
27 }
28
29 double pop(struct FIFO *f)
30 {
31     struct Elemento *aux;
32     double result;
33
34     if ( is_empty( f ) )
35     {
36         return 0.0/0.0; /*easy way to produce NaN*/
37     }
38     else
39     {
40         aux = f->prim; /* 1 */
41         result = aux->dato; /* 2 */
42         f->prim = aux->sig; /* 3 */
43         free ( aux ); /* 4 */
44         return result; /* 5 */
45     }
46 }
47
48 void destroy_queue(struct FIFO* f)
49 {
50     while ( ! is_empty( f ) )
51     {
52         pop( f );
53     }
54     free ( f );
55 }
56
57 void push(struct FIFO *f, double x)
58 {
59     struct Elemento *aux;
60
61     aux = malloc(sizeof(struct Elemento)); /*1*/
62     aux->dato = x; /*2*/

```

```

63     aux->sig = NULL; /*3*/
64
65     if ( f->prim != NULL )
66     {
67         f->ult->sig = aux; /*4*/
68         f->ult = aux; /*5*/
69     }
70     else
71     {
72         f->prim = aux; /*6*/
73         f->ult = aux; /*7*/
74     }
75 }
76
77 int main()
78 {
79     struct FIFO *fila;
80     struct Elemento *aux; double suma = 0; int num = 0;
81
82     fila = create_queue();
83
84     push(fila, 4.2);
85     push(fila, 3.7);
86     push(fila, 8.2);
87
88     /* Calcula la media mediante recorrido
89      * suponemos que la cola tiene al menos 1 elemento */
90     aux = fila->prim; /* Desde el primero */
91     while ( aux != NULL ) /* Hasta el final */
92     {
93         /* Procesa elemento */
94         suma += aux->dato;
95         ++num;
96         aux = aux->sig; /* Avanza al siguiente */
97     }
98     printf("Media elementos de la cola: %.10f\n", suma / num );
99
100    destroy_queue( fila );
101
102    return 0;
103 }
```

---

Salida del programa:

Media elementos de la cola: 5.3666666667

## Ejercicios resueltos del capítulo de Variables dinámicas

- Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 14.9: Ejemplo de uso de una variable dinámica

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int a, b;
7     int* p = (int*)malloc(sizeof(int));
8     a = 15;
9     b = 39;
10    *p = 57;
11    printf("Direccion de a: %p. a vale %d.\n", &a, a);
```

```

12     printf("Direccion de b: %p. b vale %d.\n", &b, b);
13     printf("Direccion de p: %p\n", &p);
14     printf("Direccion almacenada en p: %p\n", p);
15     printf("Valor de la var. dinamica ");
16     printf("apuntada por p (*p vale): %d.\n", *p);
17     return 0;
18 }
```

---

2. Construir un programa que muestre el tamaño en memoria de punteros a distintos tipos de datos y la dirección de memoria almacenada
3. Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 14.10: Ejemplo de uso de free

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int* p = (int*)malloc(sizeof(int));
7     *p = 12357;
8     printf("Direccion de p: %p\n", &p);
9     printf("Direccion almacenada en p: %p\n", p);
10    printf("Valor de la var. apuntada por p: %d.\n", *p);
11    free(p);
12    printf("Direccion de p: %p\n", p);
13    printf("Direccion almacenada en p: %p\n", p);
14    printf("Valor de la var. apuntada por p: %d.\n", *p);
15    return 0;
16 }
```

---

4. Indicar la salida por pantalla al ejecutarse el siguiente programa

Ejemplo 14.11: Ejemplo de doble indirección

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p;
7     int **q;
8     q = (int **) malloc(sizeof(*q));
9     *q = (int *) malloc(sizeof(int));
10    **q = 93;
11    printf("*q vale %d\n", **q);
12    p = (int *) malloc(sizeof(int));
13    *p = 126;
14    free(*q);
15    *q = p;
16    printf("**q vale %d\n", **q);
17    return 0;
18 }
```

---

## Soluciones a los ejercicios del capítulo de Variables dinámicas

1. La salida por pantalla es:

Direccion de a: 0xbfc236a4. a vale 15.  
 Direccion de b: 0xbfc236a8. b vale 39.

```
Direccion de p: 0xbfc236ac
Direccion almacenada en p: 0x81f4008
Valor de la var. dinamica apuntada por p (*p vale): 57.
```

2. Programa con variables dinámicas de distintos tipos

Ejemplo 14.12: Ejemplo de variables dinámicas de diferentes tipos

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char *p1;
7     int *p2;
8     double *p3;
9     p1 = (char *) malloc(sizeof(char));
10    *p1 = 'A';
11    printf("p1 apunta a %p\n",p1);
12    printf("*p1 ocupa %d byte",sizeof(*p1));
13    printf(" y vale %c\n",*p1);
14    p2 = (int *) malloc(sizeof(int));
15    *p2 = 12;
16    printf("p2 apunta a %p\n",p2);
17    printf("*p2 ocupa %d bytes",sizeof(*p2));
18    printf(" y vale %d\n",*p2);
19    p3 = (double *) malloc(sizeof(double));
20    *p3 = 70.95;
21    printf("p3 apunta a %p\n",p3);
22    printf("*p3 ocupa %d bytes",sizeof(*p3));
23    printf(" y vale %f\n",*p3);
24    return 0;
25 }
```

---

Salida por pantalla en la ejecución:

```
p1 apunta a 0x8818008
*p1 ocupa 1 byte y vale A
p2 apunta a 0x8818018
*p2 ocupa 4 bytes y vale 12
p3 apunta a 0x8818028
*p3 ocupa 8 bytes y vale 70.950000
```

3. La salida por pantalla es:

```
Direccion de p: 0xbfc5169c
Direccion almacenada en p: 0x8a4e008
Valor de la var. apuntada por p: 12357.
Direccion de p: 0x8a4e008
Direccion almacenada en p: 0x8a4e008
Valor de la var. apuntada por p: 0.
```

4. La salida por pantalla es:

```
*q vale 93
**q vale 126
```

## Otros ejercicios propuestos del capítulo de Variables dinámicas

1. Completar el siguiente código

Ejemplo 14.13: Calcular la media de un vector dinámico

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     unsigned tam = 10;
8     int *vector = (int*) malloc(tam*sizeof(int));
9
10    /* Rellena aquí con tu código */
11
12    free(vector);
13    return 0;
14 }
```

---

para (a) asignar valores a los elementos del *array*, (b) visualizar los valores por pantalla, (c) calcular su media aritmética y (d) mostrarla por pantalla.

2. Completar el siguiente código

Ejemplo 14.14: Asignar valores a un vector dinámico de caracteres

---

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     unsigned tam = 30;
8     char *s = (char*) malloc(tam*sizeof(char));
9
10    /* Rellena aquí con tu código */
11
12    free(s);
13    return 0;
14 }
```

---

para (a) asignar valores aleatorios (caracteres con letras de la ‘A’ a la ‘Z’) a los elementos del *array* y (b) visualizar los valores por pantalla.

3. Dada una estructura de datos dinámica, MatrizNxN, que sirve para representar una matriz por filas, escribir una función que modifique la matriz intercambiando las dos filas de índices i y j.
4. Codificar el tipo de dato estructura necesario para construir una lista dinámica con elementos que tengan dos miembros: (a) un campo entero donde almacenar el número de matrícula de un alumno y (b) un campo **sig** donde almacenar la dirección de memoria del siguiente elemento de la lista
5. Construir una función que permita visualizar por pantalla el valor del campo matricula de todos los elementos de una lista dinámica. El parámetro de la función es la dirección de memoria del primer elemento de la lista.
6. Construir una función que permita liberar el espacio en memoria de todos los elementos de una lista dinámica. El parámetro de la función es la dirección de memoria del primer elemento de la lista.
7. Construir una función que permita introducir un nuevo elemento en una lista dinámica. Los parámetros de la función son: (a) la dirección de memoria del primer elemento de la lista donde se vaya a realizar la inserción y (b) el valor del campo matricula del nuevo elemento de la lista. La inserción debe respetar el orden creciente (por número de matrícula) de los elementos de la lista previamente existentes.

8. Construir una función que permita eliminar un elemento ya existente en una lista dinámica. Los parámetros de la función son: (a) la dirección de memoria del primer elemento de la lista donde se vaya a realizar la eliminación y (b) el valor del campo matrícula del elemento a eliminar de la lista. La eliminación debe respetar la estructura de tipo lista de los elementos restantes.

9. Construir un programa que, mediante un menú de opciones, permita a un usuario elegir la operación a realizar con una lista de números de matrícula:

- Indique la operación:
- 1 para visualizar lista.
- 2 para introducir nuevo elemento.
- 3 para eliminar elemento existente.
- 4 para buscar elemento en la lista.
- 0 para salir del menú.

La posibilidad de selección debe repetirse hasta que el usuario introduzca el valor 0

10. Codificar las declaraciones necesarias para construir una lista de listas de enteros. Es decir, una lista simple enlazada en la que cada elemento de la lista contiene un puntero que señala el primer elemento de otra lista simple enlazada de números enteros.

11. ¿Se puede crear una variable dinámica de 7 bytes? Escriba un programa que lo haga. Pruebe a mostrar su tamaño en memoria. ¿A qué tipo se podría convertir el puntero de referencia para que esta variable fuese útil?



# **Parte IV**

## **Material adicional**



Dentro del material de consulta se han incluido distintos temas que quedan fuera del alcance de un curso de fundamentos de programación, pero que resulta conveniente conocer o al menos saber que existe. Los contenidos no se pueden organizar en una secuencia de aprendizaje, pues muchos de ellos dependen de conceptos más avanzados o que se ven con posterioridad y otros son totalmente independientes. Así, la lectura de esta parte se puede hacer en cualquier orden a discreción del lector.

Respecto del temario de la asignatura Fundamentos de Programación en la ETSII - UPM todo el material que se recoge en esta parte queda oficialmente fuera de programa. No obstante es posible que los profesores lo incluyan en sus clases por el indudable interés que tienen algunos de estos temas.



# Capítulo 15

## Introducción

### 15.1. El precompilador de C

#### 15.1.1. Descripción

Un aspecto característico de los programas de C es el uso de *comandos del preprocessador*. Es importante destacar que:

1. El preprocessador se ejecuta antes de que se llame al compilador.
2. El preprocessador sólo reconoce y analiza sus propios comandos y, por lo demás, ignora las reglas sintácticas y el resto de limitaciones del lenguaje.
3. El preprocessador se limita a trabajar con bloques de texto. Realiza, entre otras, alguna de las siguientes operaciones:
  - a) Incrusta el contenido completo de otro archivo.
  - b) Sustituye un texto por otro dado.
  - c) Elimina o escoge el texto que se va a compilar a partir de alguna condición lógica.

#### 15.1.2. Comandos del preprocessador

Estas funcionalidades se logran a través del uso de los siguientes comandos básicos del preprocessador:

**#include**, este comando del preprocessador seguido por la ruta de un archivo entre comillas dobles (" ") o ángulos (< >) inserta todo el contenido del archivo en el lugar donde aparece el comando **#include**. Nótese que el texto entre las comillas dobles no es parte del lenguaje de C y, por tanto, no es necesario incluir la barra doble invertida en la ruta de archivo cuando sea necesario especificar la ruta o vía de acceso completa<sup>1</sup>.

**#define macro texto**, este comando tiene dos efectos. Por un lado hace que **macro** esté *definida* (ver ejemplo más adelante) y por otro provoca la sustitución de todas las ocurrencias de **macro** en el código fuente por el **texto** indicado. La inclusión de **texto** es opcional y, en ese caso, sólo se *define* la **macro**.

**#ifdef macro #else #endif**, este comando permite escoger el código que se va a insertar en el archivo que se pasa al compilador. Cuando la macro está definida se insertará el código hasta **#else**. En caso contrario se insertará el código desde **#else** hasta **#endif**.

**#if expresion #else #endif**, este comando permite escoger el código que se va a insertar en el archivo que se pasa al compilador. Cuando la expresión sea verdadera se insertará el código hasta **#else**. En caso contrario se insertará el código desde **#else** hasta **#endif**. La expresión tiene que ser constante.

Una vez aplicado, cualquier comando del preprocessador se retira del código fuente antes de pasar el resultado al compilador.

Existen opciones dentro de estos comandos del preprocessador y otros comandos de preprocessador que permiten hacer tratamientos de texto más complejos. El uso adecuado del preprocessador puede ser una herramienta muy útil y necesaria

<sup>1</sup>En todo caso se recomienda usar la barra normal, carácter /, ya que los compiladores para Windows reconocen este carácter como separador de carpetas en la especificación de una ruta.

para realizar programas versátiles. No obstante, *no* se debe abusar del preprocesador porque hace que los programas sean más difíciles de leer y, además, debe recordarse que el preprocesador no analiza sintácticamente el código y por tanto la aplicación intensiva de su funcionalidad puede dar lugar a numerosos errores.

En el ejemplo 15.1 se puede observar la estructura de un programa simple donde se usan comandos del preprocesador.

Ejemplo 15.1: Estructura de un programa

---

```

1  #include "stdio.h"
2
3
4  /* Comentario iniciales explicando
5   * el contenido o detalle del archivo */
6
7  /* Declaraciones de variables, constantes, tipos, ...
8   * y especialmente funciones */
9
10 int a;
11
12 enum Semana { Lunes, Martes, Miercoles, Jueves, Viernes };
13
14 /*#define INGLES*/
15
16 void saludo()
17 {
18     #ifdef INGLES
19         /* Solo se compila si se define INGLES */
20         printf("hello world\n");
21     #else
22         /* Solo se compila si NO se define INGLES */
23         printf("hola mundo\n");
24     #endif
25 }/*saludo*/
26
27 int main()
28 {
29     /*Declaraciones */
30     enum Semana s;
31
32     /* A partir de aqui, sentencias */
33     saludo();
34     s = Lunes;
35     printf("hoy es el: %d\n", s);
36
37     return 0;
38 }/*main*/

```

---

Salida por pantalla en la ejecución:

```

hola mundo
hoy es el: 0

```

En este ejemplo se observan las siguientes características de estructura:

1. En primer lugar aparecen muchos comentarios distribuidos por todo el código fuente. La situación o contenido de un comentario es irrelevante para el sistema: todos son ignorados por el preprocesador y por el compilador.
2. El comando del preprocesador `#include` no se encuentra al principio del programa. Por costumbre y buena práctica normalmente los comandos `#include` se insertan al principio del archivo de código fuente. Por buena práctica, no por obligación, debe recordarse que el preprocesador realiza su tratamiento de texto antes de pasar al compilador y, estrictamente hablando, sus comandos quedan aparte del análisis sintáctico y se pueden incluir en cualquier punto del código fuente.
3. En este programa se ha declarado una variable, `a`, y un tipo de dato, `Semana`, y se ha declarado e implementado dos funciones, `saludo` y `main`. En un programa en C se permite declarar cualquier número de funciones, lo normal

es encontrar muchas funciones, especialmente en programas largos. Esa estructura es la habitual de los programas en C: una serie de declaraciones de elementos de programación, especialmente funciones. Para las funciones la declaración se complementa con un bloque entre llaves llamada implementación.

4. Es importante observar que hay una variable, `s`, declarada dentro de la función `main`. A todos los efectos esa variable pertenece (es interna) de la función `main` y no forma parte de la estructura del programa. Lo mismo ocurre con las sentencias dentro de `main` (de las llaves de `main`), todo ello forma parte de la función.
5. Finalmente se recomienda eliminar las marcas de comentario del código `#define`, compilar y ejecutar el programa para comprobar que es lo que ocurre.
6. En este ejemplo se muestra un ejemplo de uso de otro comando del preprocesador `#ifdef #else #endif`, tal y como se ha compilado este ejemplo, el texto (se han mantenido los comentarios para facilitar la localización del texto):

---

Ejemplo 15.2: Extracto con comandos ifdef

---

```

1     printf("hello world\n");
2     #else
3     /* Solo se compila si NO se define INGLES */
4     printf("hola mundo\n");
5     #endif
6 }/*saludo*/

```

---

se transforma por la acción del preprocesador en:

---

Ejemplo 15.3: Resultado del comando ifdef

---

```

1     #endif
2 }/*saludo*/

```

---

## 15.2. Programa de utilidad para compilación: make

### 15.2.1. Instalación de GNU Make en Windows

1. Dirección de GNU Make: <http://gnuwin32.sourceforge.net/packages/make.htm><sup>2</sup>
2. En esa misma página hacia la mitad en la sección de *Download* hay que descargar: *Complete packages except source Setup*<sup>3</sup>.
3. Una vez descargado, ejecutar (guardando o no), al arrancar el instalador, seleccionar las opciones por defecto, excepto la aceptación de licencia.

### 15.2.2. Make

El proceso de compilación en un programa en C puede ser en general muy complejo. Puede incluir un gran número de archivos de código fuente, incluir archivos de distintas librerías, enlazar dichas librerías, tener distintas opciones de optimización y depuración y otras muchas operaciones.

Para gestionar este proceso existen distintas herramientas que sirven para guardar las opciones de compilación, el orden de compilación y, en general, el resto de detalles necesarios durante el proceso de construcción del programa. Una de estas herramientas se llama *make*. En concreto, se va a tomar como referencia la versión de *make* del proyecto GNU.

*Make* gestiona el proceso de compilación mediante el concepto de regla (*rule*), una regla es un paso dentro del proceso de compilación que está formada por prerequisitos (*prerequisites*), objetivo (*target*) y comandos (*recipes*). La idea subyacente es que el proceso de construcción del programa se basa en la aplicación de una serie de comandos que producen resultados intermedios u objetivos a partir de archivos previos o prerequisitos. El objetivo final de este proceso es precisamente el programa que se quiere construir.

La sintaxis y capacidades de *make* son múltiples y complejas debido a que incorpora funcionalidades propias de los lenguajes de programación como variables o condicionales. No obstante, se puede configurar el proceso de compilación de una pequeña aplicación de forma muy sencilla y que se mostrará más adelante.

### 15.2.3. Uso de make para compilar y ejecutar

Como se ha comentado, el uso y escritura de archivos *makefile* puede ser muy complejo. No obstante, para la compilación de archivos sencillos se puede hacer un *makefile* muy simplificado que sirve para compilar y ejecutar un programa como se describe en el ejemplo.

#### En Windows

Se crea un archivo de nombre *makefile.mk* tal y como se haría si se escribiera un archivo de código fuente con el contenido que se muestra en: 15.4.

Ejemplo 15.4: makefile.mk

---

```

1
2
3 all: primero.exe primero.out
4
5 primero.out : primero.exe
6      primero > primero.out
7
8 primero.exe : primero.c
9      gcc primero.c -Wall -ansi -pedantic -o primero.exe
10
11 clean:
12      del *exe *.out

```

---

A continuación se llama a *make* con el siguiente comando en el *prompt* del sistema:

---

```
make -f makefile.mk
```

---

<sup>2</sup>Visto por última vez el 7 de diciembre de 2011

<sup>3</sup>Última versión vista: 3.81

Este comando compila y ejecuta el programa cada vez que se modifique el código fuente o solo ejecuta el programa si el código fuente no se ha modificado.

Es importante destacar que la indentación del archivo se tiene que poner tal y como aparece en el recuadro y que esta indentación se hace con el tabulador (no sirven espacios en blanco). De hecho, el tabulador aparece resaltado con una línea horizontal.

### En GNU - Linux

Se crea un archivo de nombre *makefile* tal y como se haría si se escribiera un archivo de código fuente con el contenido que se muestra en: 15.5.

A continuación se llama a *make* con el siguiente comando en el *prompt* del sistema:

Ejemplo 15.5: makefile

---

```
1
2
3 all: primero.exe primero.out
4
5 primero.out : primero.exe
6 _____./primero.exe > primero.out
7
8 primero.exe : primero.c
9 _____gcc primero.c -Wall -ansi -pedantic -o primero.exe
10
11 clean:
12 _____rm *exe *.out
```

---

A continuación se puede llamar a *make* escribiendo el siguiente comando en la línea de comandos:

```
make
```

Este comando compila y ejecuta el programa cada vez que se modifique el código fuente o solo ejecuta el programa si el código fuente no se ha modificado. Por defecto *make* utiliza un archivo llamado *makefile*. Por este motivo no es necesario indicarlo.

Es importante destacar que la indentación del archivo se tiene que poner tal y como aparece en el recuadro y que esta indentación se hace con el tabulador (no sirven espacios en blanco).

#### 15.2.4. Alternativas a make

En la actualización existe varias herramientas que pueden sustituir con ventaja a *make*: mejoran la sintaxis de las instrucciones de compilación, se pueden usar en distintas plataformas y/o están integradas con distintos entornos de desarrollo. Dos de las más significativas son **CMake** y **Ant**.



# Capítulo 16

# Elementos Básicos de programación

## 16.1. Otros tipos de datos

En los lenguajes de programación es frecuente encontrarse con tipos de datos con un número finito de posibilidades. Por ejemplo, valores booleanos (*true*, *false*), días de la semana (lunes, martes, miércoles, jueves, viernes, sábado, domingo) o el caso especial de los punteros.

En general todos estos tipos de datos se asimilan a un número entero mediante el empleo de una tabla (estandarizada o no) y se codifican como el correspondiente entero (sin signo). Por este motivo se suelen considerar como datos numéricos aunque estrictamente no sean números.

### 16.1.1. El tipo enumerado

No existe un tipo enumerado como tal, el programador debe definir sus propios tipos enumerados concretos mediante la elección de una serie de constantes con identificador cuyos valores son números enteros. El tipo enumerado se utiliza para representar información con un número finito de alternativas. Su tamaño y codificación son equivalentes a la de un número entero `int`. Los enumerados se definen mediante la palabra reservada `enum` y una serie de constantes definidas por el programador cuyos valores son números enteros y se codifican como tales. La sintaxis de declaración de un enumerado es:

```
enum Tipo_E { C1 = num_ent1, C2, C3, Cn };
enum Tipo_E id_var;
```

Donde `enum` es una palabra reservada que indica que se va a declarar un enumerado, `enum Tipo_E` es el nombre del tipo enumerado, los identificadores `Cn` son los posibles valores del enumerado, `num_ent1` es el equivalente numérico de `C1` y `id_var` es el identificador de una variable del tipo enumerado.

Entre llaves se escribe una lista de identificadores separados por comas que son las constantes de enumerado. Cada una de ellas opcionalmente se puede inicializar a un valor entero añadiendo `=` y el valor.

### 16.1.2. Tipo `size_t`

El tipo `size_t` es simplemente un tipo entero sin signo que se utiliza en contextos relacionados con el tamaño de algún dato simple o compuesto. Su uso es equivalente al de cualquier otro entero positivo, por ejemplo `unsigned long`, aunque su rango de representación puede ser mayor que `unsigned long`.

Es frecuente encontrar operadores o funciones en el estándar que devuelven o utilizan como parámetros datos de tipo `size_t`, por ejemplo: `sizeof` o `malloc`.

## 16.2. Tipos alfanuméricos y codificación de texto

**Estándares de 8 bits**, estos sistemas son la evolución del ASCII en los que se utilizan los lugares libres de esa tabla (lo que se llama la parte alta o ampliada de la tabla) para codificar caracteres propios del idioma local. Dentro de estos estándares se encuentran las tablas regionales de Windows (*Windows-1252*) o el estándar de caracteres de Europa Occidental (*ISO-8859-15*).

**Estándar UNICODE**, este estándar consiste en la unificación en una gran tabla de tablas de prácticamente la totalidad de caracteres que se encuentran en todos los idiomas del mundo y de una buena cantidad de caracteres gráficos (dibujos) que no forman parte de ningún idioma concreto. La codificación mediante UNICODE se realiza generalmente mediante dos sistemas, el UTF-8 (de 1 a 4 bytes) y el UTF-16 (de 2 a 4 bytes), el número de bytes depende del sistema y de la tabla donde se encuentren. Una propiedad importante es que la representación de la parte baja de la tabla ASCII (básicamente, las letras en inglés, los números y los signos de puntuación más habituales) coincide en UTF-8 y ASCII.

Una limitación importante de las codificaciones de texto en 8 bits es que no se puede determinar la tabla que se ha empleado a partir de los valores de la codificación. En UNICODE sí se puede debido a que utiliza determinadas secuencias de valores que permiten identificar la codificación del archivo.

En C no se presupone una codificación predeterminada. Normalmente se puede suponer que se utiliza la tabla de caracteres de Europa Occidental (*ISO-8859-15*), también llamada Latin-9 o una tabla equivalente en Windows. En GNU-linux la codificación por defecto es UTF-8.

Para facilitar el uso de codificación de texto en dos bytes (como en UTF-16), C incorpora un tipo de dato que se llama `wchar_t`. Básicamente este tipo es un carácter, es decir, sirve para definir cadenas alfanuméricas y permite operaciones propias de los texto, pero los caracteres ocupan 2 bytes en vez de 1. Esto significa que mezclar `char` y `wchar_t` no sea inmediato.

## 16.3. Funciones de entrada y salida estándar con formato

### 16.3.1. Aspectos Generales

La librería estándar de C proporciona funciones para realizar operaciones de entrada y salida de datos respecto de los programas. Estas funciones se encuentran declaradas en el archivo de cabeceras `<stdio.h>`. Entre estas funciones se encuentran las funciones de entrada - salida con formato. Se refieren a operaciones de entrada - salida en donde la información se encuentra en forma de texto, pero se pueden extraer o mostrar (entrada o salida) datos de todo tipo. La conversión necesaria de texto al tipo de dato que corresponda o viceversa se realiza dentro de las propias funciones, sin que los programadores se tengan que ocupar de los detalles de esta transformación.

Existen tres funciones de entrada de datos distintas según la fuente de información:

1. `scanf`, la fuente de información es la entrada estándar del programa. Normalmente el teclado.
2. `sscanf`, la fuente de información es la cadena alfanumérica que se pone como el primero de los argumentos.
3. `fscanf`, la fuente de información es un archivo dado como el primero de los argumentos.

Aunque las funciones son parecidas en comportamiento, en el caso de `sscanf` se debe tener en cuenta que, al no ser estrictamente hablando un canal, la lectura siempre empieza por el principio de la cadena, esto no ocurre a si al leer desde teclado o desde un archivo donde los caracteres leídos no se vuelven a leer en sucesivas llamadas a `scanf` o `fscanf`.

Hay tres funciones de salida:

1. `printf`, la información se escribe sobre la salida estándar del programa. Normalmente la pantalla.
2. `sprintf`, la información se escribe sobre la cadena que se proporciona como el primero de los argumentos.
3. `fprintf`, la información se escribe en un archivo dado como el primero de los argumentos.

Todas las funciones de entrada y salida de datos con formato necesitan la siguiente información:

1. El formato del texto que se va a leer o escribir. Es decir, una cadena alfanumérica donde se mostrará en general como queda el texto.
2. Respecto de los datos que hay que convertir: el tipo que se necesita y su lugar en el texto.

Toda esta información se escribe en el programa mediante los argumentos (los datos entre paréntesis y separados por comas) de las llamadas a las funciones `printf` y `scanf`. La sintaxis de los especificadores de formato en todas las variantes de `scanf` o `printf` es el mismo y en adelante se hará referencia solo a estas dos funciones. El primero de los argumentos de ambas funciones se denomina formato, es de tipo cadena alfanumérico (un texto) y normalmente se escribe como una cadena literal constante (con comillas dobles). Dentro de este argumento encontramos:

- Letras y caracteres normales que se utilizan para saber la forma del texto que se va a procesar.
- Especificadores de formato constituidos por el carácter tanto por ciento (%) seguido por otros caracteres que especifican la manera en que se tiene que realizar la conversión de los tipos de datos correspondientes.

Para usar correctamente las funciones de entrada y salida es necesario conocer cómo se escriben estos especificadores de formato.

### 16.3.2. Especificadores de formato simple

Los especificadores de formato más simples y comunes a las funciones `printf` y `scanf` son los siguientes:

Caracteres	Tipo al que corresponden
%d ó %i	int
%f	float
%c	char
%s	cadena alfanumérica ( <code>char[]</code> )

El resto de especificadores son más complejos o son diferentes para `printf` y `scanf`.

### 16.3.3. Uso simplificado de la función printf

La función **printf** sirve para mostrar por pantalla (en general para obtener datos o resultados del programa). En su uso más sencillo sirve para mostrar una cadena de texto, por ejemplo:

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

muestra en la pantalla el texto entre las comillas dobles seguido de un salto de línea (`\n`). Este uso es bastante limitado porque la información que se muestra es siempre la misma: la constante literal que hemos escrito entre los paréntesis. Si queremos mostrar el valor de los datos (variables) de nuestro programa debemos utilizar los especificadores de formato en la cadena y argumentos extra con las variables o expresiones cuyo valor queremos mostrar. En este otro ejemplo:

```
printf("La suma de tres y doce es%d.\n", 3+12);
```

se puede observar el uso de uno de los especificadores de formato indicados más arriba. Los caracteres del especificador de formato tipo entero (`%d`) se colocan en el lugar apropiado de texto que se quiere mostrar. Estos caracteres no se mostrarán en pantalla. En su lugar, se mostrará el valor del segundo de los argumentos de la función **printf**. En este caso un 5. Cualquier otro carácter se mostrará tal cual aparece en el texto, exactamente igual que en el uso anterior. Así el mensaje por pantalla será:

**La suma de tres y doce es 15.**

El programa ha calculado la expresión numérica `3+12`, ha obtenido el número (entero) 15, lo ha convertido en el texto “15” y lo ha colocado en el texto de salida en el lugar indicado por el especificador de formato.

Es posible mostrar más de un valor en un solo **printf**. Para ello es necesario colocar más especificadores de formato en el texto y más argumentos. En este ejemplo:

```
printf("El monomio es%f*x^%i\n", 3.98, 3);
```

podemos ver dos especificadores, uno para datos de tipo **float** o **double** (`%f`) y otro para datos de tipo entero (`%i`). Como se puede observar no es necesario que los especificadores se encuentren separados del resto de caracteres por espacios blancos. Por otro lado, el orden en que aparecen los especificadores de formato en la cadena de texto se corresponde con el orden de los argumentos que se deben escribir a continuación. En este ejemplo, el primer especificador (`%f`) se corresponde con el argumento 3.98 y el segundo (`%i`) con 3.

### Particularidades del uso de especificadores con printf

En el caso de usar **printf** encontramos los siguientes comportamientos de los especificadores de formato especiales para esta función:

- El especificador `%f` se puede utilizar para **float** o **double** indistintamente sin ningún problema. Pero para **double** no mostrará todos los decimales que el número tiene en memoria.
- Se puede utilizar un dato y un especificador que no se corresponda con el tipo del dato si el dato se puede convertir al tipo del especificador. Este uso *no se recomienda* y puede producir avisos del compilador.
  - Un dato tipo **char** con un especificador `%i` o `%d` produce el número en la tabla ASCII del carácter.
  - Un dato de tipo número entero con un especificador `%c` produce el carácter correspondiente al número según la tabla ASCII.

### 16.3.4. Uso simplificado de la función scanf

Las llamadas a **scanf** normalmente utilizan el argumento de formato, con, al menos, un especificador de formato y un argumento extra que consiste básicamente en una variable. La manera en que el programa recibe la información (por ejemplo, un número que teclea el usuario) es a través del valor de una variable. Así pues, cuando **scanf** termina y se ejecutan las sentencias posteriores el valor de la variable es la información recibida (escrita en el teclado).

Para usar la función **scanf** sin dominar el uso de paso de parámetro de tipo puntero es necesario aprender dos reglas muy sencillas. Para usar **scanf**:

1. con datos de tipo simple (**int**, **char**, **double**, **float**) se debe utilizar siempre como argumento una variable del tipo correspondiente precedida por el signo *umpersand* (`&`).

2. con datos de tipo cadena alfanumérica (por ejemplo, una variable declarada como `char cad[30];`) se debe usar directamente el identificador de la variable.

En este ejemplo:

```
int a; float b;
scanf("%i%f", &a, &b);
```

se piden por teclado dos valores: un entero y uno en coma flotante. Como antes, el (`%i`) se corresponde por su orden de aparición con `&a`, el primer argumento que sigue al formato. Los caracteres que el usuario introduzca en primer lugar se convertirán (si es posible) a un número entero y ese valor se guardará en la variable `a`. Análogamente los caracteres que se tecleen después se convertirán a un número en coma flotante (`%f`) y se guardarán en `b`. Al leer la información el programa se saltará todos los blancos que el usuario escriba. Pero, si introduce algún carácter que no pueda formar parte de los números se producirá un error.

En este otro ejemplo:

```
char cadena[256];
scanf("%s", cadena);
```

se puede observar el uso de una variable tipo cadena alfanumérica (sin `&`). La función `scanf` lee caracteres y los pone en la cadena alfanumérica hasta que encuentra un espacio (una palabra).

### Particularidades del uso de especificadores con `scanf`

El uso de `scanf` debe ser más cuidadoso que el de `printf`, en concreto es necesario saber que:

- Los argumentos de `scanf` son siempre variables (con o sin `&`).
- El tipo de los especificadores de formato y de las variables pasadas como argumentos debe corresponder *exactamente*.
- Se utilizan los siguientes especificadores particulares:

Especificador de formato	Tipo que corresponde
<code>%lf</code>	<code>double</code>
<code>%u</code>	<code>unsigned int</code>
<code>%hi</code>	<code>short int</code>
<code>%hu</code>	<code>unsigned short int</code>
<code>%li</code>	<code>long int</code>
<code>%lu</code>	<code>unsigned long int</code>

### Caracteres distintos de especificadores de formato

En el uso de `scanf` más básico no se suele usar otros caracteres en el texto de formato distintos de espacios en blanco y especificadores de formato. No obstante, es conveniente saber cómo interpreta `scanf` estos caracteres:

- Un espacio en blanco (cualquier espacio en blanco, incluido tabuladores o salto de línea y cualquier número de ellos) en el texto de formato le indica a `scanf` que debe saltarse todos los espacios en blanco (incluidos tabuladores y saltos de línea) hasta que encuentre un carácter distinto de blanco.
- Cualquier otro carácter que se encuentre en el texto de formato indica a `scanf` que ese carácter se debe encontrar escrito en la información que se escriba.

Por ejemplo, si el texto de formato es: “`%f , %i`”, `scanf` interpreta que se debe escribir: un número en coma flotante, seguido de cualquier número de espacios en blanco (incluido ningún espacio en blanco), seguidos de un carácter coma (‘,’), seguido de cualquier número de espacios en blanco y seguido de un número entero.

### 16.3.5. Uso avanzado de la función printf

La función `printf` permite dar formato a los datos de salida de forma muy detallada. Para ello es necesario conocer en profundidad la forma en que se pueden escribir los especificadores de formato.

La sintaxis de los especificadores de formato consta de:

`%[opciones] [ancho] [.cifras] [tamaño]tipo`

donde los corchetes indican que los campos correspondientes son opcionales (se pueden indicar o no) y cada uno se corresponde con:

`%` indica que se va a escribir un especificador de formato. Si se necesita escribir `%` en la salida se colocan dos caracteres tanto por ciento (`%%`).

**opciones** con el signo menos (`-`) se justifica el texto a la izquierda; con signo más (`+`) se muestra el signo en los números positivos; etc.

**ancho** si se pone un número, el argumento se muestra en ese número de caracteres como mínimo (útil para dar formato por columnas); si se pone un asterisco (`*`), el ancho es un número entero que se pasa como un argumento extra delante del argumento cuya conversión se está definiendo.

**cifras** si se pone un número indica: para número enteros el número de cifras a escribir y para número en coma flotante el número de decimales. Esto es especialmente útil (y necesario) si se quiere conocer con precisión los números guardados de tipo `double`, el número de cifras por defecto no basta para representar el número con la precisión que se guarda en memoria. Si se pone un asterisco (`*`), como antes, hay que pasar un argumento adicional delante del número para indicar las cifras a tener en cuenta.

**tamaño** normalmente una letra ele minúscula (`l`) o hache minúscula (`h`). Se utiliza para modificar los tipos básicos según su tamaño, por ejemplo: `%li` para `short int`.

**tipo** los indicados anteriormente. No obstante hay algunos tipos extras interesantes: para números enteros, el tipo letra `o` de octal (`o`) convierte el número a ese sistema de numeración, lo mismo para `x` de hexadecimal; para número en coma flotante la letra `e` de exponente (`e`) indica que se use notación científica.

Otros detalles de interés al usar la función `printf` son:

- El valor de retorno de la función `printf` es el número de caracteres totales que se han escrito. Si el valor es negativo indica que se ha producido un error. Normalmente este valor se descarta, pero esta información puede ser útil cuando se desea hacer una comprobación exhaustiva de errores o en casos muy especiales del uso de la función.
- Debe tenerse en cuenta que la cadena que indica el formato puede ser una variable. Normalmente las llamadas a `printf` tienen como argumento una constante literal, pero esto no es obligatorio.
- La implementación de la función `printf` de la librería estándar de C en cada sistema hace la conversión entre el carácter `'\n'` que es el carácter número 10 en la tabla ASCII al formato del salto de línea nativo en la plataforma correspondiente. Por ejemplo, en Windows se escriben dos caracteres (13 y 10) y en gnu-linux se escribe un único carácter 13.
- Cuando un argumento no se corresponde con el tipo indicado por el especificador de formato el resultado es difícil de predecir. En general, lo que se muestra es el resultado de una conversión forzada entre tipos, pero es difícil determinar la conversión que se está realizado y por tanto el resultado final. Muchos errores en programas se pueden evitar eligiendo especificadores de formato que coincidan estrictamente con el tipo de la variable que se va a mostrar. La única excepción a esta regla es la utilización mezclada de especificadores de formato y variables de tipo `char` y entero, esta mezcla sirve para encontrar el número de una letra en la tabla ASCII o la letra que se corresponde con un número en la misma tabla.
- Para `fprintf` la escritura sobre archivo no se produce inmediatamente sino que los datos quedan almacenados en un *buffer* asociado al archivo. La llamada a la función `fflush(f)` vuelca el contenido del *buffer* al archivo en disco. La llamada a esta función puede ser necesaria en algunos casos. Por ejemplo, para evitar incoherencias cuando se realizan sucesivas operaciones de lectura y escritura sobre el mismo archivo. En este caso debería llamarse a la función `fflush` después de la operación de escritura para asegurar su almacenamiento en el archivo de disco antes de la operación de lectura.

### 16.3.6. Uso avanzado de la función scanf

La función **scanf** es un ejemplo frecuente de cómo se usan parámetros de tipo puntero en muchas funciones de la librería estándar (y de otras librerías). Dado que en C el paso de parámetros siempre es por valor, no es posible cambiar el valor de un argumento dentro de una función... excepto que pasemos como argumento el puntero de una variable. En este caso, la variable se puede modificar porque la variable no es el argumento, el argumento es su puntero y ese no cambia.

Por tanto para conocer en detalle cómo funciona la función **scanf** es necesario conocer cómo se produce el paso de variables a una función mediante su puntero correspondiente. Esta es la base para entender **scanf**. No obstante vamos a revisar dos puntos que tenemos que tener muy claros en el uso de **scanf**:

1. La función **scanf** no puede comprobar si hemos pasado el puntero a una variable válida. La función recibe un puntero que debe apuntar a una variable cuyo tipo debe coincidir completamente con el tipo indicado por el especificador de formato. Esta obligación corre a cuenta del programador que utiliza la función **scanf**, su incumplimiento puede suponer errores muy graves de funcionamiento del programa. Insistimos que esta obligación tiene dos partes: la variable debe existir y la variable tiene que ser del tipo indicado por el especificador de formato.
2. Las variables de tipo vector, entre ellas especialmente las cadenas alfanuméricas, son convertibles directamente a puntero (sin poner **&**), pero cuando se hace la conversión a puntero deja de conocerse su tamaño. Es un error muy frecuente no asegurar que la variable de tipo cadena alfanumérica es capaz de almacenar la información que va recibir. La capacidad de la cadena debe ser mayor que el número de caracteres leídos. Es una muy buena costumbre utilizar, para este caso, el modificador que afecta al número de caracteres leídos tal y como se explica más adelante.

En el uso simplificado de la función **scanf** no se han incluido todos los especificadores de formato y sus modificadores, la sintaxis completa de los mismos es:

**%[\*] [ancho] [modificador]tipo**

donde:

**%** es el carácter que marca el inicio del especificador de formato.

**\*** es un carácter opcional que indica que el dato extraído de la entrada no se tiene que guardar en el resto de los argumentos, simplemente se ignora. Este modificador se usa para saltar información variable en la entrada.

**ancho** es un número que indica el número de caracteres máximo que debe leerse de la entrada. Es una limitación al número de caracteres no al tamaño de un número ni nada parecido. Este modificador es muy útil para procesar caracteres de entrada. La llamada a **scanf** que se muestra a continuación es totalmente segura dado que se van a leer como mucho 99 caracteres de la entrada. Estos 99 caracteres más el carácter nulo '**\0**' caben en la cadena declarada con tamaño 100.

```
char aux[100];
scanf("%99s", aux);
```

**modificador** sirve para modificar el tipo base de tipo entero o número en coma flotante para cubrir los tipos de tamaño especial. Puede ser una ele ( '**l**' ) o una hache ( '**h**' ) para tipos largos o cortos respectivamente. Por ejemplo: **%hi** es para **short int** y **%lf** es para **double**.

**tipo** los indicados anteriormente. Adicionalmente resulta muy útil un especificador de formato que sirve para leer un conjunto de caracteres. Para definir este conjunto se puede indicar los caracteres que lo forman (y entonces se leerán caracteres mientras que los mismo estén en la entrada) o los que no lo forman (y entonces se leerán caracteres hasta encontrar uno que no está en el conjunto, la lectura se detiene antes de leer el carácter que no forma del conjunto). Para definir un conjunto se puede poner: "**%[abcdef]**" que significa leer caracteres de la '**a**' a la '**f**'. También se puede poner "**%[a-f]**", no está recogido en el estándar de C, aunque lo reconocen la mayoría de los compiladores. También se puede poner una coma para separar rangos de caracteres. Para indicar los caracteres que no están en el conjunto se pone: "**%[^;,:]**" y en este caso se leerán hasta que se encuentre alguno de estos signos de puntuación. El acento circunflejo indica la negación. En el siguiente ejemplo: se lee una cadena formada por letras de la '**a**' a la '**f**' y dígitos del '**0**' al '**9**', después una cadena hasta el punto y coma, se quita el ';' de la entrada y se lee otra cadena:

```
char cad1[11], cad2[100], cad3[100];
scanf("%10[a-f,0-9] %99[^;];%99s", cad1, cad2, cad3);
```

También es importante conocer los siguientes detalles relacionados con la entrada de datos:

- Los datos que se escriben por teclado no se envían inmediatamente a la función **scanf**, antes es necesario hacer *flush* (operación que consiste en llevar los caracteres que se encuentran guardados en una estructura de datos temporal denominada *buffer* de entrada a la entrada propiamente dicha). Se puede hacer *flush* de dos maneras: desde el teclado, al pulsar la tecla *intro* se genera el carácter salto de línea y además se hace *flush*; desde el programa mediante una llamada explícita a la función **fflush**<sup>1</sup>.
- Cuando se leen varios datos seguidos se debe tener en cuenta que:
  - Los números se terminan de leer cuando se encuentra un carácter que no pueda formar parte del número, por ejemplo, una letra o un signo de puntuación distinto del punto decimal.
  - En los números en coma flotante la parte entero y fraccionaria se separa mediante un punto no una coma.
  - Las cadenas alfanuméricas se terminan de leer cuando se encuentra un espacio en blanco de cualquier tipo (barra espaciadora, tabulador, salto de línea).
  - Si se escribe un espacio en blanco (barra espaciadora, tabulador, salto de línea) en el formato que se pasa como primer argumento al **scanf** se saltan todos los espacios en blanco que aparezcan en la entrada hasta encontrar un carácter distinto de espacio en blanco. Solo es necesario poner un único espacio en blanco en el formato para saltar cualquier número de espacios en la entrada, incluido ningún espacio.
  - Si se escribe algún carácter distinto de un especificador de formato o un espacio en blanco, **scanf** busca que ese mismo carácter aparezca en la entrada y si no lo encuentra no continua procesando la entrada.
- El retorno de la función indica el número de datos (no de caracteres) leídos correctamente, es decir, se han leído de la entrada y se han podido convertir al tipo indicado por el especificador. Puede ocurrir que se produzcan errores cuando se intentan leer varios datos y en este caso el retorno indicará el número de datos que se han leído correctamente. Cuando se usa **fscanf** se puede llevar al final de archivo y se retorna un valor EOF. Es posible provocar el final de archivo de la entrada por teclado, se pueden usar las combinaciones de teclas *control + z* o *control + d* para lograrlo, pero no es frecuente hacer esto. Normalmente el valor del retorno se comprueba frente al número de datos que se desea leer, si coinciden significa que no se han producido errores.

---

<sup>1</sup>El estándar de C no determina el comportamiento de la función **fflush** sobre un canal de entrada. En este sentido, los desarrolladores de la librería estándar de C podrían dar a la función **fflush** algún otro comportamiento no esperado.

## 16.4. Los conceptos left-value y right-value

### 16.4.1. La asignación

Las operaciones de asignación son *binarias*, tal y como se ha mostrado en la sintaxis, tienen dos operandos, uno a la izquierda y otro a la derecha.

### 16.4.2. Left value y right value

Tal y como se ha descrito en capítulo de operadores, los datos de un programa se almacenan en memoria en un lugar concreto y con un tamaño determinado. En este sentido las variables tienen dos vertientes: por un lado son un contenedor, un lugar donde se coloca información y, por otro lado, son contenido, el valor concreto de la información. Para distinguir estos dos aspectos de las variables en C se habla de dos valores que reciben los nombres de izquierda y derecha por el lado que ocupan en una asignación.

**lvalue, l-value, left value**, en español *valor izquierdo*, o término de la izquierda. Un valor, resultado o expresión izquierda es aquel que se corresponde con una variable en cuanto a contenedor de información. Se utilizan para cambiar el valor guardado en una variable, actualizando la información a un nuevo dato concreto y borrando el anterior. Un valor izquierdo representa el lugar en memoria donde se guarda la variable. Sólo las variables tienen *l-value* aunque se pueden obtener *l-values* como resultado de algunas operaciones. Los identificadores de las variables estáticas de tipo de dato básico son los *l-value* más sencillos que existen y los más usados.

**rvalue, r-value, right value**, en español *valor derecho*, o término de la derecha. Un valor derecho se corresponde con un dato concreto. En el contexto de una variable el valor derecho es el contenido, es decir, el dato concreto en memoria, guardado en la variable correspondiente. Otros elementos de programación distintos de las variables tienen o son *r-values*. Las constantes o el resultado de una función también pueden ser *r-values*.

Como se puede deducir fácilmente, las variables (las estáticas y en general) pueden ser *l-value* y *r-value*, pero no los dos a la vez. Es decir, en una expresión, cada vez que aparece una variable se determina si aparece como *left* o *right value* a partir del lado de la asignación que ocupa. Naturalmente en el lado izquierdo de una asignación sólo puede aparecer un único *l-value*, lo que en la práctica reduce los elementos que pueden aparecer en el lado izquierdo exclusivamente a dos elementos: identificadores de variables estáticas y expresiones cuya última operación devuelve un *l-value*. Una manera sencilla y común de indicar esto último es que un operador tiene como resultado *una variable*.

## 16.5. Operadores especiales

### 16.5.1. Operadores especiales

En C existen una serie de operadores que son característicos de este lenguaje y que en otros lenguajes normalmente se consideran como sentencias o sencillamente no existen. Algunos de estos operadores son difíciles de leer, su uso no se recomienda, pero es conveniente conocer que existen.

**Operación condicional**, la operación condicional ternario se forma mediante dos símbolos de operación: el símbolo de cierre de interrogación ( ? ) y el signo de puntuación dos puntos ( : ). La operación condicional tiene tres operandos, el primero tiene que ser un escalar, para los otros dos puede haber distintas posibilidades. Lo más frecuente es que sea un tipo básico, pero no es obligatorio. El resultado de esta operación se calcula evaluando el primer operando, si es distinto de cero, 0, se evalúa el segundo y ese es el resultado de la operación; si el primer operando es igual a cero, se evalúa el tercer operando y ese es el resultado.

**Operador coma**, el operador coma tiene dos operandos de cualquier tipo, generalmente expresiones. Se evalúa la primera expresión u operando y su resultado se descarta, a continuación se evalúa la segunda expresión u operando y se utiliza como resultado. No se debe confundir con la coma de separación en otros contextos. En español se debe tener especial cuidado con este operador porque se confunde con la coma decimal. Por ejemplo, la expresión: 1,2 no es el número 1.2 sino el 2 (se descarta el 1).

En C existen otros elementos de programación que sintácticamente tienen la entidad de operadores aunque normalmente no se estudien como tales, son:

**Operación llamada a función**, este operador está formado por una pareja de paréntesis, apertura y cierre, ( ). En el desarrollo de un programa normalmente se realizan muchas llamadas a función sin que sea necesario considerar que, efectivamente, la llamada de la función es una operación. No obstante, esta categoría sintáctica hace posible, entre otras cosas, llamar a una función guardada en una variable tal y como se muestra en el capítulo de punteros a función.

**Operador indexación de matrices**, en inglés *array subscripting*, formado por dos corchetes, apertura y cierre, [ ]. Esta operación aplica al operando de la izquierda y que debe ser un puntero (o vector convertido a puntero) un desplazamiento dado por el segundo operando, el número entero resultado de la expresión dentro de los corchetes y devuelve la variable apuntada por el puntero desplazado. La operación v [ exp ] es absolutamente equivalente a \*(v + (exp)).

**Operador punto**, este operador toma como primer operando una variable de tipo estructura y como segundo operando el identificador de un campo o atributo de la estructura. Como resultado se obtiene el atributo como variable.

**Operador flecha**, este operador toma como primer operando un puntero a una estructura y como segundo operando el identificador de un campo o atributo de la estructura. Como resultado se obtiene el atributo como variable.

Finalmente existe un operador muy útil en muchas situaciones, pero que hay que manejar con extremo cuidado. Es un operador de bajo nivel que permite interpretar un dato mediante el uso de un tipo distinto al suyo. Este operador se emplea con cierta frecuencia para hacer conversiones entre tipos de datos numéricos (por ejemplo, pasar un número en coma flotante a entero) o conversiones entre punteros que apuntan a tipos distintos. Es muy importante prestar atención a los errores o avisos de compilador respecto de estas conversiones y estar muy seguros de la coherencia de la operación que estamos realizando.

**Operador conversión**, en inglés *cast*, también conversión explícita porque se escribe en la expresión donde aplica <sup>2</sup>. Este operador está formado por dos paréntesis, apertura y cierre ( ) con un tipo de dato entre ellos y se aplica al dato que se encuentra a la derecha de los paréntesis. El resultado es la conversión obligatoria del dato a la codificación indicada por el tipo entre paréntesis.

---

<sup>2</sup>en contraposición a conversión implícita, una conversión que se sobreentiende y se realiza sin que se escriba en la expresión

## 16.6. Sentencias de control especiales

En C se han definido algunas sentencias que sirven para modificar el flujo normal de un programa. Algunas se usan en bucles y otras en cualquier punto del programa. En cualquier caso son una manera de provocar un comportamiento distinto del esperado y en este sentido no se recomienda su uso.

### 16.6.1. La sentencia break en bucles

La sintaxis de la sentencia **break** es:

```
break;
```

La sentencia **break** sólo debe encontrarse en la sentencia selectiva **switch** (como se vió en el capítulo correspondiente) o en cualquiera de los bucles del lenguaje C (**while**, **for** y **do-while**). Al ejecutarse, deja el **switch** o bucle más interno (en caso de estar anidados) de todos en los que se encuentre, abandonando su ejecución. Por ejemplo, si un bucle tiene una sentencia compuesta donde se encuentra un **break**, entonces el bucle se interrumpe y pasa a la sentencia que aparezca a continuación. Si hubiese dos bucles anidados deja el bucle interno. En la figura 16.1 se muestra el diagrama de flujo de la sentencia **break** en combinación con un bucle **while**.

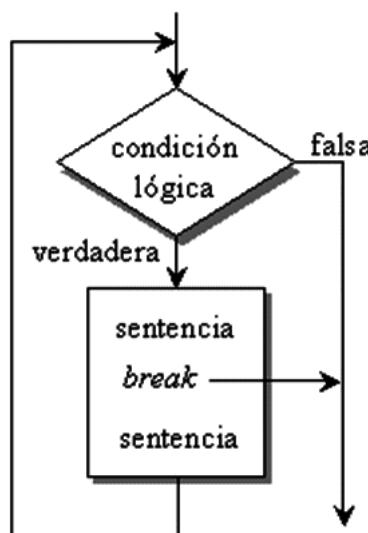


Figura 16.1: Diagrama de flujo de la sentencia de control **break** dentro de un bucle **while**

Cuando se emplea en combinación con un bucle, la sentencia **break** suele formar parte de una rama de una sentencia selectiva. En el ejemplo 16.1 se presenta la utilización de la sentencia **break** en combinación con un bucle **for**.

Ejemplo 16.1: Sentencia break en un bucle for

---

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int n;
7     for ( n = 0; n < 6; ++n ) {
8         if ( n == 3 )
9             break;
10        printf("En este bucle con break, n vale ahora %d\n", n);
11    }
12    printf("Fin del programa");
13    return 0;
14 }
```

---

Salida por pantalla en la ejecución:

```

En este bucle con break, n vale ahora 0
En este bucle con break, n vale ahora 1
En este bucle con break, n vale ahora 2
Fin del programa

```

Anidada dentro del bucle **for** hay una sentencia **if** que incluye una llamada a **break** si **n** vale 3. La ejecución de **break** hará finalizar inmediatamente la repetición de la sentencia del bucle donde se encuentre, terminará definitivamente la ejecución del bucle y continuará con la sentencia que pueda existir a continuación. Es una sentencia útil si se desea salir inmediatamente de un bucle al cumplirse una determinada condición. En este caso, cuando **n** toma el valor 3, el bucle finaliza y el último valor visualizado en pantalla será el anterior, el 3. La sentencia **break** siempre salta fuera del bucle hasta pasada la llave que cierra la sentencia a repetir.

### 16.6.2. La sentencia **continue**

La sintaxis de la sentencia **continue** es:

```
continue;
```

La sentencia **continue;** se emplea sólo dentro de la sentencia a repetir en bucles (**while**, **for** y **do-while**). Al ejecutarse la sentencia **continue**, se finaliza en ese punto dicha iteración y se continua con la siguiente repetición del bucle. En la figura 16.2 se muestra el diagrama de flujo de la sentencia **continue** en combinación con un bucle **while**.

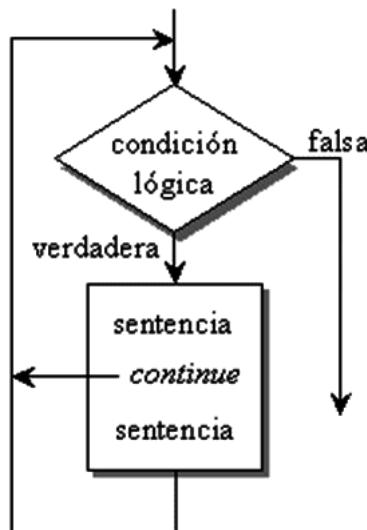


Figura 16.2: Diagrama de flujo de la sentencia de control **continue** dentro un bucle **while**

Como ocurre con la sentencia **break**, la sentencia **continue** suele formar parte de una rama de una sentencia selectiva dentro del bucle. El código fuente 16.2 del programa **rompe.c** presenta un ejemplo de utilización de la sentencia **continue** en combinación con un bucle **for**.

---

#### Ejemplo 16.2: Sentencia **continue** en un bucle **for**

```

1 #include <stdio.h>
2
3
4 int main()
5 {
6     int n;
7     for ( n = 0; n < 6; n = n+1 ) {
8         if ( n == 3 )
9             continue;
10        printf("En este bucle con continue, n vale ahora %d\n", n);
11    }
12    printf("Fin del programa");

```

```

13     return 0;
14 }
```

---

Salida por pantalla en la ejecución:

```

En este bucle con continue, n vale ahora 0
En este bucle con continue, n vale ahora 1
En este bucle con continue, n vale ahora 2
En este bucle con continue, n vale ahora 4
En este bucle con continue, n vale ahora 5
Fin del programa
```

En el bucle **for** se incluye una sentencia **continue** que no produce la finalización total del bucle pero que interrumpe y termina la repetición actual. Cuando el valor de **n** toma el valor 3, el programa salta al final del bucle y continúa ejecutándose el bucle ignorando la sentencia **printf()** que iba a continuación cuando **n** tomaba el valor 3. Posteriormente el bucle verifica el test correspondiente para ver si continúa o no su propia ejecución.

### 16.6.3. La sentencia goto

La sintaxis de **goto** o *salto incondicional* es:

```
goto etiqueta;
```

Se compone de la palabra reservada **goto** seguida de un identificador o etiqueta que indica el lugar desde donde se desea continuar la ejecución del código. El identificador recibe el nombre más específico de etiqueta. La etiqueta se puede encontrar en cualquier punto de la función con su correspondiente identificador seguido del carácter de dos puntos. Puede saltarse dentro de una función o fuera de un bucle pero no puede saltarse dentro de un bucle.

El código fuente 16.3 del programa **salto.c** incluye unos ejemplos de utilización de la sentencia **goto**. Este programa es un completo lío, pero es un buen ejemplo de porqué los desarrolladores de software están intentando eliminar el uso generalizado de la sentencia **goto**. Sólo hay un lugar razonable de empleo de **goto** en el programa: el salto fuera del bucle. Aunque podría hacerse de otras maneras e ir saltando sucesivamente fuera de los tres bucles, una única sentencia **goto** lo hace de una vez de una forma muy concisa. Algunos programadores defienden que la sentencia **goto** no debería ser empleada nunca bajo ninguna circunstancia. En cualquier caso no debería abusarse de su empleo. Existen libros completos escritos que no emplean la sentencia **goto**, son libros que siguen la denominada *Programación Estructurada*.

Ejemplo 16.3: Sentencia goto

```

1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a, b, c;
7     goto inicio;
8     masalla:
9     printf("Otra linea de este lio.\n");
10    goto elfinal;
11    inicio:
12    for(a = 1 ; a < 4 ; a = a + 1)
13    {
14        for(b = 1 ; b < 3 ; b = b + 1)
15        {
16            for(c = 1 ; c < 3 ; c = c + 1)
17            {
18                printf("A = %d  B = %d  C = %d\n", a, b, c);
19                if ((a + b + c)>5) goto fuera;
20            }
21        }
22    }
23    fuera: printf("Basta ya.\n");
24    printf("\nPrimera linea del codigo spaghetti.\n");
25    goto alli;
```

```
26     aqui:  
27     printf("Tercera linea del codigo spaghetti.\n");  
28     goto masalla;  
29     alli:  
30     printf("Segunda linea del codigo spaghetti.\n");  
31     goto aqui;  
32     elfinal:  
33     printf("Ultima linea del codigo enrevesado.\n");  
34     return 0;  
35 }
```

---

Salida por pantalla en la ejecución:

```
A = 1  B = 1  C = 1  
A = 1  B = 1  C = 2  
A = 1  B = 2  C = 1  
A = 1  B = 2  C = 2  
A = 2  B = 1  C = 1  
A = 2  B = 1  C = 2  
A = 2  B = 2  C = 1  
A = 2  B = 2  C = 2  
Basta ya.
```

```
Primera linea del codigo spaghetti.  
Segunda linea del codigo spaghetti.  
Tercera linea del codigo spaghetti.  
Otra linea de este lio.  
Ultima linea del codigo enrevesado.
```

## 16.7. Funciones avanzadas de string.h

### 16.7.1. Cuadros de funciones con string

En las tablas 16.1 y 16.2 se muestran algunas funciones extra de la librería estándar de C que trabajan con cadenas alfanuméricas.

Cuadro 16.1: Funciones avanzadas con cadenas alfanuméricas en **string.h**

Declaración	Descripción
<code>size_t strlen(const char * str);</code>	Devuelve un entero positivo con la longitud de la cadena, es decir, el número de caracteres hasta el carácter nulo.
<code>char * strcpy(char * destino, const char * fuente);</code>	Copia la cadena <b>fuente</b> a la cadena <b>destino</b> y devuelve de nuevo <b>destino</b> . La copia incluye el carácter nulo. La cadena de destino debe tener el tamaño suficiente para que quepa la cadena origen.
<code>char * strcat(char * destino, const char * fuente);</code>	Añade la cadena <b>source</b> a la cadena <b>destino</b> . El carácter nulo de la cadena destino se sustituye por el primer carácter de la cadena <b>fuente</b> . La cadena origen se añade hasta el final, incluido su carácter nulo. La cadena <b>destino</b> tiene que tener suficiente tamaño para guardar su contenido anterior y el nuevo.
<code>int strcmp(const char * str1, const char * str2);</code>	Compara dos cadenas carácter a carácter y devuelve si su contenido es igual (devuelve cero), devuelve mayor que cero si el primer carácter de la primera cadena es mayor o menor que cero en caso contrario. Para esta comparación se supone que <b>char</b> no tiene signo.
<code>char * strtok(char * str, const char * delim);</code>	Devuelve un puntero a distintas partes de una cadena <b>str</b> delimitadas por los caracteres en <b>delim</b> . En la primera llamada <b>str</b> es la cadena que se desea dividir en las sucesivas llamadas <b>str</b> debe ser NULL. El resultado es un puntero a cada una de las partes de la cadena. Cuando ya no quedan partes que separar en la cadena se devuelve NULL.
<code>char * strpbrk(const char * str1, const char * str2);</code>	Es una función de búsqueda, devuelve un puntero al primer carácter en <b>str1</b> que coincida con alguno de los caracteres en <b>str2</b>
<code>char * strchr(const char *str, int c);</code>	Igual que en el caso anterior, pero busca un sólo carácter que se pasa en el parámetro <b>c</b> .

Cuadro 16.2: Funciones con cadenas alfanuméricas en **stdlib.h**

Declaración	Descripción
<code>double atof(const char *nptr);</code>	Convierte una cadena que contiene la escritura de un número en coma flotante en el valor numérico correspondiente.
<code>int atoi(const char *nptr);</code>	Convierte una cadena que contiene la escritura de un número entero en el valor numérico correspondiente. Si el número es mayor o menor que el rango se devuelve <b>INT_MAX</b> o <b>INT_MIN</b> . Si se produce un error se devuelve cero.
<code>long int atol(const char *nptr);</code>	Convierte una cadena que contiene la escritura de un número entero en el valor numérico correspondiente.

### 16.7.2. Ejemplos de uso de funciones avanzadas con cadenas alfanuméricas

En el ejemplo 16.4 se utilizan funciones avanzadas de la librería estándar. En primer lugar se puede ver un caso muy peculiar de inicialización de cadena alfanumérica, C permite poner dos cadenas seguidas sin problemas y esto se puede aprovechar para inicializar cadena largas utilizando varias líneas de código. Obsérvese el uso de **sprintf** para inicializar las cadenas de salida, es una alternativa a **strcpy**. También resulta conveniente fijarse en cómo **strtok** descompone la cadena aprovechando los correspondientes separadores (caracteres coma), especialmente el significado que tiene volver a llamar a **strtok** con el segundo argumento a **NULL**, y cómo se aprovecha el resultado para finalizar el bucle. Finalmente dentro del bucle se puede ver la comprobación del resultado de la función **strcmp** para establecer la separación de palabras. Se sugiere cambiar el valor del límite para estudiar el comportamiento del programa.

## Ejemplo 16.4: Separación de palabras por orden alfabético

```

1
2 #include <stdio.h>
3 #include <string.h>
4
5 #define DIM 512
6
7 int main ()
8 {
9     /* un texto con palabras separadas por comas */
10    char texto[] = "mesa,silla,cama,sofa,aparador,"
11        "libreria,comoda,armario";
12    char salida1[DIM], salida2[DIM];
13    char limite[] = "libreria"; char *aux = NULL;
14    int cmp = 0;
15
16    /* Se inicializa las cadenas */
17    sprintf(salida1, "Mayores que \"%s\":\n\t", limite);
18    sprintf(salida2, "Menores que \"%s\":\n\t", limite);
19
20    aux = strtok(texto, ",");
21    while ( aux != NULL ) {
22        cmp = strcmp(aux, limite);
23        if ( cmp > 0 ) {
24            strcat(salida1, " ");
25            strcat(salida1, aux);
26        }
27        else if ( cmp < 0 ) {
28            strcat(salida2, " ");
29            strcat(salida2, aux);
30        }
31        else {
32            printf("%s coincide con el limite\n\n", aux);
33        }
34        aux = strtok(NULL, ",");
35    }/*while*/
36
37    /* Se muestra el resultado */
38    printf("%s\n\n", salida1);
39    printf("%s\n", salida2);
40
41    return 0;
42 }
```

Salida del programa:

libreria coincide con el limite

Mayores que "libreria":  
mesa silla sofa

Menores que "libreria":  
cama aparador comoda armario

El siguiente programa, el ejemplo 16.5 , no resulta trivial dado que utiliza una función de búsqueda de caracteres en una cadena y luego se utiliza la cualidad de puntero del resultado. En ese sentido, este ejemplo resulta más sencillo si se conoce y entiende con detalle el comportamiento de los punteros. Se recomienda examinarlo con mayor atención después de leer el capítulo de punteros.

En todo caso el programa realiza la descomposición de una cadena de forma similar al ejemplo anterior, esta vez se utiliza el punto y coma como separador, y luego realiza la sustitución de comas que se puedan encontrar por puntos para poder realizar la correspondiente conversión a un valor numérico. La función `strpbrk` localiza la coma en cada uno de los números, se comprueba si la ha encontrado (se devuelve algo distinto de `NULL`) y luego se sustituye. Para

sustituir la coma se usa el operador indirección sobre el puntero, naturalmente, el resultado es la variable donde se guarda la coma y se le asigna como nuevo valor un punto. A continuación ya se puede llamar a `atof` para realizar la conversión. Nótese que la variable donde se guardaba la coma está en la cadena apuntada por `token` y por eso su valor resulta modificado.

Ejemplo 16.5: Reemplazar comas de números en un texto para obtener double

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main ()
7 {
8     /* un texto con numeros con , , */
9     char texto[] = "455,558;16'89;9,684;4'012e-2;";
10    "199,775;98'09;7,779;9'908e-2";
11    char *token = NULL; char *aux = NULL;
12    double num = 0; int i = 0;
13
14    token = strtok(texto,";");
15    while ( token != NULL ) {
16        /* Se localiza ' o , en el texto */
17        aux = strpbrk(token,",");
18        /* Si se encuentra se sustituye por . */
19        if ( aux != NULL ) {
20            *aux = '.';
21        }
22        /* Se convierte a un numero */
23        num = atof(token);
24        printf("%5.7f ", num);
25        ++i;
26        if ( i > 3 ) {
27            printf("\n");
28            i = 0;
29        }/*if*/
30        token = strtok(NULL,";");
31    }/*while*/
32
33    return 0;
34 }
```

Salida del programa:

```
455.5580000 16.8900000 9.6840000 0.0401200
199.7750000 98.0900000 7.7790000 0.0990800
```



## Capítulo 17

# Elementos Avanzados de programación

### 17.1. Punteros y modelo de memoria

El concepto de puntero es la traducción a un lenguaje de alto nivel de un concepto que se encuentra a bajo nivel en la forma de trabajar conjuntamente del microprocesador y la memoria del ordenador. Así pues, antes de pasar al puntero se describirá de manera somera cómo funciona la transferencia de información entre el microprocesador y la memoria.

Las memorias de tipo RAM (*Random Access Memory*) se caracterizan porque el acceso (lectura) y modificación (escritura) de datos en memoria se realiza a través de un número que identifica cada una de las zonas de memoria accesibles de forma individual. Habitualmente estas zonas se conocen como “palabra” y suelen contener un byte de información. Este número se comporta de forma análoga a los índices de las componentes de un vector en álgebra. Cada índice, 1, 2, 3, ... se corresponde con un valor entre las componentes del vector. En informática estos números o índices reciben el nombre específico de *direcciones de memoria*.

Así pues el proceso de lectura de memoria quedaría como sigue:

1. El microprocesador pone en el bus del ordenador el número (índice, dirección de memoria) con el identificador de memoria que quiere consultar. Es decir, fija los valores de potencial (0 y 1 lógicos, usualmente 0 y 0.3 Voltios) con la representación binaria del número indicado.
2. La memoria recibe el número a través de sus pines, localiza y obtiene el dato requerido.
3. La memoria fija en el bus el contenido de los datos que contiene. Es decir, escribe en el bus la representación binaria (a través de los valores de potencial) del dato requerido.

Mientras que el proceso de escritura quedaría como sigue:

1. El microprocesador pone en el bus el número con el identificar de la zona a modificar (escribir) y luego el dato a escribir.
2. La memoria recibe ambos, localiza la zona identificada por el número y escribe en ella el dato indicado por el microprocesador.

Como se ha dicho, el número que estamos manejando y que identifica una zona en memoria se llama dirección de memoria. El nombre parece obvio si se establece la comparación típica con un sistema de buzones o de apartados de correos, cada uno de los buzones es una zona de memoria donde se puede almacenar o recuperar información (cartas o paquetes) y el número de apartado de correos es precisamente la dirección de memoria.

El siguiente paso para entender los punteros es ir más allá de la noción de dirección de memoria. Al fin y al cabo, una dirección de memoria es un número (es decir, un dato) que se puede guardar en memoria, pues bien, cuando una dirección de memoria se encuentra guardada en una variable en memoria se tiene un *puntero*.

## 17.2. Referencia: la semántica del concepto.

Una definición alternativa para un puntero es:

un puntero es una referencia

Al entender puntero como una referencia podemos estudiar en que consiste una referencia para quizás entender mejor el concepto de puntero.

En este sentido es importante matizar y profundizar en el significado del concepto referencia. Una referencia, según el diccionario de la RAE (<http://buscon.rae.es/draeI/>) es:

*5. f. En un escrito, indicación del lugar de él mismo o de otro al que se remite al lector.*

En un sentido más amplio en el concepto de referencia hay dos partes:

1. Una referencia es la información para encontrar algo o alguien. En este sentido se puede pensar en un número de teléfono como una referencia a la persona que tiene ese número, o una dirección postal como una referencia a una vivienda u oficina.
2. Aquello a que hace referencia. Es decir, lo que se busca y encuentra mediante la información que indica la referencia. La persona que contesta a un cierto número de teléfono es “aquello a que hace referencia”, la persona está referida<sup>1</sup> por ese número. La casa que se encuentra en una dirección postal está referida por esa dirección postal.

Cuando existe una relación de referencia entre dos cosas, ambas, la referencia y la cosa referida son importantes. Pero no hay que olvidar que son distintas: una persona no es su número de móvil, ni una casa su dirección postal. Entre ellos existen relaciones de referencia, pero no son lo mismo.

Ahora bien, en informática se emplea la palabra referencia para denominar a cualquier mecanismo para localizar o encontrar un dato. En este sentido, en el lenguaje C, un puntero es una referencia, es decir, sirve para localizar o encontrar un dato en memoria. En lenguajes distintos de C se pueden encontrar otras formas de realizar referencias, no obstante, todas ellas suelen estar basadas en punteros aunque no de forma explícita.

En el sentido de referencia como forma de encontrar alguna otra cosa, se podría decir que una dirección postal es la referencia de una casa, que el nombre de un aula (por ejemplo, Aula E1) es una referencia del aula (nos permite localizarla en un plano), etc. Es conveniente tener presente éstos u otros ejemplos para comprender y profundizar en el concepto de referencia.

Naturalmente no se debe confundir la referencia con aquello a lo que hace referencia. En los ejemplos anteriores se observa claramente que una cosa es la casa o el aula y otra muy distinta una dirección postal o el nombre del aula. De la misma manera, un puntero es una referencia de una variable, es decir, un puntero guarda la dirección de memoria donde se encuentra esa variable, pero la variable y el puntero son dos cosas distintas.

### 17.2.1. El tipo de los datos referidos

Dado que los punteros son referencias, es importante saber a qué van a apuntar, es decir, a qué hacen referencia. En este sentido, la declaración de un puntero se realiza indicando que se declara un puntero y especificando el tipo de dato al que apunta. Esto es necesario porque C es un lenguaje de programación fuertemente tipado, es decir, todas las expresiones tienen un tipo conocido en tiempo de compilación.

Así pues, al utilizar punteros tenemos que manejar dos tipos de datos, el primero de estos tipos es, naturalmente, el tipo puntero, es decir, el tipo que sirve para almacenar direcciones de memoria. El segundo es libre, puede ser un número entero, una letra o cualquier tipo ya definido, pero lo importante es que este tipo también debe estar determinado.

Cuando un puntero es una referencia de datos de un tipo concreto se dice que apunta a ese tipo. Así pues, una variable “puntero que apunta a entero” es de tipo puntero y las variables a las que puede hacer referencia son de tipo entero.

No se debe confundir nunca el tipo del objeto referido con el tipo puntero. Como ya se ha dicho, referencia y objeto referido son cosas distintas. Esta dualidad de tipo provoca que todos los punteros tengan una relación muy cercana entre sí. Por ejemplo, comparten codificación y tamaño. Sin embargo existen muchos punteros diferentes y su comportamiento en algunas operaciones es muy distinto.

Como ya se ha indicado el principal uso de los punteros es servir de referencia a otros datos. Cuando se maneja un puntero, en su condición de referencia, se tienen dos valores: la referencia (el puntero) y lo referido (otra variable) y siempre serán necesarios dos tipos para determinar el tipo de los resultados de las operaciones con punteros.

<sup>1</sup>La acción de referencia es referir. Véase RAE.

## 17.3. Otras soluciones para implementar matrices dinámicas

### 17.3.1. Matriz dinámica de filas de tamaño conocido y constante

Si se puede imponer un tamaño de fila conocido y constante, es decir, un número de columnas constante, se pueden construir matrices con un número de filas variables con cierta facilidad. Éste es un caso que se presenta con cierta frecuencia.

En el ejemplo 17.1 se muestra un programa que trabaja con una matriz cuyo número de filas es variable y que representa un array de puntos en el espacio de tres dimensiones de tamaño variable.

Ejemplo 17.1: Matriz dinámica con número de columnas constante

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void desplazar_x(double point_array[][3], unsigned tam, double desp)
5 {
6     unsigned i;
7     for ( i = 0; i < tam; ++i ) {
8         point_array[i][0] += desp;
9     }
10 }
11
12 void mostrar(double (*point_array)[3], unsigned tam)
13 {
14     unsigned i;
15     for ( i = 0; i < tam; ++i ) {
16         printf("[%3.3f,%t%3.3f,%t%3.3f]\n",
17                point_array[i][0], point_array[i][1], point_array[i][2] );
18     }
19 }
20
21 int main()
22 {
23     unsigned i, tam = 4;
24     double (*point_array)[3];
25
26     point_array = ( double(*)[3] ) malloc( tam*sizeof(double[3]) );
27
28     for ( i = 0; i < tam; ++i ) {
29         point_array[i][0] = i;
30         point_array[i][1] = 10*i;
31         point_array[i][2] = 100*i;
32     }
33
34     desplazar_x(point_array, tam, 3.3);
35     mostrar(point_array, tam);
36     free(point_array);
37
38     return 0;
39 }
```

---

Salida por pantalla en la ejecución:

```
[3.300,0.000,0.000]
[4.300,10.000,100.000]
[5.300,20.000,200.000]
[6.300,30.000,300.000]
```

Es conveniente revisar este ejemplo con especial atención porque la declaración de los tipos no es trivial. En concreto, resulta interesante examinar:

1. La declaración de la variable puntero cuya finalidad es guardar el vector dinámico cuyos elementos son *arrays*. El paréntesis que engloba el asterisco y el nombre de la variable marca una gran diferencia: indica que la variable es un *único* puntero, pero luego el corchete posterior indica que el tipo al que apunta es un *array* de 3 elementos.

2. Al hacer la conversión (*cast*) entre las variables puntero de nuevo hay un paréntesis. El que resulta de eliminar el identificador de la variable en la declaración anterior. Ese es el tipo del puntero tal cuál. Es de destacar que el operador suma de punteros aplicado a una variable de este tipo desplaza el puntero en memoria un espacio equivalente a 3 números *double*. Es decir, *point\_array* + 1 es el puntero al segundo de los puntos.
3. En el ejemplo se muestran además dos alternativas para pasar parámetros de este tipo de matrices dinámicas. Ambas también son válidas para el paso de matrices estáticas declaradas como *double matriz[N][3]*. De hecho, las variables de tipo matriz estática son convertibles a punteros de estos tipos y el operador indexación aplicado dos veces tiene exactamente el mismo efecto que en estas variables dinámicas, tanto en la primera vez como en la segunda.

### 17.3.2. Matriz dinámica a partir de un vector unidimensional

También se puede utilizar un vector unidimensional dinámico para guardar una matriz. En este caso los valores de la matriz se guardarán por filas, todas seguidas en el vector. Esta representación tiene un inconveniente: no se puede usar el operador indexación dos veces para obtener un elemento de la matriz. Es decir dada una matriz, *mat*, no se puede usar *mat[i][j]* sino *mat[i\*columnas+j]*.

En el ejemplo 17.2, se muestra un programa que trabaja con una matriz cuyo número de filas y columnas es variable, pero que se guarda como un vector.

Ejemplo 17.2: Matriz dinámica usando un vector dinámico

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void matriz_mas_numero(int *matriz, unsigned M, unsigned N, int x)
6 {
7     /* C99: typedef int (*Matriz)[N]; Matriz mat = matriz; */
8     unsigned i, j;
9     for ( i = 0; i < M; ++i ) {
10         for ( j = 0; j < N; ++j ) {
11             matriz[i*N+j] += x;
12             /* C99: mat[i][j] += x; */
13         }
14     }
15 }
16
17 int main()
18 {
19     unsigned M = 3, N = 4, i, j;
20     int *matriz = (int*) malloc(M*N*sizeof(int));
21
22     /* rellena la matriz */
23     for ( i = 0; i < M; ++i ) {
24         for ( j = 0; j < N; ++j ) {
25             matriz[i*N+j] = 3*i+j;
26         }
27     }
28
29     matriz_mas_numero(matriz, M, N, 4);
30
31     /* muestra la matriz */
32     for ( i = 0; i < M; ++i ) {
33         for ( j = 0; j < N; ++j ) {
34             printf("%4i ", matriz[i*N+j]);
35         }
36         printf("\n");
37     }
38
39     free(matriz);
40     return 0;
41 }
```

Salida por pantalla en la ejecución:

4	5	6	7
7	8	9	10
10	11	12	13

En el ejemplo se puede observar algunas particularidades de esta implementación:

1. No hay en la definición de tipos que nos indique que las variables `matriz` son, de hecho, matrices.
2. No se emplean los dos operadores indexación, sino uno solo, pero sí se emplean dos índices para obtener los valores de la matriz.
3. En la declaración de parámetros, lo único que indica que trabajamos con una matriz es la declaración de los dos tamaños de la matriz.
4. En C99 se podría retocar este programa según los comentarios que aparecen en el ejemplo.

## 17.4. Recursividad

La recursividad, recurrencia o recursión es la propiedad mediante la cual una rutina puede llamarse a sí misma para realizar una tarea, es decir, se define en función de sí misma. Aunque puede emplearse en C, no todos los lenguajes de programación admiten el uso de la recursividad.

En el análisis de un problema que se quiere resolver utilizando la recursividad:

1. La solución del problema se reduce a uno esencialmente igual pero algo menos complejo que se supone resuelto, mediante la llamada recursiva, y a partir del cuál se resuelve el problema original.
2. Se tienen que determinar las condiciones de *corte*, *salida* o *terminales* que sirven para parar la recursividad de forma que no se realicen nuevas llamadas recursivas y evitar una recursividad infinita.
3. Debe existir un caso *trivial*, *base* o *no recursivo* que consiste en un problema muy reducido que hace innecesaria la recursividad porque la solución es directa y evidente. Normalmente esta solución o caso trivial se produce cuando se cumple la condición terminal.

La recursividad es una herramienta muy potente y útil en la resolución de muchos problemas, aunque no se debe abusar de la recursividad porque pueden encontrarse problemas de uso de memoria y rendimiento.

Por ejemplo, puede usarse en la definición matemática del factorial de un número:

**Sin recursividad**  $n!=1$  si  $n=0$ ;  $n!=n*(n-1)*(n-2)*\dots*1$  si  $n>0$

**Recursivamente**  $n!=1$  si  $n=0$ ;  $n!=n*(n-1)!$  si  $n>0$

La solución recursiva del factorial se muestra a continuación en el ejemplo 17.3.

Ejemplo 17.3: Calculo recursivo del factorial

---

```
1
2 #include <stdio.h>
3
4 long factorial(long n)
5 {
6     /* condicion terminal o de corte */
7     if ( n > 1 ) {
8         /* Llamada recursiva */
9         return n*factorial(n-1);
10    }/*if*/
11    else {
12        /* Caso trivial */
13        return 1;
14    }/*else*/
15 }/*factorial*/
16
17 int main()
18 {
19     long n = 7, fact;
20
21     fact = factorial(n);
22     printf("El factorial de %ld es %ld\n", n, fact);
23
24     return 0;
25 }/*main*/
```

---

Salida del programa:

El factorial de 7 es 5040

Para analizar como se comporta la recursividad se pueden intercalar distintas sentencias con `printf` para comprobar como se produce la ejecución de una función recursiva. En el ejemplo 17.4 que se muestra a continuación se han añadido estas sentencias.

## Ejemplo 17.4: Calculo recursivo detallado del factorial

---

```

1
2 #include <stdio.h>
3
4 long factorial(long n)
5 {
6     printf("Entro a resolver el factorial de %ld\n", n);
7     if ( n > 1 ) {
8         long fact, fact_1;
9         printf("Me pongo a resolver el factorial de n-1\n");
10        fact_1 = factorial(n-1);
11        printf("El factorial de n-1: %ld es %ld\n", n-1, fact_1);
12        fact = n*fact_1;
13        printf("El factorial de n: %ld es %ld\n", n, fact);
14        return fact;
15    }/*if*/
16    else {
17        printf("Caso trivial el factorial de 1 es 1\n");
18        return 1;
19    }/*else*/
20 }/*factorial*/
21
22 int main()
23 {
24     long n = 4, fact;
25
26     fact = factorial(n);
27     printf("El factorial de %ld es %ld\n", n, fact);
28
29     return 0;
30 }/*main*/

```

---

Salida del programa:

```

Entro a resolver el factorial de 4
Me pongo a resolver el factorial de n-1
Entro a resolver el factorial de 3
Me pongo a resolver el factorial de n-1
Entro a resolver el factorial de 2
Me pongo a resolver el factorial de n-1
Entro a resolver el factorial de 1
Caso trivial el factorial de 1 es 1
El factorial de n-1: 1 es 1
El factorial de n: 2 es 2
El factorial de n-1: 2 es 2
El factorial de n: 3 es 6
El factorial de n-1: 3 es 6
El factorial de n: 4 es 24
El factorial de 4 es 24

```

## 17.5. Campos de Bits

Una extensión al tipo `struct` cuyo uso se encuentra en ocasiones en la programación de algunos dispositivos electrónicos y de sistemas de comunicaciones son los campos de bits o *Bit Fields*.

Básicamente, este tipo se usan para representar agrupaciones de datos de tipo entero que no ocupan el tamaño habitual de los datos predefinidos para los tipos enteros sino un tamaño fijo e inferior al usual. Esta circunstancia se produce con cierta frecuencia en la implementación de sistemas de bajo nivel donde se busca un uso de la memoria muy ajustado. Por ejemplo, si tenemos un sistema de comunicación que necesita enviar telegramas con una cierta cabecera que incluye el tipo del telegrama, pero ese tipo de telegrama solo tiene 32 alternativas o códigos, se puede usar un entero que solo ocupe 5 bits en vez de los 8 mínimos (usando `char`)<sup>2</sup>. Así mismo se ha codificado otros dos enteros reducidos para un destinatario y un campo para indicar la longitud del mensaje.

Ejemplo 17.5: Estructura de campos de bits

---

```

1 #include <stdio.h>
2
3 struct Telegrama
4 {
5     unsigned int codigo : 5;
6     unsigned int destino : 5;
7     unsigned int longitud : 15;
8     char *contenido;
9 };
10
11 void enviar(struct Telegrama *t)
12 {
13     unsigned i;
14
15     printf("codigo: %d, destino: %d, longitud: %d \n",
16            t->codigo, t->destino, t->longitud);
17
18     for ( i = 0; i < t->longitud; i++ )
19     {
20         printf("%c ", t->contenido[i]);
21     }/* for i */
22
23     printf("\n");
24 }/*enviar*/
25
26 int main()
27 {
28     struct Telegrama tel;
29
30     printf("sizeof(Telegrama): %d \n", sizeof(struct Telegrama));
31     tel.codigo = 3;
32     tel.destino = 9;
33     tel.longitud = 4;
34     tel.contenido = "hola";
35     enviar(&tel);
36     return 0;
37 }/*main*/

```

---

Salida del programa:

```

sizeof(Telegrama): 8
codigo: 3, destino: 9, longitud: 4
h o l a

```

En el ejemplo 17.5 se ilustra el uso de este tipo, aunque en este caso no se incluye una implementación que transmita el telegrama sino simplemente muestra su contenido por pantalla. En este ejemplo es conveniente resaltar:

<sup>2</sup>Nótese que el tamaño de palabra, es decir, el tamaño mínimo de dato que el sistema informático pueda gestionar influye en el tamaño mínimo real que ocupa un dato en memoria.

1. El tamaño del tipo es de 8 bytes <sup>3</sup>. La forma de distribución en memoria de los campos de las estructura o de los campos de bits no es trivial ni universal. Diferentes plataformas, compiladores o sistemas operativos pueden tener distintos modos de distribución en memoria. En general se tiende a que el acceso a los campos sea eficiente por encima del uso de memoria. Esto normalmente se traduce en colocar relleno o *padding* en la estructura para que el inicio de los campos ocupen determinadas direcciones de memoria.
2. Aunque la representación de los campos en memoria ocupe menos bits que de forma habitual, en el programa se trabaja con ellos como si fuesen un entero normal y corriente.
3. Es conveniente poner siempre si el entero es con o sin signo y así se ha hecho.
4. El uso de campos de bits es equivalente a guardar varios enteros en una variable entera mediante operadores de bits (*bitwise*) lógicos y de desplazamiento.

## 17.6. Tipos unión

Los tipos **union** se usan para proporcionar flexibilidad al almacenamiento en memoria de una misma información que puede aparecer de diferentes formas. Tienen cierta relación con las estructuras en el sentido de que hay que declararlos específicamente para cada caso y que, en cierta manera, también agrupan información.

Cuando varios campos se agrupan en una unión el objetivo es usar la misma zona de memoria para guardar alternativamente alguno de los campos de la unión. Esto tiene como consecuencia que el tamaño de la unión en memoria es el tamaño del campo más grande y que los campos de una unión no se pueden utilizar simultáneamente (como en las estructuras) sino exclusivamente, es decir, si uso el primer campo no debo usar el segundo y viceversa.

Aunque en ocasiones su uso tiene ventajas importantes no se recomienda abusar de las uniones porque no resultan especialmente eficientes en el uso de memoria y hacen más difíciles de leer e interpretar los programas donde aparecen.

Ejemplo 17.6: Declaración y uso de una unión

---

```

1 #include <stdio.h>
2 #include <string.h>
3
4 union DireccionUnion
5 {
6     unsigned char ip[4];
7     char nombre[255];
8 };
9
10 enum FormaIP { Numero, Nombre };
11
12 struct Direccion
13 {
14     enum FormaIP opcion;
15     union DireccionUnion dato;
16 };
17
18 void mostrar(struct Direccion *d)
19 {
20     switch ( d->opcion )
21     {
22     case Numero:
23         printf("%d.%d.%d.%d \n", d->dato.ip[0], d->dato.ip[1],
24                d->dato.ip[2], d->dato.ip[3]);
25         break;
26     case Nombre:
27         printf("%s \n", d->dato.nombre);
28         break;
29     } /*switch*/
30 } /*mostrar*/
31
32

```

<sup>3</sup>Comprobado en un ubuntu de 32 bits, en otros sistemas puede ser diferente, especialmente en sistemas de 64 bits.

```

33 int main()
34 {
35     struct Direccion dir;
36     printf("sizeof(union DireccionUnion): %d \n",
37             sizeof(union DireccionUnion));
38
39     dir.opcion = Numero;
40     dir.dato.ip[0] = 138;
41     dir.dato.ip[1] = 100;
42     dir.dato.ip[2] = 255;
43     dir.dato.ip[3] = 255;
44     mostrar(&dir);
45
46     dir.opcion = Nombre;
47     strcpy (dir.dato.nombre, "www.etsii.upm.es");
48     mostrar(&dir);
49
50     return 0;
51 }

```

---

Salida del programa:

```

sizeof(union DireccionUnion): 255
138.100.255.255
www.etsii.upm.es

```

En el ejemplo 17.6 se ha usado un tipo **union** para usar dos maneras de guardar la dirección de un ordenador en Internet, a través de su número IP o de su nombre. En este ejemplo es conveniente destacar los siguientes aspectos del uso del tipo **union**:

1. Es prácticamente obligatorio el uso de algún dato extra donde se guarde cual es el campo de la unión que se usa en cada momento. En este ejemplo se ha definido una estructura para guardar este dato, **opcion**, junto con la propia unión.
2. La unión ocupa 255 bytes independientemente de que la dirección se exprese como números (en cuyo caso solo hacen falta 4 bytes) o como nombre (donde si se necesitan los 255 bytes).
3. Se necesita una sentencia condicional (se ha usado un **switch**) para seleccionar cómo se debe trabajar con la unión cuando se usa cada campo de la misma.

## 17.7. Definición e implementación de una pila

Una pila es una estructura dinámica de datos que se caracteriza porque tiene tres operaciones:

1. insertar un nuevo elemento al principio de la estructura,
2. eliminar un elemento del principio de la estructura y
3. conocer si la estructura tiene algún elemento.

Un nombre alternativo a cola es FILO, del inglés *First In, Last Out* que describe el orden de entrada y salida a la estructura. La implementación utiliza como elemento base un **struct** con un puntero al siguiente elemento de la pila. También es conveniente guardar el principio de la pila en una estructura aparte para simplificar la declaración de las operaciones necesarias. Su comportamiento es análogo a una pila de platos, pero respetando que no se ponen o quitan platos del medio de la pila, solo se pone o quita el plato de arriba.

El ejemplo 17.7 muestra el código fuente de un programa en C que implementa las tres funciones que se pueden aplicar a una pila. Estas funciones se han llamado: **is\_empty**, **pop** y **push** porque reciben nombres similares a estos en algunas librerías de estructuras de datos. Aunque no es estrictamente necesario se ha definido un tipo de dato **struct** para la propia pila: FILO. Esto se hace para facilitar la lectura del código fuente, ya que el tipo FILO aparece como parámetro de las funciones que se aplican a la misma. Es importante señalar que la inicialización del puntero a NULL (comentada con la letra A) es absolutamente imprescindible para que la pila funcione correctamente, esta inicialización

permite saber que la pila está vacía. Otro punto a considerar es el comportamiento de la función `pop` cuando la lista está vacía, en esta implementación se ha optado por devolver `NaN` (*Not a Number*).

Por otro lado se recomienda que el lector se familiarice con las manipulaciones que se hacen con los punteros y las variables dinámicas manejadas en el programa. Una buena manera de hacer esto es dibujar un diagrama donde se dibuje paso a paso los cambios en una pila existente cuando se ejecuta cada una de las líneas de código fuente de las funciones. Para facilitar esta tarea se ha escrito un comentario con un número que se puede usar para marcar los cambios de cada paso en el diagrama.

También se ha hecho un pequeño avance de lo que sería la definición de la pila en un archivo de cabeceras separado. Para ello se han introducido delante del programa principal las declaraciones que el programa principal necesita y después del programa los detalles de implementación de la pila, incluida la definición de las estructuras `FILO` y `Elemento`. Como se puede ver, dado que el programa solo usa punteros a la estructura, solo es necesario saber que la estructura existe (la primera declaración). Como el programa solo llama a las funciones, solo le hace falta saber el prototipo, declaración o cabecera de las mismas, es decir, el nombre de la función con el retorno y los parámetros, pero sin el código de implementación (que va entre llaves). Para la declaración de la estructura y las funciones se omiten las llaves y se cierra la declaración con punto y coma. Para poner el código fuente de la pila en otros archivos distintos del archivo donde se escribe el programa principal bastaría con colocar el código antes de `main` en un archivo de cabeceras (extensión .h, líneas de la 5 a la 12) y el código fuente después de `main` en un archivo de implementación (extensión .c, líneas de la 40 hasta el final) y luego compilarlo todo.

Finalmente, el diseño de este ejemplo está muy próximo a lo que se conoce como programación orientada a objetos. Se ha seguido una forma de programar que se conoce como *programación a objetos en C* y que se caracteriza fundamentalmente por usar estructuras y funciones asociadas cuyo primer parámetro siempre es un puntero a la estructura. Si se utiliza un archivo de cabeceras aparte, se puede comprobar que sería imposible usar una pila sin que se haya inicializado o usar sus campos directamente. Solo las funciones son capaces de manejar la estructura y esto reduce las posibilidades de que el programador pueda cometer errores (en programación orientada a objetos este concepto se denomina encapsulación).

Ejemplo 17.7: Implementación de una pila de números en coma flotante

---

```

1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 struct FILO;
6
7 struct FILO* create_stack();
8 void destroy_stack(struct FILO*);
9
10 int is_empty(const struct FILO *f);
11 double pop(struct FILO *f);
12 void push(struct FILO *f, double x);
13
14 int main()
15 {
16     int i;
17     struct FILO *pila;
18
19     pila = create_stack();
20
21     push(pila, 4.2);
22     push(pila, 3.7);
23     push(pila, 8.2);
24
25     printf("El primero es %f\n", pop(pila));
26
27     push(pila, 7.0);
28
29     for ( i = 0; i < 3; ++i )
30     {
31         printf("%f\n", pop(pila));
32     }
33
34     printf("vacia: %i\n", is_empty(pila));

```

```

35
36     destroy_stack(pila);
37
38     return 0;
39 }
40
41
42 struct Elemento
43 {
44     double dato;
45     struct Elemento *sig;
46 };
47
48 struct FILO
49 {
50     struct Elemento *prim;
51 };
52
53 struct FILO* create_stack()
54 {
55     struct FILO* p = malloc(sizeof(struct FILO));
56     p->prim = NULL; /* 1 */
57     return p;
58 }
59
60 int is_empty(const struct FILO *p)
61 {
62     return p->prim == NULL;
63 }
64
65 double pop(struct FILO *p)
66 {
67     struct Elemento *aux;
68     double result;
69
70     aux = p->prim; /* 1 */
71     result = aux->dato; /* 2 */
72     p->prim = aux->sig; /* 3 */
73     free ( aux ); /* 4 */
74     return result; /* 5 */
75 }
76
77 void push(struct FILO *p, double x)
78 {
79     struct Elemento *aux;
80
81     aux = malloc( sizeof(struct Elemento) ); /*1*/
82     aux->dato = x; /*2*/
83     aux->sig = p->prim; /*3*/
84     p->prim = aux; /*4*/
85 }
86
87 void destroy_stack(struct FILO* f)
88 {
89     while ( ! is_empty( f ) )
90     {
91         pop( f );
92     }
93     free ( f );
94 }
```

Salida del programa:

El primero es 8.200000

```
7.000000
3.700000
4.200000
vacia: 1
```



# Bibliografía

- [1] J.L. Antonakos and K.C. Mansfield. *Programación estructurada en C.* Prentice-Hall, 1997.
- [2] B.W. Kernighan and D.M. Ritchie. *El lenguaje de programación C.* Prentice-Hall, 1991.
- [3] Diego Rodríguez-Losada and Otros. *Introducción a la programación en C.* Servicio Publicaciones ETSII.
- [4] Herbert Schildt. *Aplique Turbo C.* McGraw-Hill.
- [5] Herbert Schildt. *C Manual de Referencia.* McGraw-Hill.
- [6] B. Stroustrup. *El lenguaje de programación C.* Addison-Wesley, 1993.



# Listado de programas y ejemplos

2.1. Programa vacío . . . . .	28
2.2. Programa vacío sin avisos . . . . .	29
2.3. Muestra un mensaje . . . . .	29
2.4. Muestra datos numéricos . . . . .	30
2.5. Buen estilo . . . . .	31
2.6. Estilo confuso . . . . .	31
3.1. Estructura de un programa básico . . . . .	39
3.2. Precaución en el uso de macros (define) . . . . .	40
4.1. Tamaño de tipos de datos básicos . . . . .	48
4.2. Declaraciones de variables de tipo básico . . . . .	54
4.3. Declaraciones de constantes de tipo básico . . . . .	54
5.1. Operadores aritméticos enteros y en coma flotante . . . . .	63
5.2. Operadores de comparación y lógicos . . . . .	64
5.3. Operadores de bits o bitwise . . . . .	64
5.4. Horas equivalentes en semanas, días y horas . . . . .	65
5.5. Expresión numérica compleja . . . . .	66
6.1. Sentencia if muy sencilla . . . . .	70
6.2. Sentencia if con una rama . . . . .	71
6.3. Sentencia if con dos ramas . . . . .	71
6.4. Sentencias if una detrás de otra . . . . .	71
6.5. Sentencias if anidadas . . . . .	72
6.6. Sentencias if anidadas en rama else . . . . .	73
6.7. Sentencia switch . . . . .	74
6.8. Cálculo del mayor de tres valores numéricos reales . . . . .	76
6.9. Cálculo de índice de masa corporal . . . . .	77
6.10. Cálculo de las soluciones reales de la ecuación de segundo grado . . . . .	78
6.11. Programa que simula una calculadora . . . . .	78
7.1. Sentencia while . . . . .	82
7.2. Sentencia do-while . . . . .	83
7.3. Sentencia for . . . . .	84
7.4. Lista de temperaturas . . . . .	85
7.5. Lista de temperaturas, estilo confuso . . . . .	86
7.6. Sentencias repetitivas combinadas . . . . .	87
7.7. Repetición con bucle while . . . . .	89
7.8. Repetición con bucle do-while . . . . .	89
7.9. Repetición con bucle for . . . . .	89
7.10. Cálculo del factorial de un número . . . . .	90
7.11. Bucle para calcular una potencia . . . . .	90
7.12. calcula años para duplicar inversión . . . . .	91
7.13. tablas de multiplicar . . . . .	91
7.14. primos entre el 1 y el 300 . . . . .	92
7.15. Juego de adivinar un número . . . . .	93
8.1. Uso de parámetros por valor . . . . .	100
8.2. Declaración y uso de la función esPositivo . . . . .	101
8.3. Declaración y uso de la función sumadigito . . . . .	101
8.4. Función cubo de un valor numérico . . . . .	102
8.5. Pasar de hexadecimal a decimal . . . . .	102

8.6. Funcion de dependiente de cuadrantes . . . . .	103
8.7. Tablas de multiplicar del 4 y del 7 . . . . .	104
9.1. Distribución de variables en memoria . . . . .	107
9.2. Declaraciones de punteros a tipo básico . . . . .	108
9.3. Declaración y conversión de punteros genéricos . . . . .	109
9.4. Operadores de punteros . . . . .	110
9.5. Puntero que apunta a variable estática . . . . .	112
9.6. Paso de parámetro por valor . . . . .	112
9.7. Paso de un puntero por valor . . . . .	113
9.8. Uso de parámetros por referencia . . . . .	114
9.9. Ejemplo de uso de punteros . . . . .	115
9.10. Ejemplo de uso de parametros por referencia . . . . .	116
9.11. Ejemplo de uso combinado de retorno de función y de parametro por referencia . . . . .	116
9.12. Funcion que incrementa el valor de un parametro . . . . .	117
9.13. Funcion que devuelve una direccion de memoria . . . . .	117
10.1. Declaración y uso de un vector de enteros . . . . .	127
10.2. Inicialización de un vector y multiplicación por 3 . . . . .	127
10.3. Inicialización parcial de valores de un vector y cuenta de ceros . . . . .	128
10.4. Comprobar si 2 vectores están ordenados . . . . .	128
10.5. Inicialización y uso básico de una matriz . . . . .	129
10.6. Producto de matriz por vector . . . . .	130
10.7. Ejemplo de uso de un vector de enteros . . . . .	131
10.8. Ejemplo de funcion con parametro de tipo vector . . . . .	131
10.9. Simulacion del lanzamiento de dos dados . . . . .	132
10.10Uso de una matriz 2 x 3 . . . . .	132
10.11Ejemplos de vector de enteros y vector de reales . . . . .	133
10.12Ejemplo de funcion con parametros de tipo vector . . . . .	134
11.1. Definición de cadena alfanumérica . . . . .	140
11.2. Manipulación de apellidos y nombre . . . . .	140
11.3. Ejemplo de uso de cadenas . . . . .	141
11.4. Ejemplo de uso de funciones de cadenas . . . . .	141
11.5. Funcion que copia cadena . . . . .	142
11.6. Ejemplos de lectura por teclado de cadenas . . . . .	142
11.7. Ejemplos de funcion con parametro de tipo cadena . . . . .	143
12.1. Declaración de una estructura . . . . .	146
12.2. Funciones y struct . . . . .	147
12.3. Estructura con datos auxiliares de un vector . . . . .	148
12.4. Ejemplo de uso de una estructura . . . . .	150
12.5. Estructuras y funciones . . . . .	150
12.6. Estructuras de puntos en el espacio 2D . . . . .	151
12.7. Funciones con punteros a estructuras . . . . .	152
12.8. Array de struct . . . . .	153
13.1. Uso de fopen . . . . .	159
13.2. Llamada a fopen con cadena alfanumérica . . . . .	159
13.3. Cierre de archivo con fclose . . . . .	160
13.4. Escribe un archivo con números aleatorios . . . . .	162
13.5. Lectura de archivo . . . . .	165
13.6. Lee un archivo escrito previamente . . . . .	166
13.7. Lee un archivo línea a línea . . . . .	167
13.8. Escribe datos de distintos tipos . . . . .	168
13.9. Lee datos de distintos tipos . . . . .	169
13.10Cuenta caracteres en un archivo de texto . . . . .	170
13.11Copia un archivo de texto en otro . . . . .	171
13.12Borra un archivo de disco . . . . .	172
13.13Renombra un archivo de disco . . . . .	172
14.1. Reserva de memoria con malloc . . . . .	175
14.2. Múltiples punteros y variables dinámicas . . . . .	177
14.3. Asignaciones entre punteros . . . . .	179

14.4. Creación y uso de un vector dinámico . . . . .	181
14.5. Matriz dinámica . . . . .	182
14.6. Variable dinámica de tipo estructura . . . . .	183
14.7. Implementación de una cola de números en coma flotante . . . . .	185
14.8. Implementación de una cola con recorrido . . . . .	188
14.9. Ejemplo de uso de una variable dinámica . . . . .	189
14.10. Ejemplo de uso de free . . . . .	190
14.11. Ejemplo de doble indirección . . . . .	190
14.12. Ejemplo de variables dinámicas de diferentes tipos . . . . .	191
14.13. Calcular la media de un vector dinámico . . . . .	192
14.14. Asignar valores a un vector dinámico de caracteres . . . . .	192
15.1. Estructura de un programa . . . . .	200
15.2. Extracto con comandos ifdef . . . . .	201
15.3. Resultado del comando ifdef . . . . .	201
15.4. makefile.mk . . . . .	202
15.5. makefile . . . . .	203
16.1. Sentencia break en un bucle for . . . . .	215
16.2. Sentencia continue en un bucle for . . . . .	216
16.3. Sentencia goto . . . . .	217
16.4. Separación de palabras por orden alfabético . . . . .	219
16.5. Reemplazar comas de números en un texto para obtener double . . . . .	221
17.1. Matriz dinámica con número de columnas constante . . . . .	225
17.2. Matriz dinámica usando un vector dinámico . . . . .	226
17.3. Calculo recursivo del factorial . . . . .	228
17.4. Calculo recursivo detallado del factorial . . . . .	229
17.5. Estructura de campos de bits . . . . .	230
17.6. Declaración y uso de una unión . . . . .	231
17.7. Implementación de una pila de números en coma flotante . . . . .	233