

PROGRAMACIÓN EN C

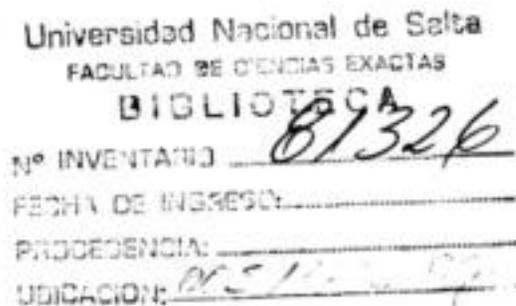
Metodología, algoritmos y estructura de datos



Luis Joyanes Aguilar

Ignacio Zahonero Martinez

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática/Escuela Universitaria de Informática
Universidad Pontificia de Salamanca. *Campus Madrid*



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

CONTENIDO

Prólogo	xv
---------------	----

PARTE I. METODOLOGÍA DE LA PROGRAMACIÓN

Capítulo 1. Introducción a la ciencia de la computación y a la programación	2
1.1. ¿Qué es una computadora?	4
1.2. Organización física de una computadora (hardware)	4
1.2.1. Dispositivos de Entrada/Salida (E/S)	5
1.2.2. La memoria central (interna)	6
1.2.3. La Unidad Central de Proceso (UCP)	9
1.2.4. El microprocesador	10
1.2.5. Memoria auxiliar (externa)	10
1.2.6. Proceso de ejecución de un programa	12
1.2.7. Comunicaciones: módems, redes, telefonía RDSI y ADSL	12
1.2.8. La computadora personal multimedia ideal para la programación	13
1.3. Concepto de algoritmo	15
1.3.1. Características de los algoritmos	16
1.4. El software (los programas)	17
1.5. Los lenguajes de programación	19
1.5.1. Instrucciones a la computadora	20
1.5.2. Lenguajes máquina	20
1.5.3. Lenguajes de bajo nivel	21
1.5.4. Lenguajes de alto nivel	22
1.5.5. Traductores de lenguaje	22
1.5.5.1. Intérpretes	23
1.5.5.2. Compiladores	23
1.5.6. La compilación y sus fases	23
1.6. El lenguaje C: historia y características	25
1.6.1. Ventajas de C	25
1.6.2. Características técnicas de C	26
1.6.3. Versiones actuales de C	26
1.7. Resumen	27
Capítulo 2. Fundamentos de programación	28
2.1. Fases en la resolución de problemas	30
2.1.1. Análisis del problema	31
2.1.2. Diseño del algoritmo	32
2.1.3. Herramientas de la programación	33
2.1.4. Codificación de un programa	36

2.1.5. Compilación y ejecución de un programa	37
2.1.6. Verificación y depuración de un programa	38
2.1.7. Documentación y mantenimiento	38
2.2. Programación modular	49
2.3. Programación estructurada	40
2.3.1. Recursos abstractos	40
2.3.2. Diseño descendente (<i>top-down</i>)	40
2.3.3. Estructuras de control	41
2.3.4. Teorema de la programación estructurada: estructuras básicas	42
2.4. Representación gráfica de los algoritmos	42
2.4.1. Diagramas de flujo	43
2.5. Diagramas de Nassi-Schneiderman (N-S)	52
2.6. El ciclo de vida del software	53
2.6.1. Análisis	54
2.6.2. Diseño	55
2.6.3. Implementación (codificación)	55
2.6.4. Pruebas e integración	56
2.6.5. Verificación	56
2.6.6. Mantenimiento	57
2.6.7. La <i>obsolescencia</i> : programas obsoletos	57
2.6.8. Iteración y evolución del software	57
2.7. Métodos formales de verificación de programas	58
2.7.1. Aserciones	58
2.7.2. Precondiciones y postcondiciones	59
2.7.3. Reglas para prueba de programas	60
2.7.4. Invariantes de bucles	60
2.7.5. Etapas a establecer la exactitud (corrección) de un programa	62
2.7.6. Programación segura contra fallos	63
2.8. Factores en la calidad del software	64
2.9. Resumen	65
2.10. Ejercicios	65
2.11. Ejercicios resueltos	66

PARTE II. FUNDAMENTOS DE PROGRAMACIÓN EN C

Capítulo 3. El lenguaje C: elementos básicos	72
3.1. Estructura general de un programa en C	74
3.1.1. Directivas del preprocesador	76
3.1.2. Declaraciones globales	77
3.1.3. Función <code>main()</code>	78
3.1.4. Funciones definidas por el usuario	78
3.1.5. Comentarios	80
3.2. Creación de un programa	82
3.3. El proceso de ejecución de un programa en C	83
3.4. Depuración de un programa en C	86
3.4.1. Errores de sintaxis	86
3.4.2. Errores lógicos	87
3.4.3. Errores de regresión	88
3.4.4. Mensajes de error	88
3.4.5. Errores en tiempo de ejecución	88
3.5. Pruebas	89
3.6. Los elementos de un programa en C	90
3.6.1. Tokens (elementos léxicos de los programas)	90
3.6.2. Identificadores	90
3.6.3. Palabras reservadas	91
3.6.4. Comentarios	91

3.6.5.	Signos de puntuación y separadores	92
3.6.6.	Archivos de cabecera	92
3.7.	Tipos de datos en C	92
3.7.1.	Enteros (<i>int</i>)	93
3.7.2.	Tipos de coma flotante (<i>float</i> / <i>double</i>)	94
3.7.3.	Caracteres (<i>char</i>)	95
3.8.	El tipo de dato LÓGICO	96
3.8.1.	Escritura de valores lógicos	97
3.9.	Constantes	97
3.9.1.	Constantes literales	98
3.9.2.	Constantes definidas (simbólicas)	101
3.9.3.	Constantes enumeradas	101
3.9.4.	Constantes declaradas <i>const</i> y <i>volatile</i>	101
3.10.	Variables	103
3.10.1.	Declaración	103
3.10.2.	Inicialización de variables	105
3.10.3.	Declaración o definición	105
3.11.	Duración de una variable	106
3.11.1.	Variables locales	106
3.11.2.	Variables globales	106
3.11.3.	Variables dinámicas	107
3.12.	Entradas y salidas	108
3.12.1.	Salida	108
3.12.2.	Entrada	111
3.12.3.	Salida de cadenas de caracteres	112
3.12.4.	Entrada de cadenas de caracteres	112
3.13.	Resumen	113
3.14.	Ejercicios	113
Capítulo 4.	Operadores y expresiones	114
4.1.	Operadores y expresiones	116
4.2.	Operador de asignación	116
4.3.	Operadores aritméticos	117
4.3.1.	Asociatividad	119
4.3.2.	Uso de paréntesis	120
4.4.	Operadores de incrementación y decrementación	120
4.5.	Operadores relacionales	123
4.6.	Operadores lógicos	125
4.6.1.	Evaluación en cortocircuito	127
4.6.2.	Asignaciones <i>booleanas</i> (lógicas)	128
4.7.	Operadores de manipulación de bits	129
4.7.1.	Operadores de asignación adicionales	130
4.7.2.	Operadores de desplazamiento de bits (<i>>></i> , <i><<</i>)	131
4.7.3.	Operadores de direcciones	131
4.8.	Operador condicional	132
4.9.	Operador coma	133
4.10.	Operadores especiales <i>()</i> , <i>[]</i>	133
4.10.1.	El operador <i>()</i>	133
4.10.2.	El operador <i>[]</i>	133
4.11.	El operador <i>sizeof</i>	134
4.12.	Conversiones de tipos	135
4.12.1.	Conversión implícita	135
4.12.2.	Reglas	135
4.12.3.	Conversión explícita	136
4.13.	Prioridad y asociatividad	136
4.14.	Resumen	137

4.15. Ejercicios	137
4.16. Problemas	139
Capítulo 5. Estructuras de selección: sentencias <code>if</code> y <code>switch</code>	142
5.1. Estructuras de control	144
5.2. La sentencia <code>if</code>	144
5.3. Sentencia <code>if</code> de dos alternativas: <code>if-else</code>	147
5.4. Sentencias <code>if-else</code> anidadas	150
5.4.1. Sangría en las sentencias <code>if</code> anidadas	151
5.4.2. Comparación de sentencias <code>if</code> anidadas y secuencias de sentencias <code>if</code>	152
5.5. Sentencia de control <code>switch</code>	154
5.5.1. Caso particular de <code>case</code>	159
5.5.2. Uso de sentencias <code>switch</code> en menús	159
5.6. Expresiones condicionales: el operador <code>?:</code>	159
5.7. Evaluación en cortocircuito de expresiones lógicas	161
5.8. Puesta a punto de programas	162
5.9. Errores frecuentes de programación	163
5.10. Resumen	164
5.11. Ejercicios	165
5.12. Problemas	167
Capítulo 6. Estructuras de control: bucles	168
6.1. La sentencia <code>while</code>	170
6.1.1. Operadores de incremento y decremento (<code>++</code> , <code>--</code>)	172
6.1.2. Terminaciones anormales de un ciclo	174
6.1.3. Diseño eficiente de bucles	174
6.1.4. Bucles <code>while</code> con cero iteraciones	174
6.1.5. Bucles controlados por centinelas	175
6.1.6. Bucles controlados por indicadores (banderas)	175
6.1.7. La sentencia <code>break</code> en los bucles	177
6.1.8. Bucles <code>while (true)</code>	178
6.2. Repetición: el bucle <code>for</code>	180
6.2.1. Diferentes usos de bucles <code>for</code>	184
6.3. Precauciones en el uso de <code>for</code>	185
6.3.1. Bucles infinitos	186
6.3.2. Los bucles <code>for</code> vacíos	187
6.3.3. Sentencias nulas en bucles <code>for</code>	188
6.3.4. Sentencias <code>break</code> y <code>continue</code>	188
6.4. Repetición: el bucle <code>do...while</code>	190
6.4.1. Diferencias entre <code>while</code> y <code>do-while</code>	191
6.5. Comparación de bucles <code>while</code> , <code>for</code> y <code>do-while</code>	193
6.6. Diseño de bucles	193
6.6.1. Bucles para diseño de sumas y productos	194
6.6.2. Fin de un bucle	194
6.6.3. Otras técnicas de terminación de bucle	196
6.6.4. Bucles <code>for</code> vacíos	196
6.7. Bucles anidados	197
6.8. Resumen	201
6.9. Ejercicios	201
6.10. Problemas	203
6.11. Proyectos de programación	206
Capítulo 7. Funciones	208
7.1. Concepto de función	210
7.2. Estructura de una función	211
7.2.1. Nombre de una función	213

7.2.2.	Tipo de dato de retorno	
7.2.3.	Resultados de una función	
7.2.4.	Llamada a una función	
7.3.	Prototipos de las funciones	
7.3.1.	Prototipos con un número no especificado de parámetros	
7.4.	Parámetros de una función	
7.4.1.	Paso de parámetros por valor	
7.4.2.	Paso de parámetros por referencia	
7.4.3.	Diferencias entre paso de variables por valor y por referencia	
7.4.4.	Parámetros const de una función	
7.5.	Funciones en línea, macros con argumentos	
7.5.1.	Creación de macros con argumentos	
7.6.	Ámbito (alcance)	
7.6.1.	Ámbito del programa	229
7.6.2.	Ámbito del archivo fuente	230
7.6.3.	Ámbito de una función	230
7.6.4.	Ámbito de bloque	230
7.6.5.	Variables locales	231
7.7.	Clases de almacenamiento	231
7.7.1.	Variables automáticas	231
7.7.2.	Variables externas	231
7.7.3.	Variables registro	232
7.7.4.	Variables estáticas	232
7.8.	Concepto y uso de funciones de biblioteca	234
7.9.	Funciones de carácter	234
7.9.1.	Comprobación alfabética y de dígitos	235
7.9.2.	Funciones de prueba de caracteres especiales	236
7.9.3.	Funciones de conversión de caracteres	236
7.10.	Funciones numéricas	237
7.10.1.	Funciones matemáticas	237
7.10.2.	Funciones trigonométricas	238
7.10.3.	Funciones logarítmicas y exponenciales	238
7.10.4.	Funciones aleatorias	239
7.11.	Funciones de fecha y hora	240
7.12.	Funciones de utilidad	243
7.13.	Visibilidad de una función	244
7.13.1.	Variables locales frente a variables globales	245
7.13.2.	Variables estáticas y automáticas	247
7.14.	Compilación separada	249
7.15.	Variables registro (register)	250
7.16.	Recursividad	251
7.17.	Resumen	254
7.18.	Ejercicios	
7.19.	Problemas	
Capítulo 8.	Arrays (listas y tablas)	258
8.1.	Arrays	260
8.1.1.	Declaración de un array	260
8.1.2.	Subíndices de un array	261
8.1.3.	Almacenamiento en memoria de los arrays	262
8.1.4.	El tamaño de los arrays	263
8.1.5.	Verificación del rango del índice de un array	264
8.2.	Inicialización de un array	264
8.3.	Arrays de caracteres y cadenas de texto	266
8.4.	Arrays multidimensionales	269
8.4.1.	Inicialización de arrays multidimensionales	270

8.4.2.	Acceso a los elementos de los arrays bidimensionales	271
8.4.3.	Lectura y escritura de arrays bidimensionales	272
8.4.4.	Acceso a elementos mediante bucles	272
8.4.5.	Arrays de más de dos dimensiones	274
8.4.6.	Una aplicación práctica	274
8.5.	Utilización de arrays como parámetros	276
8.5.1.	Precauciones	279
8.5.2.	Paso de cadenas como parámetros	281
8.6.	Ordenación de listas	282
8.6.1.	Algoritmo de la burbuja	282
8.7.	Búsqueda en listas	284
8.7.1.	Búsqueda secuencial	285
8.8.	Resumen	289
8.9.	Ejercicios	290
8.10.	Problemas	291
Capítulo 9.	Estructuras y uniones	294
9.1.	Estructuras	296
9.1.1.	Declaración de una estructura	297
9.1.2.	Definición de variables de estructuras	297
9.1.3.	Uso de estructuras en asignaciones	298
9.1.4.	Inicialización de una declaración de estructuras	299
9.1.5.	El tamaño de una estructura	300
9.2.	Acceso a estructuras	300
9.2.1.	Almacenamiento de información en estructuras	300
9.2.2.	Lectura de información de una estructura	302
9.2.3.	Recuperación de información de una estructura	302
9.3.	Estructuras anidadas	303
9.3.1.	Ejemplo de estructuras anidadas	304
9.4.	Arrays de estructuras	307
9.4.1.	Arrays como miembros	308
9.5.	Utilización de estructuras como parámetros	309
9.6.	Uniones	310
9.7.	Enumeraciones	311
9.7.1.	sizeof de tipos de datos estructurados	314
9.7.2.	typedef	314
9.8.	Campos de bit	315
9.9.	Resumen	319
9.10.	Ejercicios	320
9.11.	Problemas	321
Capítulo 10.	Punteros (apuntadores)	322
10.1.	Direcciones en memoria	324
10.2.	Concepto de puntero (apuntador)	325
10.2.1.	Declaración de punteros	326
10.2.2.	Inicialización (iniciación) de punteros	327
10.2.3.	Indirección de punteros	321
10.2.4.	Punteros y verificación de tipos	321
10.3.	Punteros null y void	330
10.4.	Punteros a punteros	331
10.5.	Punteros y arrays	332
10.5.1.	Nombres de arrays como punteros	332
10.5.2.	Ventajas de los punteros	333
10.6.	Arrays de punteros	334
10.6.1.	Inicialización de un array de punteros a cadenas	335
10.7.	Punteros de cadenas	335
10.7.1.	Punteros versus arrays	334

10.8. Aritmética de punteros	336
10.8.1. Una aplicación de punteros: conversión de caracteres	338
10.9. Punteros constantes frente a punteros a constantes	339
10.9.1. Punteros constantes	339
10.9.2. Punteros a constantes	339
10.9.3. Punteros constantes a constantes	340
10.10. Punteros como argumentos de funciones	341
10.11. Punteros a funciones	343
10.11.1. Inicialización de un puntero a una función	343
10.11.2. Aplicación de punteros a función para ordenación	347
10.11.3. Arrays de punteros de funciones	348
10.11.4. Una aplicación de punteros de funciones	349
10.12. Punteros a estructuras	350
10.13. Resumen	351
10.14. Ejercicios	352
10.15. Problemas	353
Capítulo 11. Asignación dinámica de memoria	354
11.1. Gestión dinámica de la memoria	356
11.1.1. Almacén libre (<i>free store</i>)	357
11.2. Función <code>malloc ()</code>	357
11.2.1. Asignación de memoria de un tamaño desconocido	361
11.2.2. Uso de <code>malloc ()</code> para arrays multidimensionales	362
11.3. Liberación de memoria, función <code>free ()</code>	363
11.4. Funciones de asignación de memoria <code>calloc ()</code> y <code>realloc ()</code>	364
11.4.1. Función <code>calloc ()</code>	364
11.4.2. Función <code>realloc ()</code>	365
11.5. Asignación de memoria para arrays	368
11.5.1. Asignación de memoria interactivamente	369
11.5.2. Asignación de memoria para un array de estructuras	371
11.6. Arrays dinámicos	373
11.7. Reglas de funcionamiento de la asignación de memoria	374
11.8. Resumen	376
11.9. Ejercicios	376
11.10. Problemas	377
Capítulo 12. Cadenas	378
12.1. Concepto de cadena	380
12.1.1. Declaración de variables de cadena	381
12.1.2. Inicialización de variables de cadena	381
12.2. Lectura de cadenas	382
12.2.1. Función <code>getchar ()</code>	385
12.2.2. Función <code>putchar ()</code>	385
12.2.3. Función <code>puts ()</code>	386
12.2.4. Funciones <code>getch ()</code> y <code>getche ()</code>	387
12.3. La biblioteca <code>string.h</code>	387
12.3.1. La palabra reservada <code>const</code>	389
12.4. Arrays y cadenas como parámetros de funciones	389
12.5. Asignación de cadenas	391
12.5.1. La función <code>strcpy ()</code>	391
12.6. Longitud y concatenación de cadenas	392
12.6.1. La función <code>strlen ()</code>	392
12.6.2. Las funciones <code>strcat ()</code> y <code>strncat ()</code>	393
12.7. Comparación de cadenas	394
12.7.1. La función <code>strcmp ()</code>	395
12.7.2. La función <code>stricmp ()</code>	395

12.7.3. La función <code>strncmp ()</code>	396
12.7.4. La función <code>strnicmp ()</code>	396
12.8. Inversión de cadenas	397
12.9. Conversión de cadenas	397
12.9.1. Función <code>strupr ()</code>	397
12.9.2. Función <code>strlwr ()</code>	398
12.10. Conversión de cadenas a números	399
12.10.1. Función <code>atoi ()</code>	399
12.10.2. Función <code>atof ()</code>	400
12.10.3. Función <code>atol ()</code>	400
12.10.4. Entrada de números y cadenas	400
12.11. Búsqueda de caracteres y cadenas	401
12.11.1. Función <code>strchr ()</code>	401
12.11.2. Función <code>strrchr ()</code>	402
12.11.3. Función <code>strspn ()</code>	402
12.11.4. Función <code>strcspn ()</code>	403
12.11.5. Función <code>strpbrk ()</code>	403
12.11.6. Función <code>strstr ()</code>	404
12.11.7. Función <code>strtok ()</code>	404
12.12. Resumen	405
12.13. Ejercicios	406
12.14. Problemas	407

PARTE III. ESTRUCTURA DE DATOS

Capítulo 13. Entradas y salidas por archivos	410
13.1. Flujos	412
13.2. Puntero <code>FILE</code>	412
13.3. Apertura de un <i>archivo</i>	413
13.3.1. Modos de apertura de un archivo	414
13.3.2. <code>NULL</code> y <code>EOF</code>	415
13.3.3. Cierre de archivos	415
13.4. Creación de un archivo secuencial	416
13.4.1. Funciones <code>putc ()</code> y <code>fputc ()</code>	416
13.4.2. Funciones <code>getc ()</code> y <code>fgetc ()</code>	417
13.4.3. Funciones <code>fputs ()</code> y <code>fgets ()</code>	417
13.4.4. Funciones <code>fprint ()</code> y <code>fscanf ()</code>	417
13.4.5. Función <code>feof ()</code>	421
13.4.6. Función <code>rewind ()</code>	421
13.5. Archivos binarios en C	421
13.5.1. Función de salida <code>fwrite ()</code>	423
13.5.2. Función de lectura <code>fread ()</code>	424
13.6. Funciones para acceso aleatorio	426
13.6.1. Función <code>fseek ()</code>	426
13.6.2. Función <code>ftell ()</code>	431
13.7. Datos externos al programa con <i>argumentos de main ()</i>	431
13.8. Resumen	434
13.9. Ejercicios	435
13.10. Problemas	436
Capítulo 14. Listas enlazadas	438
14.1. Fundamentos teóricos	440
14.2. Clasificación de las listas enlazadas	441
14.3. Operaciones en listas enlazadas	442
14.3.1. Declaración de un nodo	442

14.3.2.	Puntero de cabecera y cola	443
14.3.3.	El puntero nulo	444
14.3.4.	El operador -> de selección de un miembro	445
14.3.5.	Construcción de una lista	445
14.3.6.	Insertar un elemento en una lista	447
14.3.7.	Búsqueda de un elemento	453
14.3.8.	Supresión de un nodo en una lista	454
14.4.	Lista doblemente enlazada	456
14.4.1.	Declaración de una lista doblemente enlazada	457
14.4.2.	Insertar un elemento en una lista doblemente enlazada	458
14.4.3.	Supresión de un elemento en una lista doblemente enlazada	459
14.5.	Listas circulares	462
14.5.1.	Insertar un elemento en una lista circular	462
14.5.2.	Supresión de un elemento en una lista circular	463
14.6.	Resumen	467
14.7.	Ejercicios	468
14.8.	Problemas	468
Capítulo 15.	Pilas y colas	470
15.1.	Concepto de pila	472
15.1.1.	Especificaciones de una pila	473
15.2.	El tipo pila implementado con arrays	473
15.2.1.	Especificación del tipo pila	475
15.2.2.	Implementación de las operaciones sobre pilas	477
15.2.3.	Operaciones de verificación del estado de la pila	478
15.3.	Colas	481
15.4.	El tipo cola implementada con arrays	483
15.4.1.	Definición de la especificación de una cola	483
15.4.2.	Especificación del tipo cola	483
15.4.3.	Implementación del tipo cola	484
15.4.4.	Operaciones de la cola	486
15.5.	Realización de una cola con una lista enlazada	487
15.5.1.	Declaración del tipo cola con listas	488
15.5.2.	Codificación de las operaciones del tipo cola con listas	489
15.6.	Resumen	492
15.7.	Ejercicios	493
15.8.	Problemas	494
Capítulo 16.	Árboles	494
16.1.	Árboles generales	494
16.1.1.	Representación de un árbol	501
16.2.	Resumen de definiciones	504
16.3.	Árboles binarios	504
16.3.1.	Equilibrio	504
16.3.2.	Árboles binarios completos	504
16.4.	Estructura de un árbol binario	511
16.4.1.	Diferentes tipos de representaciones en C	511
16.5.	Operaciones en árboles binarios	511
16.6.	Árboles de expresión	511
16.6.1.	Reglas para la construcción de árboles de expresión	511
16.7.	Recorrido de un árbol	511
16.7.1.	Recorrido <i>preorden</i>	511
16.7.2.	Recorrido <i>enorden</i>	521
16.7.3.	Recorrido <i>postorden</i>	522
16.7.4.	Profundidad de un árbol binario	521
16.8.	Árbol binario de búsqueda	525

16.8.1. Creación de un árbol binario de búsqueda	525
16.8.2. Implementación de un nodo de un árbol binario de búsqueda	527
16.9. Operaciones en árboles binarios de búsqueda	528
16.9.1. Búsqueda	528
16.9.2. Insertar un nodo	529
16.9.3. Función <code>insertar()</code>	530
16.9.4. Eliminación	531
16.9.5. Recorridos de un árbol	535
16.9.6. Determinación de la altura de un árbol	535
16.10. Aplicaciones de árboles en algoritmos de exploración	536
16.10.1. Visita a los nodos de un árbol	537
16.11. Resumen	537
16.12. Ejercicios	539
16.13. Problemas	540
16.14. Referencias bibliográficas para lecturas posteriores	542
APÉNDICES	543
Apéndice A. Lenguajes C ANSI. Guía de referencia	545
Apéndice B. Códigos de caracteres ASCII	575
Apéndice C. Palabras reservadas de C++	579
Apéndice D. Guía de sintaxis ANSI/ISO estándar C++	599
Apéndice E. Biblioteca de funciones ANSI C	649
Apéndice F. Recursos (Libros/Revistas/URL de Internet de C/C++)	713
ÍNDICE	727

PRÓLOGO

INTRODUCCIÓN

¿Por qué un libro de C al principio del siglo XXI? A pesar de haber cumplido ya sus bodas de plata (25 años de vida), C viaja con toda salud hacia los 30 años de edad que cumplirá el próximo año. Sigue siendo una de las mejores opciones para la programación de los sistemas actuales y el medio más eficiente de aprendizaje para emigrar a los lenguajes reina, por excelencia, en el mundo orientado a objetos y componentes y el mundo Web (C++, Java, ...) que dominan el campo informático y de la computación.

¿Cuáles son las características que hacen tan popular a este lenguaje de programación e idóneo como primer lenguaje de programación en las carreras profesionales de programador (de aplicaciones y de sistemas) y del ingeniero de software? Podemos citar algunas muy sobresalientes:

- Es muy *portable* (transportable entre un gran número de plataformas *hardware* y plataformas *software*, *sistemas operativos*). Existen numerosos compiladores para todo tipo de plataformas sobre los que corren los mismos programas fuentes o con ligeras modificaciones.
- Es *versátil* y de *bajo nivel*, por lo que es idóneo para tareas relativas a la programación del sistema.
- A pesar de ser un excelente lenguaje para programación de sistemas, es también un eficiente y potente lenguaje para *aplicaciones de propósito general*.
- Es un lenguaje pequeño, por lo que es relativamente fácil construir compiladores de C y además es también fácil de aprender.
- Todos los compiladores suelen incluir potentes y excelentes bibliotecas de funciones compatibles con el *estándar ANSI*. Los diferentes fabricantes suelen añadir a sus compiladores funcionalidades diversas que aumentan la eficiencia y potencia de los mismos y constituye una notable ventaja respecto a otros lenguajes.
- El lenguaje presenta una *interfaz excelente* para los sistemas operativos Unix y Windows, junto con el ya acreditado Linux.
- Es un lenguaje *muy utilizado para la construcción de*: sistemas operativos, ensambladores, programas de comunicaciones, intérpretes de lenguajes, compiladores de lenguajes, editores de textos, bases de datos, utilidades, controladores de red, etc.

Por todas estas razones y nuestra experiencia docente, decidimos escribir esta obra que, por otra parte, pudiera completar nuestras otras obras de programación escritas para C++, Java, Turbo Pascal y Visual Basic. Basados en estas premisas este libro se ha escrito pensando en que pudiera servir de

referencia y guía de estudio para un primer curso de *introducción a la programación*, con una segunda parte que, a su vez, sirviera como continuación, y de *introducción a las estructuras de datos* todo ello utilizando C, y en particular la versión estándar **ANSI C**, como lenguaje de programación. El objetivo final que busca es, *no sólo* describir la sintaxis de C, *sino* y, *sobre todo*, mostrar las características más sobresalientes del lenguaje, a la vez que se enseñan técnicas de programación estructurada. **Así** pues, los objetivos fundamentales del libro son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Un medio de resolución de problemas mediante técnicas de programación.
- Una introducción a la informática y a las ciencias de la computación usando una herramienta de programación denominada C (**ANSI C**).
- Enseñanza de las reglas de sintaxis más frecuentes y eficientes del lenguaje C.

En resumen, éste es un libro diseñado para *enseñar a programar utilizando C*, no un libro diseñado para enseñar C, aunque también pretende conseguirlo. **No** obstante, confiamos que los estudiantes que utilicen este libro se conviertan de un modo razonable en acérrimos seguidores y adeptos de C, al igual que nos ocurre a casi todos los programadores que comenzamos a trabajar con este lenguaje. **Así** se tratará de enseñar las técnicas clásicas y avanzadas de programación estructurada.

LA EVOLUCIÓN DE C: C++

C es un lenguaje de programación de propósito general que ha estado y sigue estando asociado con el sistema operativo **UNIX**. El advenimiento de nuevos sistemas operativos como **Windows** (95, 98, NT, 2000 o el recientemente anunciado *XP* sobre la plataforma *NET*) o el ya muy popular **Linux**, la versión abierta, gratuita de Unix que junto con el entorno *Gnome* está comenzando a revolucionar el mundo de la programación. Esta revolución, paradójicamente, proporciona fuerza al lenguaje de programación de sistemas C. Todavía y durante muchos años C seguirá siendo uno de los lenguajes líderes en la enseñanza de la programación tanto a nivel profesional como universitario. Como reconocen sus autores Kernighan y Ritchie, en *El Lenguaje de Programación C*, 2.^a edición, C, aunque es un lenguaje idóneo para escribir compiladores y sistemas operativos, sigue siendo, sobre todo, un lenguaje para escribir aplicaciones en numerosas disciplinas. Ésta es la razón por la que a algo más de un año para cumplir los 30 años de vida, C sigue siendo el lenguaje más empleado en *Facultades y Escuelas de Ciencias e Ingeniería*, y en los *centros de enseñanza de formación profesional*, y en particular los innovadores *ciclos de grado superior*, así como en *centros de enseñanza media y secundaria*, para el aprendizaje de legiones de promociones (generaciones) de estudiantes y profesionales.

Las ideas fundamentales de C provienen del lenguaje BCPL, desarrollado por Martin Richards. La influencia de BCPL sobre C continuó, indirectamente, a través del lenguaje **B**, escrito por Ken Thompson en 1979 para escribir el primer sistema UNIX de la computadora DEC de Digital PDP-7. BCPL y B son lenguajes «*sin tipos*» en contraste con C que posee una variedad de tipos de datos.

En 1975 se publica *Pascal User Manual and Report* la especificación del joven lenguaje Pascal (Wirth, Jensen 75) cuya suerte corre en paralelo con C, aunque al contrario que el compilador de Pascal construido por la casa Borland, que prácticamente no se comercializa, C sigue siendo uno de los reyes de la iniciación a la programación. En 1978 se publicó la primera edición de la obra *The C Programming Language* de Kernighan y Ritchie, conocido por K&R.

En 1983 el American National Standards Institute (**ANSI**) nombró un comité para conseguir una definición estándar de C. La definición resultante se llamó **ANSI C**, que se presentó a finales de **1988** y se aprobó definitivamente por **ANSI** en 1989 y en 1990 se aprobó por **ISO**. La segunda edición *The C Programming Language* se considera también el manual del estándar **ANSI C**. Por esta razón la especificación estándar se suele conocer como **ANSVISO C**. Los compiladores modernos soportan todas las características definidas en ese estándar.

Conviviendo con C se encuentra el lenguaje C++, una evolución lógica suya, y que es tal el estado de simbiosis y sinergia existente entre ambos lenguajes que en muchas ocasiones se habla de C/C++ para definir a los compiladores que siguen estas normas, dado que C++ se considera un superconjunto de C.

C++ tiene sus orígenes en C, y, sin lugar a dudas, Kernighan y Ritchie — inventores de C, — son «padres espirituales» de C++. Así lo manifiesta Bjarne Stroustrup — inventor de C++ — en el prólogo de su afamada obra *The C++ Programming Language*. C se ha conservado así como un subconjunto de C++ y es, a su vez, extensión directa de su predecesor BCPL de Richards. Pero C++, tuvo muchas más fuentes de inspiración; además de los autores antes citados, cabe destacar de modo especial, Simula 67 de Dahl que fue su principal inspirador; el concepto de *clase, clase derivada y funciones virtuales* se tomaron de Simula; otra fuente importante de referencia fue Algol 68 del que se adoptó el concepto de *sobrecarga de operadores* y la *libertad de situar una declaración en cualquier lugar* en el que pueda aparecer una sentencia. Otras aportaciones importantes de C++ como son las plantillas (*templates*) y la genericidad (*tipos genéricos*) se tomaron de Ada, Clu y ML.

C++ se comenzó a utilizar como un «C con clases» y fue a principios de los ochenta cuando comenzó la revolución C++, aunque su primer uso comercial, fuera de una organización de investigación, comenzó en julio de 1983. Como Stroustrup cuenta en el prólogo de la 3.ª edición de su citada obra, C++ nació con la idea de que el autor y sus colegas no tuvieran que programar en ensamblador ni en otros lenguajes al uso (*léase Pascal, BASIC, FORTRAN,...*). La explosión del lenguaje en la comunidad informática hizo inevitable la estandarización. proceso que comenzó en 1987 [Stroustrup94]. Así nació una primera fuente de estandarización *The Annotated C++ Reference Manual* [Ellis 89]¹. En diciembre de 1989 se reunió el comité X3J16 de ANSI, bajo el auspicio de Hewlett-Packard y en junio de 1991 pasó el primer esfuerzo de estandarización internacional de la mano de ISO, y así comenzó a nacer el estándar ANSVISO C++. En 1995 se publicó un borrador estándar para su examen público y en noviembre de 1997 fue finalmente aprobado el estándar C++ internacional, aunque ha sido en 1998 cuando el proceso se ha podido dar por terminado (ANSI/ISO C++ Draft Standard).

El libro definitivo y referencia obligada para conocer y dominar C++ es la 3.ª edición de la obra de Stroustrup [Stroustrup 97] y actualizada en la *Special Edition* [Stroustrup 2000]².

OBJETIVOS DEL LIBRO

C++ es un superconjunto de C y su mejor extensión. Éste es un tópico conocido por toda la comunidad de programadores del mundo. Cabe preguntarse como hacen muchos autores, profesores, alumnos y profesionales ¿se debe aprender primero C y luego C++? Stroustrup y una gran mayoría de programadores, contestan así: «*No sólo es innecesario aprender primero C, sino que además es una mala idea*». Nosotros no somos tan radicales y pensamos que se puede llegar a C++ procediendo de ambos caminos, aunque es lógico la consideración citada anteriormente, ya que efectivamente los hábitos de programación estructurada de C pueden retrasar la adquisición de los conceptos clave de C++, pero también es cierto que en muchos casos ayuda considerablemente en el aprendizaje.

Este libro supone que el lector no es programador de C, ni de ningún otro lenguaje, aunque también somos conscientes que el lector que haya seguido un primer curso de programación en algoritmos o en algún lenguaje estructurado, llámese Pascal o cualquier otro, éste le ayudará favorablemente al correcto y rápido aprendizaje de la programación en C y obtendrá el máximo rendimiento de esta obra. Sin embargo, si ya conoce C++, naturalmente no tendrá ningún problema, en su aprendizaje, muy al contrario, bastará que lea con detalle las diferencias esenciales de los apéndices C y D de modo que irá

¹ Traducida al español por el autor de este libro junto con el profesor Miguel Katnb, de la Universidad de la Habana [Ellis 94]

² Esta obra se encuentra en proceso de traducción al español por un equipo de profesores de vanas universidades españolas coordinadas por el autor de esta obra

integrando gradualmente los nuevos conceptos que irá encontrando a medida que avance en la obra con los conceptos clásicos de C++. El libro pretende enseñar a programar utilizando dos conceptos fundamentales:

1. *Algoritmos* (conjunto de instrucciones programadas para resolver una tarea específica).
2. *Datos* (una colección de datos que se proporcionan a los algoritmos que se han de ejecutar para encontrar una solución: los datos se organizarán en *estructuras de datos*).

Los dos primeros aspectos, *algoritmos* y *datos*, han permanecido invariables a lo largo de la corta historia de la informática/computación, pero la *interrelación* entre ellos **sí** que ha variado y continuará haciéndolo. Esta *interrelación* se conoce como *paradigma de programación*.

En el paradigma de programación *procedimental* (*procedural* o *por procedimientos*) un problema se modela directamente mediante un conjunto de algoritmos. Un problema cualquiera, la nómina de una empresa o la gestión de ventas de un almacén, se representan como una serie de procedimientos que manipulan datos. Los datos se almacenan separadamente y se accede a ellos o bien mediante una posición global o mediante parámetros en los procedimientos. Tres lenguajes de programación clásicos, FORTRAN, Pascal y C, han representado el arquetipo de la programación *procedimental*, también relacionada estrechamente y —a veces— conocida como *programación estructurada*. La programación con soporte en C++, proporciona el paradigma *procedimental* con un énfasis en funciones, plantillas de funciones y algoritmos genéricos.

En la década de los setenta, el enfoque del diseño de programas se desplazó desde el paradigma *procedimental* al orientado a objetos apoyado en los *tipos abstractos de datos (TAD)*. En este paradigma un problema modela un conjunto de abstracciones de datos. En C++ estas abstracciones se conocen como *clases*. Las clases contienen un conjunto de instancias o ejemplares de la misma que se denominan *objetos*, de modo que un programa actúa como un conjunto de objetos que se relacionan entre **sí**. La gran diferencia entre ambos paradigmas reside en el hecho de que los algoritmos asociados con cada clase se conocen como *interfaz pública* de la clase y los datos se almacenan privadamente dentro de cada objeto de modo que el acceso a los datos está oculto al programa general y se gestionan a través de la interfaz.

Así pues, en resumen, los objetivos fundamentales de esta obra son: introducción a la *programación estructurada* y *estructuras de datos* con el lenguaje estándar C de ANSVISO; otro objetivo complementario es preparar al lector para su emigración a C++, para lo cual se han escrito dos apéndices completos C y D que presentan una amplia referencia de palabras reservadas y una guía de sintaxis de C++ con el objeto de que el lector pueda convertir programas escritos en C a C++ (con la excepción de las propiedades de orientación a objetos que se salen fuera del ámbito de esta obra).

EL LIBRO COMO HERRAMIENTA DOCENTE

La experiencia de los autores desde hace muchos años con obras muy implantadas en el mundo universitario como *Programación en C++*, *Programación en Turbo Pascal* (en su 3.ª edición), *estructura de datos*, *Fundamentos de programación* (en su 2.ª edición y en preparación la 3.ª edición) y *Programación en BASIC* (que alcanzó tres ediciones y numerosas reimpresiones en la década de los ochenta), nos ha llevado a mantener la estructura de estas obras, actualizándola a los contenidos que se prevén para los estudiantes del futuro siglo XXI. Por ello en el contenido de la obra hemos tenido en cuenta no sólo las directrices de los planes de estudio españoles de ingeniería informática e ingeniería técnica informática (antiguas licenciaturas y diplomaturas en informática) y licenciaturas en ciencias de la computación, sino también de ingenierías tales como industriales, telecomunicaciones, agrónomos o minas, o las más recientes incorporadas, en España, como ingeniería en geodesia. Asimismo, en el diseño de la obra se han tenido en cuenta las directrices oficiales vigentes en España para la Formación Profesional de Grado Superior; por ello se ha tratado de que el contenido de la obra contemple los programas propuestos para el ciclo de desarrollo de *Aplicaciones Informáticas* en el módulo de *Programación en Lenguaje Estructurado*; también se ha tratado en la medida de lo posible de que pueda servir de

referencia al ciclo de *Administración de Sistemas Informáticos* en el módulo de *Fundamentos de Programación*.

Nuestro conocimiento del mundo educativo latinoamericano nos ha llevado a pensar también en las carreras de *ingeniería de sistemas computacionales* y las *licenciaturas en informática y en sistemas de información*, carreras hermanas de las citadas anteriormente.

Por todo lo anterior, el contenido del libro intenta seguir un programa estándar de un primer curso de introducción a la programación y, según situaciones, un segundo curso de programación de nivel medio en asignaturas tales como *Metodología de la Programación*, *Fundamentos de Programación*, *Introducción a la Programación*, ... Asimismo, se ha buscado seguir las directrices emanadas de la ACM-IEEE para los cursos **CS1** y **CS8** en los planes recomendados en los *Computing Curricula* de 1991 y las recomendaciones de los actuales *Computing Curricula 2001* en las áreas de conocimiento *Programming Fundamentals* [PF,10] y *Programming Languages* [PL,11], así como las vigentes en universidades latinoamericanas que conocemos, y con las que tenemos relaciones profesionales.

El contenido del libro abarca los citados programas y comienza con la *introducción a los algoritmos y a laprogramación*, para llegar a *estructuras de datos*. Por esta circunstancia la estructura del curso no ha de ser secuencial en su totalidad sino que el profesor/maestro y el alumno/lector podrán estudiar sus materias en el orden que consideren más oportuno. Ésta es la razón principal por la cual el libro se ha organizado en tres partes y en seis apéndices.

Se trata de describir el *paradigma* más popular en el mundo de la programación: el *procedimental* y preparar al lector para su inmersión en el ya implantado *paradigma orientado a objetos*. Los cursos de programación en sus niveles inicial y medio están evolucionando para aprovechar las ventajas de nuevas y futuras tendencias en ingeniería de software y en diseño de lenguajes de programación, específicamente diseño y programación orientada a objetos. Algunas facultades y escuelas de ingenieros, junto con la nueva formación profesional (ciclos formativos de nivel superior) en España y en Latinoamérica, están introduciendo a sus alumnos en la programación orientada a objetos, inmediatamente después del conocimiento de la programación estructurada, e incluso —en ocasiones antes—. Por esta razón, una metodología que se podría seguir sería impartir un curso de *fundamentos de programación* seguido de *estructuras de datos* y luego seguir con un segundo nivel de *programación avanzada* que constituyen las tres partes del libro. Pensando en aquellos alumnos que deseen continuar su formación estudiando C++ se han escrito los apéndices C y D, que les permita adaptarse fácilmente a las particularidades básicas de C++ y poder continuar sin esfuerzo la parte primera y avanzar con mayor rapidez a las siguientes partes del libro.

CARACTERÍSTICAS IMPORTANTES DEL LIBRO

Programación en C, utiliza los siguientes elementos clave para conseguir obtener el mayor rendimiento del material incluido en sus diferentes capítulos:

- *Contenido*. Enumera los apartados descritos en el capítulo.
- *Introducción*. Abre el capítulo con una breve revisión de los puntos y objetivos más importantes que se tratarán y todo aquello que se puede esperar del mismo.
- *Conceptos clave*. Enumera los términos informáticos y de programación más notables que se tratarán en el capítulo.
- *Descripción del capítulo*. Explicación usual de los apartados correspondientes del capítulo. En cada capítulo se incluyen ejemplos y ejercicios resueltos. Los listados de los programas completos o parciales se escriben en letra *courier* con la finalidad principal de que puedan ser identificados fácilmente por el lector.
- *Resumen del capítulo*. Revisa los temas importantes que los estudiantes y lectores deben comprender y recordar. Busca también ayudar a reforzar los conceptos clave que se han aprendido en el capítulo.

- **Ejercicios.** Al final de cada capítulo se proporciona a los lectores una lista de ejercicios sencillos de modo que le sirvan de oportunidad para que puedan medir el avance experimentado mientras leen y siguen - e n su caso— las explicaciones del profesor relativas al capítulo.
- **Problemas.** Después del apartado *Ejercicios*, se añaden una serie de actividades y proyectos de programación que se le proponen al lector como tarea complementaria de los ejercicios y de un nivel de dificultad algo mayor.

A lo largo de todo el libro se incluyen una serie de recuadros —sombreados o no— que ofrecen al lector consejos, advertencias y reglas de uso del lenguaje y de técnicas de programación, con la finalidad de que puedan ir asimilando conceptos prácticos de interés que les ayuden en el aprendizaje y construcción de programas eficientes y de fácil lectura.

- **Recuadro.** Conceptos importantes que el lector debe considerar durante el desarrollo del capítulo.
- **Consejo.** Ideas, sugerencias, recomendaciones, ... al lector, con el objetivo de obtener el mayor rendimiento posible del lenguaje y de la programación.
- **Precaución.** Advertencia al lector para que tenga cuidado al hacer uso de los conceptos incluidos en el recuadro adjunto.
- **Reglas.** Normas o ideas que el lector debe seguir preferentemente en el diseño y construcción de sus programas.

ORGANIZACIÓN DEL LIBRO

El libro se divide en tres partes que unidas constituyen un curso completo de programación en C. Dado que el conocimiento es acumulativo, los primeros capítulos proporcionan el fundamento conceptual para la comprensión y aprendizaje de C y una guía a los estudiantes a través de ejemplos y ejercicios sencillos y los capítulos posteriores presentan de modo progresivo la programación en C en detalle, en el paradigma *procedimental*. Los apéndices contienen un conjunto de temas importantes que incluyen desde guías de sintaxis de ANSI/ISO C, hasta o una biblioteca de funciones y clases, junto con una extensa bibliografía de algoritmos, estructura de datos, programación orientada a objetos y una amplia lista de sitios de Internet (URLs) donde el lector podrá complementar, ampliar y profundizar en el mundo de la programación y en la introducción a la ingeniería de software.

PARTE I. METODOLOGÍA DE LA PROGRAMACIÓN

Esta parte es un primer curso de programación para alumnos principiantes en asignaturas de introducción a la programación en lenguajes estructurados. Esta parte sirve tanto para cursos de C como de C++ (en este caso con la ayuda de los apéndices C y D). Esta parte comienza con una introducción a la informática y a las ciencias de la computación como a la programación. Describe **los** elementos básicos constitutivos de un programa y las herramientas de programación utilizadas tales como algoritmos, diagramas de flujo, etc. Asimismo se incluye un curso del lenguaje C y técnicas de programación que deberá emplear el lector en su aprendizaje de programación. La obra se estructura en tres partes: *Metodología de programación* (conceptos básicos para el análisis, diseño y construcción de programas), *Fundamentos de programación en C* (sintaxis, reglas y criterios de construcción del lenguaje de programación C junto con temas específicos de C como punteros, arrays, cadenas,...), *Estructura de datos* (en esta parte se analizan los archivos y las estructuras dinámicas de datos tales como listas enlazadas, pilas, colas y árboles). Completa la obra una serie de apéndices que buscan esencialmente proporcionar información complementaria de utilidad para el lector en su período de aprendizaje en programación en C, así como *un pequeño curso de C++* en forma de palabras reservadas y guía de referencia de sintaxis que permita al lector emigrar al lenguaje C++ facilitándole para ello las reglas y normas necesarias para convertir programas escritos en C a programas escritos en C++.

Capítulo 1. Introducción a la ciencia de la computación y a la programación. Proporciona una revisión de las características más importantes necesarias para seguir bien un curso de programación básico y avanzado en C. Para ello se describe la organización física de una computadora junto con los conceptos de *algoritmo* y de *programa*. Asimismo se explican los diferentes tipos de lenguajes de programación y una breve historia del lenguaje C.

Capítulo 2. Fundamentos de programación. En este capítulo se describen las fases de resolución de un problema y los diferentes tipos de programación (*modular* y *estructurada*). Se explican también las herramientas de programación y representaciones gráficas utilizadas más frecuentemente en el mundo de la programación.

PARTE II. FUNDAMENTOS DE PROGRAMACIÓN EN C

Capítulo 3. El lenguaje C: Elementos básicos. Enseña la estructura general de un programa en C junto con las operaciones básicas de creación, ejecución y depuración de un programa. Se describen también los elementos clave de un programa (palabras reservadas, comentarios, tipos de datos, constantes y variables,...) junto con los métodos para efectuar entrada y salida de datos a la computadora.

Capítulo 4. Operadores y expresiones. Se describen los conceptos y tipos de operadores y expresiones, conversiones y precedencias. Se destacan operadores especiales tales como *manipulación de bits*, *condicional*, `sizeof`, `()`, `[]`, `::`, `coma`, etc.

Capítulo 5. Estructuras de selección: sentencias *if* y *switch* Introduce al concepto de estructura de control y, en particular, estructuras de selección, tales como *if*, *if-else*, *case* y *switch*. Expresiones condicionales con el operador `?:`, evaluación en cortocircuito de expresiones lógicas, errores frecuentes de programación y puesta a punto de programas.

Capítulo 6. Estructuras repetitivas: bucles (*for*, *while* y *do-while*). El capítulo introduce las estructuras repetitivas (*for*, *while* y *do-while*). Examina la repetición (*iteración*) de sentencias en detalle y compara los bucles controlados por centinela, bandera, etc. Explica precauciones y reglas de uso de diseño de bucles. Compara los tres diferentes tipos de bucles, así como el concepto de bucles anidados.

Capítulo 7. Funciones. Examina el diseño y construcción de módulos de programas mediante funciones. Se define la estructura de una función, prototipos y parámetros. El concepto de funciones en línea (*inline*). Uso de bibliotecas de funciones, clases de almacenamiento, ámbitos, visibilidad de una función. Asimismo se introduce el concepto de recursividad y plantillas de funciones.

Capítulo 8. Arrays (listas y tablas). Examina la estructuración de los datos en arrays o grupos de elementos dato del mismo tipo. El capítulo presenta numerosos ejemplos de arrays de uno, dos o múltiples índices. Se realiza una introducción a los algoritmos de ordenación y búsqueda de elementos en una lista.

Capítulo 9. Estructuras y uniones. Conceptos de estructuras, declaración, definición, iniciación, uso y tamaño. Acceso a estructuras, arrays de estructuras y estructuras anidadas. Uniones y enumeraciones.

Capítulo 10. Punteros (apuntadores). Presenta una de las características más potentes y eficientes del lenguaje C, los *punteros*. Este capítulo proporciona explicación detallada de los punteros, arrays de punteros, punteros de cadena, aritmética de punteros, punteros constantes, punteros como argumentos de funciones, punteros a funciones y a estructuras.

Capítulo 11. *Asignación dinámica de memoria.* En este capítulo se describe la gestión dinámica de la memoria y las funciones asociadas para esas tareas: `malloc ()`, `free ()`, `calloc ()`, `realloc ()`. Se dan reglas de funcionamiento de esas funciones y para asignación y liberación de memoria. También se describe el concepto de arrays dinámicos y asignación de memoria para arrays.

Capítulo 12. *Cadenas.* Se examina el concepto de cadena (*string*) así como las relaciones entre punteros, arrays y cadenas en C. Se introducen conceptos básicos de manipulación de cadenas junto con operaciones básicas tales como longitud, concatenación, comparación, conversión y búsqueda de caracteres y cadenas. Se describen las funciones más notables de la biblioteca `string.h`.

PARTE III. ESTRUCTURA DE DATOS

Esta parte es clave en el aprendizaje de técnicas de programación. Tal es su importancia que los planes de estudio de cualquier carrera de ingeniería informática o de ciencias de la computación incluyen una asignatura troncal denominada *Estructura de datos*.

Capítulo 13. *Archivos.* El concepto de archivo junto con su definición e implementación es motivo de estudio en este capítulo. Las operaciones usuales se estudian con detenimiento.

Capítulo 14. *Listas enlazadas.* Una lista enlazada es una estructura de datos que mantiene una colección de elementos, pero el número de ellos no se conoce por anticipado o varía en un amplio rango. La lista enlazada se compone de elementos que contienen un valor y un puntero. El capítulo describe los fundamentos teóricos y las operaciones que se pueden realizar en la lista enlazada. También se describen los distintos tipos de listas enlazadas.

Capítulo 15. *Pilas y colas. Colas de prioridades.* Las ideas abstractas de pila y cola se describen en el capítulo. Pilas y colas se pueden implementar de diferentes maneras, bien con vectores (arrays) o con listas enlazadas.

Capítulo 16. *Árboles.* Los árboles son otro tipo de estructura de datos dinámica y no lineal. Las operaciones básicas en los árboles junto con sus operaciones fundamentales se estudian en el Capítulo 21.

APÉNDICES

En todos los libros dedicados a la enseñanza y aprendizaje de técnicas de programación es frecuente incluir apéndices de temas complementarios a los explicados en los capítulos anteriores. Estos apéndices sirven de guía y referencia de elementos importantes del lenguaje y de la programación de computadoras.

Apéndice A. *Lenguaje ANSI C. Guía de referencia.* Descripción detallada de los elementos fundamentales del estándar C.

Apéndice B. *Códigos de caracteres ASCII.* Listado del juego de caracteres del código ASCII utilizado en la actualidad en la mayoría de las computadoras.

Apéndice C. *Palabras reservadas de C++.* Listado por orden alfabético de las palabras reservadas en ANSI/ISO C++, al estilo de diccionario. Definición y uso de cada palabra reservada, con ejemplos sencillos de aplicación.

Apéndice D. *Guía de sintaxis ANSI/ISO estándar C++.* Referencia completa de sintaxis de C++ para que junto con las palabras reservadas facilite la migración de programas C a C++ y permita al lector convertir programas estructurados escritos en C a C++.

Apéndice E. Biblioteca de funciones estándar ANSI C. Diccionario en orden alfabético de las funciones estándar de la biblioteca estándar de ANSI/ISO C++, con indicación de la sintaxis del prototipo de cada función, una descripción de su misión junto con algunos ejemplos sencillos de la misma.

Apéndice F. Recursos de C (Libros, Revistas, URLs de Internet). Enumeración de los libros más sobresalientes empleados por los autores en la escritura de esta obra, así como otras obras importantes complementarias que ayuden al lector que desee profundizar o ampliar aquellos conceptos que considere necesario conocer con más detenimiento. Asimismo se adjuntan direcciones de Internet importantes para el programador de C junto con las revistas más prestigiosas del sector informático y de computación en el campo de programación.

AGRADECIMIENTOS

Un libro nunca es fruto único del autor, sobre todo si el libro está concebido como libro de texto y autoaprendizaje, y pretende llegar a lectores y estudiantes de informática y de computación, y, en general, de ciencias e ingeniería, formación profesional de grado superior, . . . , así como autodidactas en asignaturas relacionadas con la programación (*introducción, fundamentos, avanzada*, etc.). Esta obra no es una excepción a la regla y son muchas las personas que nos han ayudado a terminarla. En primer lugar nuestros colegas de la *Universidad Pontificia de Salamanca en el campus de Madrid*, y en particular del *Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software* de la misma que desde hace muchos años nos ayudan y colaboran en la impartición de las diferentes asignaturas del departamento y sobre todo en la elaboración de los programas y planes de estudio de las mismas. A todos ellos les agradecemos públicamente su apoyo y ayuda.

Asimismo deseamos expresar nuestro agradecimiento a la innumerable cantidad de colegas (profesores y maestros) de universidades españolas y latinoamericanas que utilizan nuestros libros para su clases y laboratorios de prácticas. Estos colegas no sólo usan nuestros textos sino que nos hacen sugerencias y nos dan consejos de cómo mejorarlos. Nos sería imposible citarlos a todos por lo que sólo podemos mostrar nuestro agradecimiento eterno por su apoyo continuo.

De igual modo no podemos olvidarnos de *la razón fundamental de ser de este libro: los lectores*. A ellos también mi agradecimiento eterno. A nuestros alumnos de España y Latinoamérica; a los que no siendo alumnos personales, lo son «virtuales» al saber que existen y que con sus lecturas, sus críticas, sus comentarios, hacen que sigamos trabajando pensando en ellos; y a los numerosos lectores profesionales o autodidactas que confían en nuestras obras y en particular en ésta. A todos ellos nuestro reconocimiento más sincero de gratitud.

Además de estos compañeros en docencia, no puedo dejar de agradecer, una vez más, a nuestra editora —y, sin embargo, amiga— **Concha Fernandez**, las constantes muestras de afecto y comprensión que siempre tiene, y ésta no ha sido una excepción, hacia nuestras personas y nuestra obra. Sus continuos consejos, sugerencias y recomendaciones, siempre son acertadas y, además, fáciles de seguir; por si eso no fuera suficiente, *siempre* benefician a la obra.

A riesgo de ser reiterativos, nuestro reconocimiento y agradecimiento eterno a todos: alumnos, lectores, colegas, profesores, maestros, monitores y editores. Gracias por vuestra inestimable e impagable ayuda.

En Carhelejo, Jaén (Andalucía) y en Madrid, Febrero de 2001.

Los autores

P A R T E I

**METODOLOGÍA
DE LA PROGRAMACIÓN**

CAPÍTULO 1

INTRODUCCIÓN A LA CIENCIA DE LA COMPUTACIÓN Y A LA PROGRAMACIÓN

CONTENIDO

- | | |
|---|--|
| 1.1. ¿Qué es una computadora? | 1.5. Los lenguajes de programación. |
| 1.2. ¿Qué es programación? | 1.6. El lenguaje C: historia y características. |
| 1.3. Organización física de una computadora. | 1.7. <i>Resumen.</i> |
| 1.4. Algoritmos y programas. | |

INTRODUCCIÓN

Las computadoras electrónicas modernas son uno de los productos más importantes del siglo xx y especialmente de las dos últimas décadas. Son una herramienta esencial en muchas áreas: industria, gobierno, ciencia, educación..., en realidad en casi todos los campos de nuestras vidas. El papel de los programas de computadoras es esencial; sin una lista de instrucciones a seguir, la computadora es virtualmente inútil. Los lenguajes de programación nos permiten escribir esos programas y, por consiguiente, comunicarnos con las computadoras.

En esta obra, **usted** comenzará a estudiar la ciencia de **las** computadoras o informática a través de uno de los lenguajes de programación **más** versátiles disponibles hoy día: el lenguaje C. **Este** capítulo le introduce a la computadora y **sus** componentes, así como a los lenguajes de programación, y a la metodología a seguir para la resolución de problemas con computadoras y con una herramienta denominada C.

La principal razón para que **las** personas aprendan lenguajes y técnicas de programación es *utilizar la computadora como una herramienta para resolver problemas*.

CONCEPTOS CLAVE

- Algoritmo.
- Athlon.
- Byte.
- **CD-ROM.**
- **Compilación independiente.**
- Compilador.
- Computadora.
- Disquete.
- **DVD.**
- **Editor.**
- GB.
- *Hardware.*
- Hz.
- Intel.
- Intérprete.
- **KB.**
- Lenguaje de programación.
- Lenguaje ensamblador.
- Lenguaje máquina.
- **MB.**
- Memoria.
- Memoria auxiliar.
- Memoria central.
- Módem,
- **MHz.**
- Microprocesador.
- Ordenador.
- Pentium.
- Portabilidad.
- *software.*
- Unidad Central de Proceso.

1.1. ¿QUÉ ES UNA COMPUTADORA?

Una **computadora**¹ es un dispositivo electrónico utilizado para procesar información y obtener resultados. Los datos y la información se pueden introducir en la computadora por la **entrada** (*input*) y a continuación se procesan para producir una **salida** (*output*, resultados), como se observa en la Figura 1.1. La computadora se puede considerar como una unidad en la que se ponen ciertos datos, o *entrada de datos*. La computadora procesa estos datos y produce unos *datos de salida*. Los datos de entrada y los datos de salida pueden ser, realmente, cualquier cosa, texto, dibujos o sonido. El sistema más sencillo de comunicarse con la computadora una persona es mediante un teclado, una pantalla (monitor) y un ratón (*mouse*). Hoy día existen otros dispositivos muy populares tales como escáneres, micrófonos, altavoces, cámaras de vídeo, etc.; de igual manera, a través de *módems*, es posible conectar su computadora con otras computadoras a través de la red **Internet**.

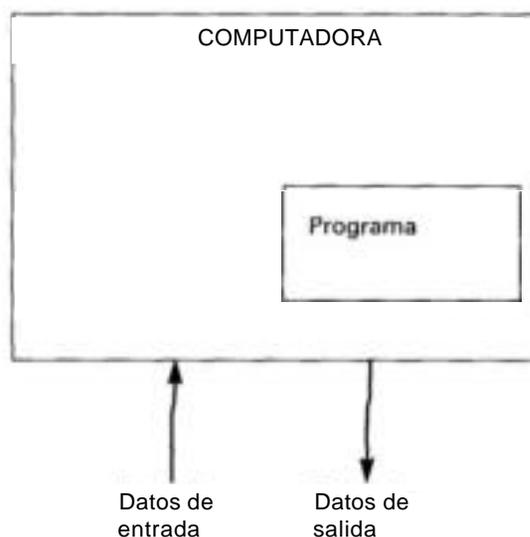


Figura 1.1. Proceso de información en una computadora.

Los componentes físicos que constituyen la computadora, junto con los dispositivos que realizan las tareas de entrada y salida, se conocen con el término **hardware** (traducido en ocasiones por *materia*). El conjunto de instrucciones que hacen funcionar a la computadora se denomina **programa** que se encuentra almacenado en su memoria; a la persona que escribe programas se llama *programador* y al conjunto de programas escritos para una computadora se llama **software** (traducido en ocasiones por *logical*). Este libro se dedicará casi exclusivamente al *software*, pero se **hará** una breve revisión del *hardware* como recordatorio o introducción según sean los conocimientos del lector en esta materia,

1.2. ORGANIZACIÓN FÍSICA DE UNA COMPUTADORA (HARDWARE)

La mayoría de las computadoras, grandes o pequeñas, están organizadas como se muestra en la Figura 1.2. Ellas constan fundamentalmente de tres componentes principales: *unidad central de proceso* (**UCP**) o *procesador* (compuesta de la **UAL**, Unidad aritmético-lógica y la **UOC**, Unidad de Control), la *memoria principal* o *central* y el *programa*.

¹ En España está muy extendido el término **ordenador** para referirse a la traducción de la palabra inglesa **computer**.

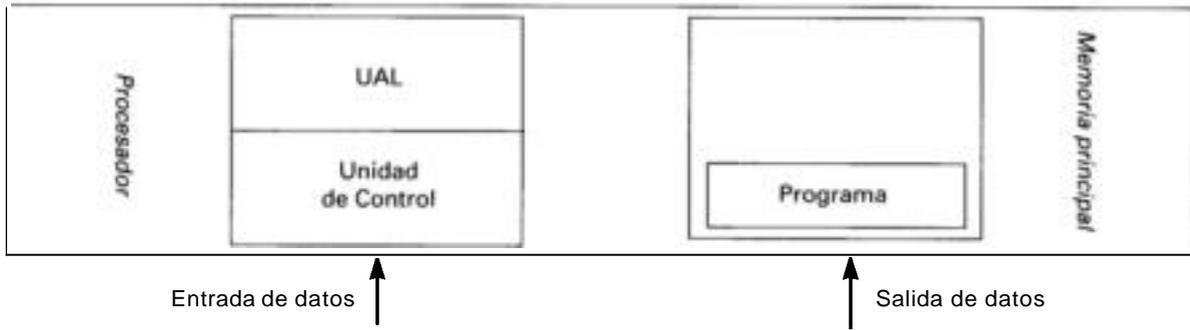


Figura 1.2. Organización física de una computadora.

Si a la organización física de la Figura 1.2 se le añaden los dispositivos para comunicación con la computadora, aparece la estructura típica de un sistema de computadora: *dispositivos de entrada, dispositivo de salida, memoria externa y el procesador/memoria central con su programa* (Fig. 1.3).

1.2.1. Dispositivos de Entrada/Salida (E/S)

Los dispositivos de *Entrada/Salida (E/S)* [*Input/Output (I/O*, en inglés)] permiten la comunicación entre la computadora y el usuario. *Los dispositivos de entrada*, como su nombre indica, sirven para introducir datos (información) en la computadora para su proceso. Los datos se *leen* de los dispositivos de entrada y se almacenan en la memoria central o interna. Los dispositivos de entrada convierten la información de entrada en señales eléctricas que se almacenan en la memoria central. Dispositivos de entrada típicos son los **teclados**; otros son: **lectores de tarjetas** —ya en desuso—, **lápices Ópticos**, **palancas de mando (joystick)**, **lectores de códigos de barras**, **escáneres**, **micrófonos**, etc. Hoy día tal vez el dispositivo de entrada más popular es el **ratón** (mouse) que mueve un puntero electrónico sobre la pantalla que facilita la interacción usuario-máquina².

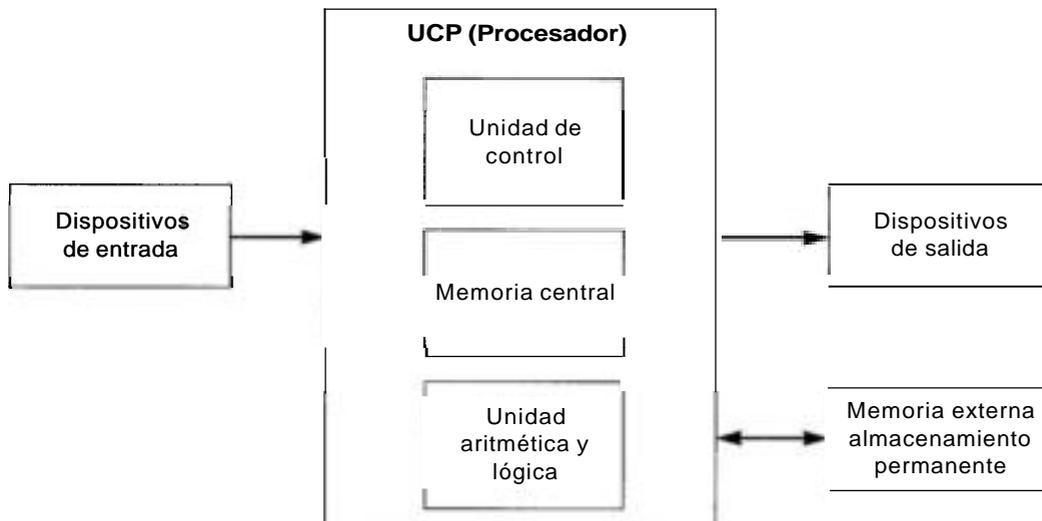


Figura 1.3. Organización física de una computadora.

² Todas las acciones a realizar por el usuario se realizarán con el ratón con la excepción de las que se requieren de la escritura de datos por teclado.

Los *dispositivos de salida* permiten representar los resultados (salida) del proceso de los datos. El dispositivo de salida típico es la **pantalla** (CRT)³ o **monitor**. Otros dispositivos de salida son: **impresoras** (imprimen resultados en papel), **trazadores gráficos** (*plotters*), **reconocedores de voz**, **altavoces**, etc.

El teclado y la pantalla constituyen —en muchas ocasiones— un Único dispositivo, denominado **terminal**. Un teclado de terminal es similar al teclado de una máquina de escribir moderna con la diferencia de algunas teclas extras que tiene el terminal para funciones especiales. Si está utilizando una computadora personal, el teclado y el monitor son dispositivos independientes conectados a la computadora por cables. En ocasiones a la impresora se la conoce como **dispositivo de copia dura** («*hard copy*»), debido a que la escritura en la impresora es una copia permanente (dura) de la salida, y a la pantalla se le denomina en contraste: **dispositivo de copia blanda** («*soft copy*»), ya que se pierde la pantalla actual cuando se visualiza la siguiente.

Los dispositivos de entrada/salida y los dispositivos de almacenamiento secundario o auxiliar (memoria externa) se conocen también con el nombre de *dispositivos periféricos* o simplemente **periféricos** ya que, normalmente, son externos a la computadora. Estos dispositivos son unidad de discos (disquetes, CD-ROM, DVDs, cintas, videocámaras, etc.).



Figura 1.4. Dispositivo de salida (impresora)

1.2.2. La memoria central (interna)

La **memoria central** o simplemente **memoria** (*interna o principal*) se utiliza para almacenar información (RAM, Random Access Memory). En general, la información almacenada en memoria puede ser de dos tipos: las *instrucciones* de un programa y los *datos* con los que operan las instrucciones. Por ejemplo, para que un programa se pueda *ejecutar* (correr, rodar, funcionar..., en inglés *run*), debe ser situado en la memoria central, en una operación denominada *carga* (*load*) del programa. Después, cuando se ejecuta (se realiza, funciona) el programa, *cualquier dato a procesar por el programa se debe llevar a la memoria* mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta con él a fin de poder funcionar.

³ Cathode Ray Tube: Tubo de rayos catódicos

Ejecución

Cuando un programa se ejecuta (realiza, funciona) en una computadora, se dice que se *ejecuta*.

Con el objetivo de que el procesador pueda obtener los datos de la memoria central más rápidamente, la mayoría de los procesadores actuales (muy rápidos) utilizan con frecuencia una *memoria* denominada *caché* que sirva para almacenamiento intermedio de datos entre el procesador y la memoria principal. La memoria caché —en la actualidad— se incorpora casi siempre al procesador.

La memoria central de una computadora *es* una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de *celdas de memoria* (estas celdas o posiciones de memoria se denominan también *palabras*, aunque no «guardan» analogía con las palabras del lenguaje). El número de celdas de memoria de la memoria central, dependiendo del tipo y modelo de computadora; hoy día el número suele ser millones (32.64, 128, etc.) Cada celda de memoria consta de un cierto número de bits (normalmente 8, un *byte*).

La unidad elemental de memoria se llama *byte* (octeto). Un *byte* tiene la capacidad de almacenar un carácter de información, y está formado por un conjunto de unidades más pequeñas de almacenamiento denominadas *bits*, que son dígitos binarios (0 o 1).

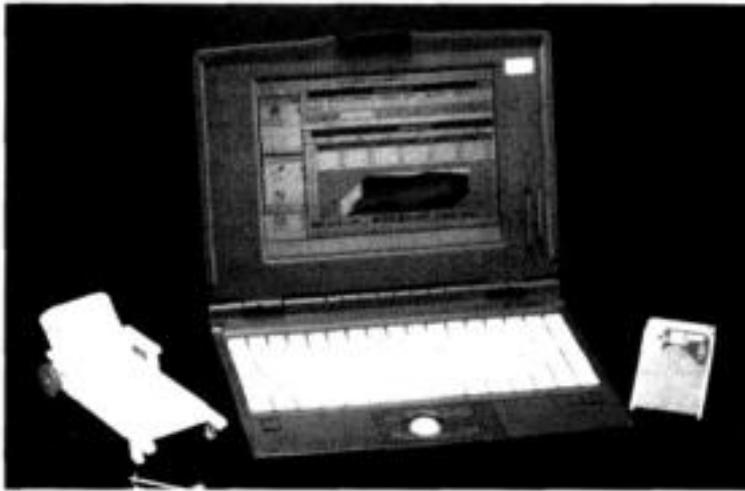


Figura 1.5. Computadora portátil digital.

Generalmente, se acepta que un byte contiene ocho bits. Por consiguiente, si se desea almacenar la frase

Hola Mortimer todo va bien

la computadora utilizará exactamente 27 bytes consecutivos de memoria. Obsérvese que, además de las letras, existen cuatro espacios en blanco y un punto (un espacio es un carácter que emplea también un byte). De modo similar, el número del pasaporte

1'5 487891

ocupará 9 bytes, pero si se almacena como

P5 748 7891

ocupará 11. Estos datos se llaman *alfanuméricos*, y pueden constar del alfabeto, dígitos o incluso caracteres especiales (símbolos: \$, #, *, etc.).

Mientras que cada carácter de un dato alfanumérico se almacena en un byte, la información numérica se almacena de un modo diferente. Los datos numéricos ocupan 2, 4 e incluso 8 bytes consecutivos, dependiendo del tipo de dato numérico (se verá en el Capítulo 3).

Existen dos conceptos importantes asociados a cada celda o posición de memoria: su dirección y su contenido. Cada celda o byte tiene asociada una única dirección que indica su posición relativa en memoria y mediante la cual se puede acceder a la posición para almacenar o recuperar información. La información almacenada en una posición de memoria es su contenido. La Figura 1.6 muestra una memoria de computadora que consta de 1.000 posiciones en memoria con direcciones de 0 a 999. El contenido de estas direcciones o posiciones de memoria se llaman palabras, de modo que existen palabras de 8, 16, 32 y 64 bits. **Por** consiguiente, si trabaja con una máquina de 32 bits, significa que en cada posición de memoria de su computadora puede alojar 32 bits, es decir 32 dígitos, bien ceros o unos.

Siempre que una nueva información se almacena en una posición, se destruye (desaparece) cualquier información que en ella hubiera y no se puede recuperar. La dirección es permanente y única, el contenido puede cambiar mientras se ejecuta un programa.

La memoria central de una computadora puede tener desde unos centenares de millares de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos para definir el tamaño de la memoria central: Kilo-byte (**KB o Kb**) igual a 1.024 bytes (2^{10}) —prácticamente se toman 1.000— y Megabyte (**MB o Mb**) igual a 1.024×1.024 bytes (2^{20}) —prácticamente se considera un 1.000.000—.

Tabla 1.1. Unidades de medida de almacenamiento.

Byte	Byte (b)	<i>equivale a</i>	8 bits
Kilobyte	Kbyte (Kb)	<i>equivale a</i>	1.24 bytes
Megabyte	Mbyte (Mb)	<i>equivale a</i>	1.024 Kbytes
Gigabyte	Gbyte (Gb)	<i>equivale a</i>	1.024 Mbytes
Terabyte	Tbyte (Tb)	<i>equivale a</i>	1.024 Gbytes

$$1 \text{ Tb} = 1.024 \text{ Gb} = 1.024 \text{ Mb} = 1.048.576 \text{ Kb} = 1.073.741.824 \text{ b}$$

En la actualidad, las computadoras personales tipo PC suelen tener memorias centrales de 32 a 64 Mb, aunque ya es muy frecuente ver PC con memorias de 128 Mb y 192 Mb.

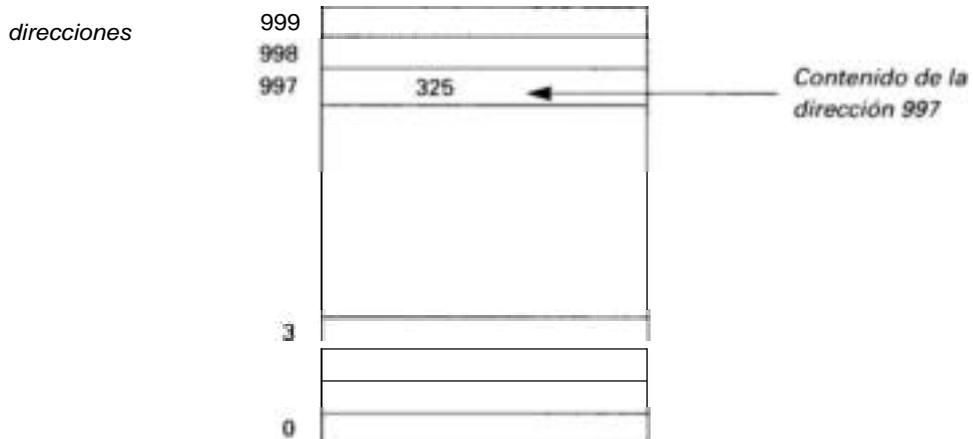


Figura 1.6. Memoria central de una computadora.

La memoria principal es la encargada de almacenar los programas y datos que se están ejecutando y su principal característica es que el acceso a los datos o instrucciones desde esta memoria es muy rápido.

En la memoria principal se almacenan:

- Los **datos enviados para procesarse desde los dispositivos de entrada.**
- **Los programas que realizarán los procesos.**
- **Los resultados obtenidos preparados para enviarse a un dispositivo de salida.**

En la memoria principal se pueden distinguir dos tipos de memoria: RAM y ROM. La memoria **RAM** (Random Access Memory, Memoria de acceso aleatorio) almacena los datos e instrucciones a procesar. Es un tipo de memoria volátil (su contenido se pierde cuando se apaga la computadora); esta memoria es, en realidad, la que se suele conocer como memoria principal o de trabajo; en esta memoria se pueden escribir datos y leer de ella. La memoria ROM (Read Only Memory) es una memoria permanente en la que no se puede escribir (viene pregrabada «grabada» por el fabricante; es una **memoria de sólo lectura**). Los programas almacenados en ROM no se pierden al apagar la computadora y cuando se enciende, se lee la información almacenada en esta memoria. Al ser esta memoria de sólo lectura, los programas almacenados en los chips ROM no se pueden modificar y suelen utilizarse para almacenar los programas básicos que sirven para arrancar la computadora.

1.2.3. La Unidad Central de Proceso (UCP)

La *Unidad Central de Proceso*, **UCP** (*Central Processing Unit*, **CPU**, en inglés), dirige y controla el proceso de información realizado por la computadora. La UCP procesa o manipula la información almacenada en memoria; puede recuperar información desde memoria (esta información son datos o instrucciones: programas). También puede almacenar los resultados de estos procesos en memoria para su uso posterior.

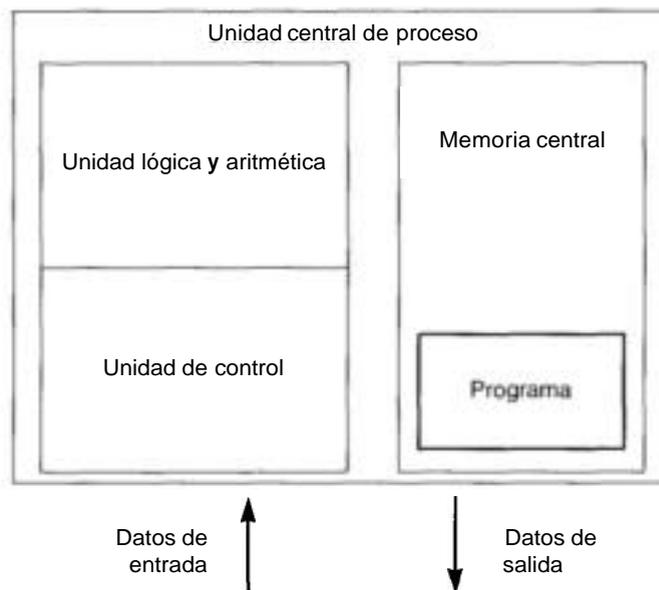


Figura 1.7. Unidad Central de Proceso.

La UCP consta de dos componentes: *unidad de control (UC)* y *unidad aritmético-lógica (UAL)* (Fig. 1.7). La **unidad de control** (*Control Unit, CU*) coordina las actividades de la computadora y determina qué operaciones se deben realizar y en qué orden; asimismo controla y sincroniza todo el proceso de la computadora.

La **unidad aritmético-lógica** (*Aritmetic-Logic Unit, ALU*) realiza operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones. Los datos en la memoria central se pueden *leer* (recuperar) o *escribir* (cambiar) por la UCP.

1.2.4. El microprocesador

El **microprocesador** es un *chip (un circuito integrado)* que controla y realiza las funciones y operaciones con los datos. Se suele conocer como *procesador* y es el cerebro y corazón de la computadora. En realidad el microprocesador representa a la Unidad Central de Proceso.

La velocidad de un microprocesador se mide en megahercios (**MHz**) y manipulan palabras de 4 a 64 bits. Los microprocesadores históricos van desde el 8080 hasta el 80486/80586 pasando por el 8086, 8088, 80286 y 80386, todos ellos del fabricante **Intel**. Existen otras empresas como **AMD** y **Cyrix**, con modelos similares. Los microprocesadores de segunda generación de Intel son los Pentium, Pentium MMX, Pentium II con velocidades de 233, 266, 300 y 450 MHz. Los microprocesadores más modernos (de 3.ª generación) son los Pentium III con frecuencias de 450 hasta 1 GHz.

La guerra de los microprocesadores se centró en el año 2000 en torno a **AMD**, que ofrecen ya procesadores **Athlon** de 1 GHz y de 1.2 GHz. Intel presentó a finales de noviembre de 2000 su nueva arquitectura **Pentium IV** —la generación siguiente a la familia x86—, que ofrecen *chips* de velocidades de 1.3, 1.4 y 1.5 GHz y anuncian velocidades de hasta 2 GHz.

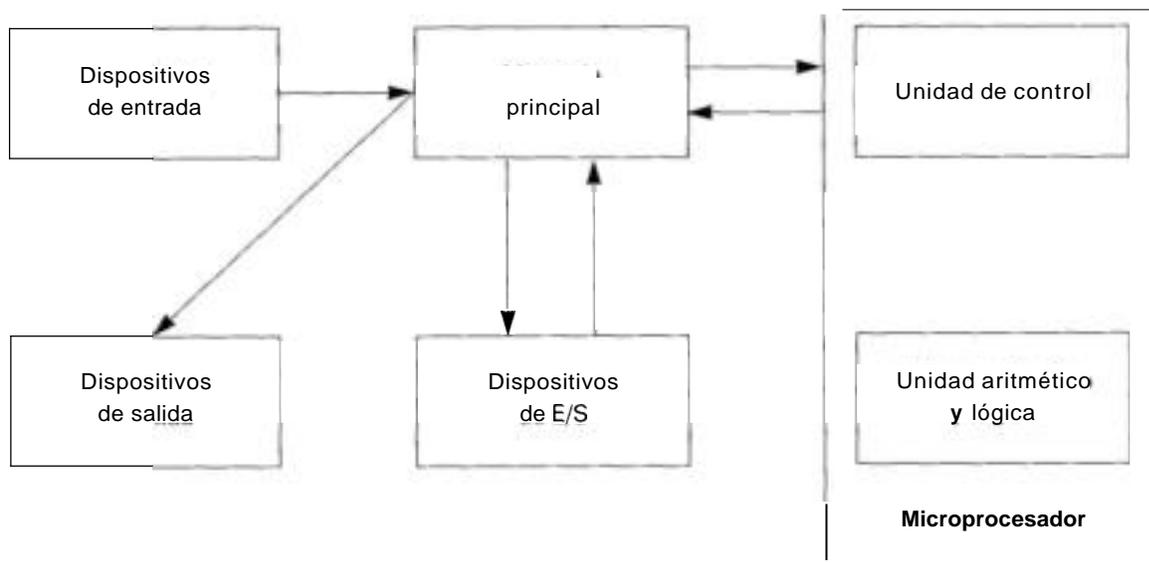


Figura 1.8. Organización física de una computadora con un microprocesador.

1.2.5. Memoria auxiliar (externa)

Cuando un programa se ejecuta, se debe situar primero en memoria central de igual modo que los datos. Sin embargo, la información almacenada en la memoria se pierde (borra) cuando se *apaga* (desconecta de la red eléctrica) la computadora y, por otra parte, la memoria central es limitada en capacidad. Por

esta razón, para poder disponer de almacenamiento permanente, tanto para programas como para datos, se necesitan *dispositivos de almacenamiento secundario, auxiliar o masivo* («mass storage», o «secondary storage»).

Los **dispositivos de almacenamiento o memorias auxiliares** (*externas o secundarias*) más comúnmente utilizados son: *cintas magnéticas, discos magnéticos, discos compactos (CD-ROM Compact Disk Read Only Memory)*, y videodiscos digitales (**DVD**). Las cintas son utilizadas principalmente por sistemas de computadoras grandes similares a las utilizadas en los equipos de audio. Los discos y disquetes magnéticos se utilizan por todas las computadoras, especialmente las medias y pequeñas —las computadoras personales—. Los discos pueden ser *duros*, de gran capacidad de almacenamiento (su capacidad mínima es de 10 Mb), **disquetes o discos flexibles** («floppy disk») (360 Kb a 1,44 Mb). El tamaño físico de los disquetes y por el que son conocidos es de 5¼ (5,25)", 3½ (3,5)". Las dos caras de los discos se utilizan para almacenar información. La capacidad de almacenamiento varía en función de la intensidad de su capa ferromagnética y pueden ser de doble densidad (DD) o de alta densidad (HD). El disquete normal suele ser de 3,5" y de 1,44 Mb de capacidad.



Figura 1.9. Memorias auxiliares: Unidad y lector ZIP de 100 Mb.

Otro dispositivo cada vez más utilizado en una computadora es el **CD-ROM** (*Compact Disk*) que es un disco de gran capacidad de almacenamiento (650 Mb) merced a la técnica utilizada que es el láser. El videodisco digital (**DVD**) es otro disco compacto de gran capacidad de almacenamiento (equivale a 26 CD-ROM) que por ahora es de 4,7 Gb.

Existen unos tipos de discos que se almacenan en unas unidades especiales denominadas *zip* que tienen gran capacidad de almacenamiento comparada con los disquetes tradicionales de 1.44 Mb. Estos disquetes son capaces de almacenar 100 Mb.

La información almacenada en la memoria central es *volátil* (desaparece cuando se apaga la computadora) y la información almacenada en la memoria auxiliar es *permanente*.

Esta información se organiza en unidades independientes llamadas **archivos (ficheros, file** en inglés). Los resultados de los programas se pueden guardar como *archivos de datos* y los programas que se escriben se guardan como *archivos de programas*, ambos en la memoria auxiliar. Cualquier tipo de archivo se puede transferir fácilmente desde la memoria auxiliar hasta la memoria central para su proceso posterior.

En el campo de las computadoras es frecuente utilizar la palabra memoria y almacenamiento o memoria externa, indistintamente. En este libro —y recomendamos su uso— se utilizará el término memoria sólo para referirse a la memoria central.

Comparación de la memoria central y la memoria auxiliar

La memoria central o principal es mucho más rápida y cara que la memoria auxiliar. Se deben transferir los datos desde la memoria auxiliar hasta la memoria central, antes de que puedan ser procesados. Los datos en memoria central son: *volátiles* y desaparecen cuando se *apaga* la computadora. Los datos en memoria auxiliar son *permanentes* y no desaparecen cuando se *apaga* la computadora.

Las computadoras modernas necesitan comunicarse con otras computadoras. Si la computadora se conecta con una *tarjeta de red* se puede conectar a una red de datos locales (*red de área local*). De este modo se puede acceder y compartir a cada una de las memorias de disco y otros dispositivos de entrada y salida. Si la computadora tiene un *módem*, se puede comunicar con computadoras distantes. Se pueden conectar a una red de datos o *enviar correo electrónico* a través de las redes corporativas Intranet/Extranet o la propia red Internet. También es posible enviar y recibir mensajes de fax.

1.2.6. Proceso de ejecución de un programa

La Figura 1.10 muestra la comunicación en una computadora cuando se ejecuta un programa, a través de los dispositivos de entrada y salida. El ratón y el teclado introducen datos en la memoria central cuando se ejecuta el programa. Los datos intermedios o auxiliares se transfieren desde la unidad de disco (archivo) a la pantalla y a la unidad de disco, a medida que se ejecuta el programa.

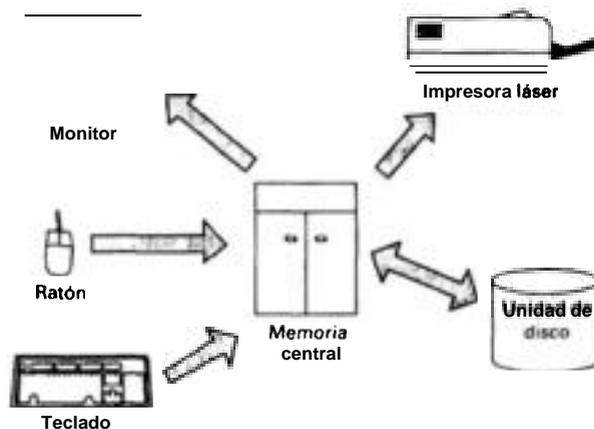


Figura 1.10. Proceso de ejecución de un programa.

1.2.7. Comunicaciones: módems, redes, telefonía RDSI y ADSL

Una de las posibilidades más interesantes de las computadoras es la comunicación entre ellas, cuando se encuentran en sitios separados físicamente y se encuentran enlazadas por vía telefónica. Estas computadoras se conectan en redes LAN (Red de Área Local) y WAN (Red de Área Ancha), aunque hoy día, las redes más implantadas son las redes que se conectan con tecnología Internet, y, por tanto, conexión a la red Internet. Estas redes son *Intranet* y *Extranet*, y se conocen como redes corporativas, ya que enlazan computadoras de los empleados con las empresas. Las instalaciones de las comunicaciones requieren de líneas telefónicas analógicas o digitales y de módems.

El módem es un dispositivo periférico que permite intercambiar información entre computadoras a través de una línea telefónica. El módem es un acrónimo de Modulador-Demodulador, y es un dispositivo que transforma las señales digitales de la computadora en señales eléctricas analógicas telefónicas y viceversa, con lo que es posible transmitir y recibir información a través de la línea telefónica. Estas operaciones se conocen como *modulación* (se transforman los datos digitales de la computadora para que puedan ser enviados por la línea telefónica como analógicos) y *demodulación* (transforman los datos analógicos recibidos mediante la línea telefónica en datos digitales para que puedan ser leídos por la computadora).

Un módem convierte señal analógica en digital y viceversa.

Los módems permiten, además de las conexiones entre computadoras, envío y recepción de fax, acceso a Internet, etc. Una de las características más importantes de un módem es la velocidad. Cifras usuales son **33,600 (33 K)** baudios (1 baudio es 1 bit por segundo, *bps*) y **56,000** baudios (**56 K**).

Los módems pueden ser de tres tipos: interno (es una tarjeta que se conecta a la placa base internamente); externo (es un dispositivo que se conecta externamente a la computadora a través de puertos COM, USB, etc.); PC-Card, son módems del tipo tarjeta de crédito, que sirve para conexión a las computadoras portátiles.

Además de los módems analógicos, es posible la conexión a Internet y a las redes corporativas de las compañías mediante la Red Digital de Sistemas Integrados (RDSI, en inglés, IDSN), que permite la conexión a 128 Kbps, disponiendo de dos líneas telefónicas, cada una de ellas a 64 Kbps.

También, se está comenzando a implantar la tecnología digital **ADSL**, que permite la conexión a Internet a velocidad similar a la red RDSI, **128 Kbps** y a **256 Kbps**, según sea para «subir» o «bajar» datos a la red, respectivamente, pudiendo llegar a **2M bps**.

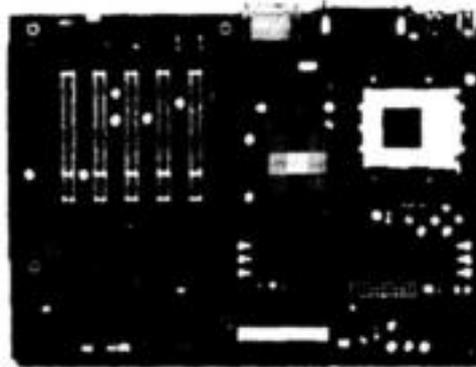


Figura 1.11. Módem comercial.

1.2.8. La computadora personal multimedia ideal para la programación

Hoy día, las computadoras personales profesionales y domésticas que se comercializan son prácticamente todas ellas multimedia, es decir, incorporan características *multimedia* (CD-ROM, DVD, tarjeta de sonido, altavoces y micrófono) que permiten integrar texto, sonido, gráficos e imágenes en movimiento. Las computadoras multimedia pueden leer discos CD-ROM y DVD de gran capacidad de almacenamiento. Esta característica ha hecho que la mayoría de los fabricantes de software comercialicen sus compiladores (programas de traducción de lenguajes de programación) en CD-ROM, almacenando en un solo disco, lo que antes necesitaba seis, ocho o doce disquetes, y cada vez será más frecuente el uso del DVD.



Figura 1.12. Computadora multimedia.

El estudiante de informática o de computación actual, y mucho más el profesional, dispone de un amplio abanico de computadoras a precios asequibles y con prestaciones altas. En el cuarto trimestre del año 2000, un PC de escritorio típico para aprender a programar, y posteriormente utilizar de modo profesional, es posible encontrarlo a precios en el rango entre 100.000 pesetas y 200.000/300.000 pesetas (US\$ 500 a US\$ 1.000/1.500), dependiendo de prestaciones y fabricante (según sean «clónicos» o fabricados por marcas acreditadas como HP, IBM, Compaq), aunque la mayoría de las ofertas suelen incluir, como mínimo, 64 MB de RAM, CD-ROM, monitores de 15'', tarjetas de sonido, etc. La Tabla 1.2 resume nuestra propuesta y recomendación de características medias de un/a computador/a PC.

Tabla 1.2. Características de un PC ideal.

Procesador	Microprocesador de las marcas Intel o AMD, de 800 Mz o superior.
Memoria	128 Mb y recomendable para aplicaciones profesionales 256 o 512 Mb.
Caché	Memoria especial que usa el procesador para acelerar sus operaciones. 512 Kb o 128 Kb.
Disco duro	20 Gigabytes (mínimo).
Internet	Preparado para Internet (incluso con módem instalado de 56 Kb).
Vídeo	Memoria de vídeo, con un mínimo de 4 Mb.
Monitor	17" o 19" (pantalla tradicional o plana "TFT").
Almacenamiento	CD-RW, DVD.
Puertos	Serie, paralelo y USB.
Marcas	HP, Compaq, Dell, IBM, El System, Fujitsu, Inves, ..

1.3. CONCEPTO DE ALGORITMO

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo este libro nos referiremos a la *metodología necesaria para resolver problemas mediante programas*, concepto que se denomina **metodología de la programación**. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, **algoritmo** proviene de *Mohammed al-KhoWârizmi*, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra *algorismus* derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV a.C.) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth —inventor de Pascal, Modula-2 y Oberon— tituló uno de sus más famosos libros, *Algoritmos + Estructuras de datos = Programas*, significándonos que sólo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

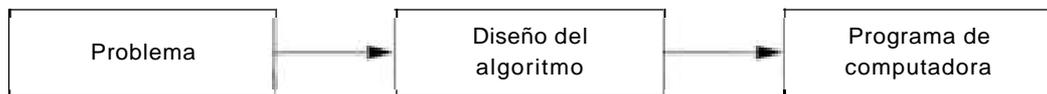


Figura 1.13. Resolución de un problema.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo* que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo.*)
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación.*)
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será *el diseño de algoritmos*. A la enseñanza y práctica de esta tarea se dedica gran parte de este libro.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, *la solución de un problema se puede expresar mediante un algoritmo*.

1.3.1. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada*, *Proceso* y *Salida*. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: ingredientes y utensilios empleados.

Proceso: elaboración de la receta en la cocina.

Salida: terminación del plato (por ejemplo, cordero).

Ejemplo 1.1

Un cliente ejecuta un pedido u una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, rechazará el pedido. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

1. Inicio.
 2. Leer el pedido.
 3. Examinar la ficha del cliente.
 4. Si el cliente es solvente, aceptar pedido; en caso contrario, rechazar pedido.
 5. Fin.
-

Ejemplo 1.2

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etc.

1. Inicio.
2. Poner X igual a 2 ($X = 2$, X variable que representa a los divisores del número que se busca N).
3. Dividir N por X (N/X).
4. Si el resultado de N/X es entero, entonces N no es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a X ($X \leftarrow X + 1$).

6. Si X es igual a N , entonces N es un número primo; en caso contrario, bifurcar al punto 3.
7. Fin.

Por ejemplo, si N es 131, los pasos anteriores serían:

1. Inicio.
2. $X = 2$.
3. $131/X$. Como el resultado no es entero, se continúa el proceso.
5. $X \leftarrow 2 + 1$, luego $X = 3$.
6. Como X no es 131, se bifurca al punto 3.
3. $131/X$ resultado no es entero.
5. $X \leftarrow 3 + 1$, $X = 4$.
6. Como X no es 131 bifurca al punto 3.
3. $131/X...$, etc.
7. Fin.

Ejemplo 1.3

Realizar la suma de todos los números pares entre 2 y 1000.

El problema consiste en sumar $2 + 4 + 6 + 8 \dots + 1000$. Utilizaremos las palabras SUMA y NUMERO (*variables*, serán denominadas más tarde) para representar las sumas sucesivas $(2+4)$, $(2+4+6)$, $(2+4+6+8)$, etc. La solución se puede escribir con el siguiente algoritmo:

1. Inicio.
2. Establecer SUMA a 0.
3. Establecer NUMERO a 2.
4. Sumar NUMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
5. Incrementar NUMERO en 2 unidades.
6. Si NUMERO \leq 1000 bifurcar al paso 4 ; en caso contrario, escribir el ultimo valor de SUMA y terminar el proceso.
7. Fin.

1.4. EL SOFTWARE (LOS PROGRAMAS)

Las operaciones que debe realizar el *hardware* son especificadas por una lista de instrucciones, llamadas programas, o *software*. El software se divide en dos grandes grupos: *software del sistema* y *software de aplicaciones*.

El **software del sistema** es el conjunto de programas indispensables para que la máquina funcione; se denominan también *programas del sistema*. Estos programas son, básicamente, *el sistema operativo*, *los editores de texto*, *los compiladores/intérpretes* (lenguajes de programación) y *los programas de utilidad*.

Uno de los programas más importante es el **sistema operativo**, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas. El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser: *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios), atendiendo al número de usuarios y *monocarga* (una sola tarea) o *multitarea* (múltiples tareas) según las tareas (procesos) que puede realizar simultáneamente. C corre prácticamente en todos los sistemas operativos, Windows 95, Windows NT, Windows 2000, UNIX, Lynx... , y en casi todas las computadoras personales actuales PC, Mac, Sun, etc.

Los *lenguajes de programación* sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados *traductores (compiladores o intérpretes)* convier-



Figura 1.14. Diferentes programas de software.

ten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1, *bits*) que ésta pueda entender.

Los *programas de utilidad*⁴ facilitan el uso de la computadora. Un buen ejemplo es un *editor de textos* que permite la escritura y edición de documentos. Este libro ha sido escrito en un editor de textos o *procesador de palabras* («*word procesor*»).

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etc. es decir, los programas que podrá escribir en C, se denominan *programas de aplicación*. A lo largo del libro se verán pequeños programas de aplicación que muestran los principios de una buena programación de computadora.

Se debe diferenciar entre el acto de crear un programa y la acción de la computadora cuando ejecuta las instrucciones del programa. La creación de un programa se hace inicialmente en papel y a continuación se introduce en la computadora y se convierte en lenguaje entendible por la computadora.

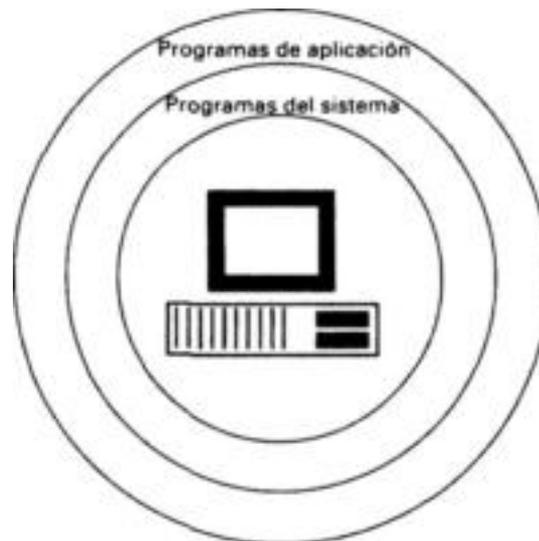


Figura 1.15. Relación entre programas de aplicación y programas del sistema.

⁴ *Utility*: programa de utilidad

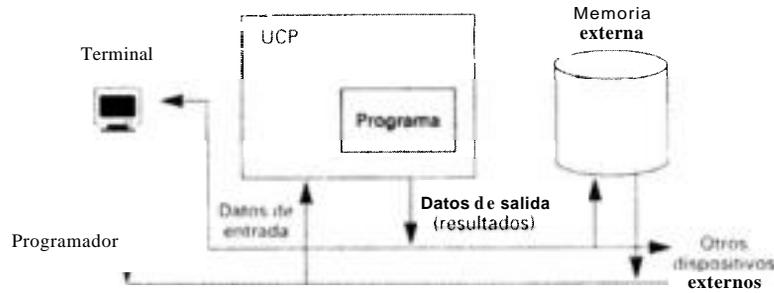


Figura 1.16. Acción de un programador

La Figura 1.16 muestra el proceso general de ejecución de un programa: aplicación de una entrada (*datos*) al programa y obtención de una salida (*resultados*). La entrada puede tener una variedad de formas, tales como números o caracteres alfabéticos. La salida puede también tener formas, tales como datos numéricos o caracteres, señales para controlar equipos o robots, etc.

La ejecución de un programa requiere —generalmente— unos datos como entrada (Fig. 1.17), además del propio programa, para poder producir una salida.

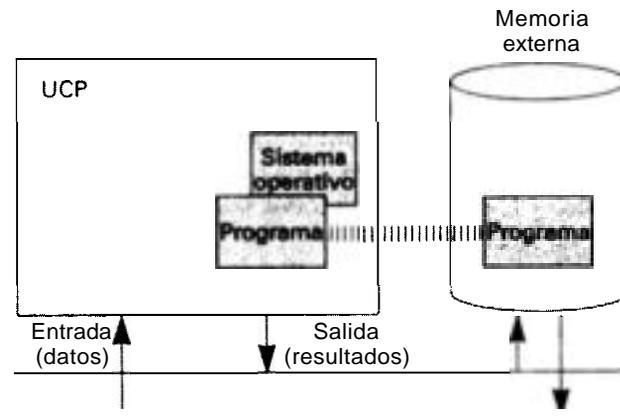


Figura 1.17. Ejecución de un programa

I.5. LOS LENGUAJES DE PROGRAMACIÓN

Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de *interpretar* el algoritmo, lo que significa:

- Comprender las instrucciones de cada paso.
- Realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina *programa*. Un programa se escribe en un *lenguaje de programación* y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman *programación*. Así pues, los lenguajes utilizados para escribir programas de computadoras son los *lenguajes de programación* y *programadores* son los escritores y diseñadores de programas.

Los principales tipos de lenguajes utilizados en la actualidad son tres:

- *Lenguajes máquina.*
- *Lenguaje de bajo nivel (ensamblador).*
- *Lenguajes de alto nivel.*

1.5.1. Instrucciones a la computadora

Los diferentes pasos (*acciones*) de un algoritmo se expresan en los programas como *instrucciones, sentencias o proposiciones* (normalmente el término instrucción se suele referir a los lenguajes máquina y bajo nivel, reservando la sentencia o proposición para los lenguajes de alto nivel). Por consiguiente, un programa consta de una secuencia de instrucciones, cada una de las cuales especifica ciertas operaciones que debe ejecutar la computadora.

La elaboración de un programa requerirá conocer el juego o repertorio de instrucciones del lenguaje. Aunque en el Capítulo 3 se analizarán con más detalle las instrucciones, adelantaremos los tipos fundamentales de instrucciones que una computadora es capaz de manipular y ejecutar. Las instrucciones básicas y comunes a casi todos los lenguajes de programación se pueden condensar en cuatro grupos:

- *Instrucciones de entrada/salida.* Instrucciones de transferencia de información y datos entre dispositivos periféricos (teclado, impresora, unidad de disco, etc.) y la memoria central.
- *Instrucciones aritmético-lógicas.* Instrucciones que ejecutan operaciones aritméticas (suma, resta, multiplicación, división, potenciación), lógicas (operaciones *and*, *or*, *not*, etc.).
- *Instrucciones selectivas.* Instrucciones que permiten la selección de tareas alternativas en función de los resultados de diferentes expresiones condicionales.
- *Instrucciones repetitivas.* Instrucciones que permiten la repetición de secuencias de instrucciones un número determinado de veces.

1.5.2. Lenguajes máquina

Los **lenguajes máquina** son aquellos que están escritos en lenguajes directamente inteligibles por la máquina (computadora), ya que sus instrucciones son *cadena binarias* (cadenas o series de caracteres —dígitos— 0 y 1) que especifican una operación, y las posiciones (dirección) de memoria implicadas en la operación se denominan *instrucciones de máquina o código máquina*. El código máquina es el conocido código binario.

Las instrucciones en lenguaje máquina dependen del *hardware* de la computadora y, por tanto, diferirán de una computadora a otra. El lenguaje máquina de un PC (computadora personal) será diferente de un sistema HP (Hewlett Packard), Compaq o un sistema de IBM.

Las *ventajas* de programar en lenguaje máquina son la posibilidad de cargar (transferir un programa a la memoria) sin necesidad de traducción posterior, lo que supone una velocidad de ejecución superior a cualquier otro lenguaje de programación.

Los *inconvenientes* —en la actualidad— superan a las ventajas, lo que hace prácticamente no recomendables los lenguajes máquina. Estos inconvenientes son:

- Dificultad y lentitud en la codificación.
- Poca fiabilidad.
- Dificultad grande de verificar y poner a punto los programas.
- Los programas sólo son ejecutables en el mismo procesador (UPC, *Unidad Central de Proceso*).

Para evitar los lenguajes máquina, desde el punto de vista del usuario, se han creado otros lenguajes que permiten escribir programas con instrucciones similares al lenguaje humano (por desgracia casi siempre inglés, aunque existen raras excepciones, como es el caso de las versiones españolas del lenguaje LOGO). Estos lenguajes se denominan de *alto y bajo nivel*.

1.5.3. Lenguajes de bajo nivel

Los **lenguajes de bajo nivel** son más fáciles de utilizar que los lenguajes máquina, pero, al igual, que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el *ensamblador* (*assembly language*). Las instrucciones en lenguaje ensamblador son instrucciones conocidas como **nemotécnicos** (*mnemonics*). Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés, ADD, SUB, DIV, etc.; en español, SUM, RES, DIV, etc.

Una instrucción típica de suma sería:

```
ADD M, N, P
```

Esta instrucción podía significar «*sumar el número contenido en la posición de memoria M al número almacenado en la posición de memoria N y situar el resultado en la posición de memoria P*». Evidentemente, es mucho más sencillo recordar la instrucción anterior con un nemotécnico que su equivalente en código máquina:

```
0110 1001 1010 1011
```

Un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente por la computadora —en esto se diferencia esencialmente del lenguaje máquina—, sino que requiere una fase de *traducción* al lenguaje máquina.

El programa original escrito en lenguaje ensamblador se denomina *programa fuente* y el programa traducido en lenguaje máquina se conoce como *programa objeto*, ya directamente inteligible por la computadora.

El traductor de programas fuente a objeto es un programa llamado *ensamblador* (*assembler*), existente en casi todas las computadoras (Fig. 1.18).

No se debe confundir —aunque en español adoptan el mismo nombre— *el programa ensamblador* (*assembler*), encargado de efectuar la traducción del programa fuente escrito a lenguaje máquina, con el *lenguaje ensamblador* (*assembly language*), lenguaje de programación con una estructura y gramática definidas.

Los lenguajes ensambladores presentan la *ventaja* frente a los lenguajes máquina de su mayor facilidad de codificación y, en general, su velocidad de cálculo.



Figura 1.18. Programa ensamblador.

Los *inconvenientes* más notables de los lenguajes ensambladores son:

- Dependencia total de la máquina, lo que impide la transportabilidad de los programas (posibilidad de ejecutar un programa en diferentes máquinas). El lenguaje ensamblador del PC es distinto del lenguaje ensamblador del Apple Macintosh.
- La formación de los programas es más compleja que la correspondiente a los programadores de alto nivel, ya que exige no sólo las técnicas de programación, sino también el conocimiento del interior de la máquina.

Hoy día los lenguajes ensambladores tiene sus aplicaciones muy reducidas en la programación de aplicaciones y se centran en aplicaciones de tiempo real, control de procesos y de dispositivos electrónicos, etc.

1.5.4. Lenguajes de alto nivel

Los *lenguajes de alto nivel* son los más utilizados por los programadores. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensambladores. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la máquina; esto es, las instrucciones del programa de la computadora no dependen del diseño del *hardware* o de una computadora en particular. En consecuencia, los programas escritos en lenguaje de alto nivel son *portables* o *transportables*, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras; al contrario que los programas en lenguaje máquina o ensamblador, que sólo se pueden ejecutar en un determinado tipo de computadora.

Los lenguajes de alto nivel presentan las siguientes *ventajas*:

- El tiempo de formación de los programadores es relativamente corto comparado con otros lenguajes.
- La escritura de programas se basa en reglas sintácticas similares a los lenguajes humanos. Nombres de las instrucciones, tales como READ, WRITE, PRINT, OPEN, etc.
- Las modificaciones y puestas a punto de los programas son más fáciles.
- Reducción del coste de los programas.
- Transportabilidad.

Los *inconvenientes* se concretan en:

- Incremento del tiempo de puesta a punto, al necesitarse diferentes traducciones del programa fuente para conseguir el programa definitivo.
- No se aprovechan los recursos internos de la máquina, que se explotan mucho mejor en lenguajes máquina y ensambladores.
- Aumento de la ocupación de memoria.
- El tiempo de ejecución de los programas es mucho mayor.

Al igual que sucede con los lenguajes ensambladores, los programas fuente tienen que ser traducidos por los programas traductores, llamados en este caso *compiladores* e *intérpretes*.

Los lenguajes de programación de alto nivel existentes hoy son muy numerosos aunque la práctica demuestra que su uso mayoritario se reduce a

C C++ # COBOL FORTRAN Pascal Visual BASIC Java

están muy extendidos:

Ada-95 Modula-2 Prolog LISP Smalltalk Eiffel

son de gran uso en el mundo profesional:

Borland Delphi C++ Builder Power Builder

Aunque hoy día el mundo Internet consume gran cantidad de recursos en forma de lenguajes de programación tales como **HTML, XML, JavaScript, ...**

1.5.5. Traductores de lenguaje

Los *traductores de lenguaje* son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina.

Los traductores se dividen en:

- *Intérpretes.*
- *Compiladores.*

7.5.5.7. Intérpretes

Un *intérprete* es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta. Los programas intérpretes clásicos como BASIC, prácticamente ya no se utilizan, aunque las versiones Qbasic y QuickBASIC todavía se pueden encontrar y corren en las computadoras personales. Sin embargo, está muy extendida la versión interpretada del lenguaje Smalltalk, un lenguaje orientado a objetos puro.

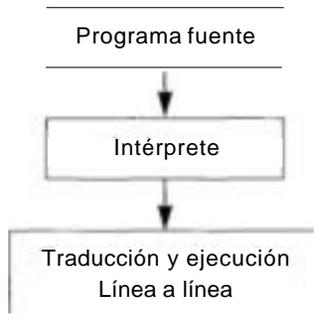


Figura 1.19. Intérprete.

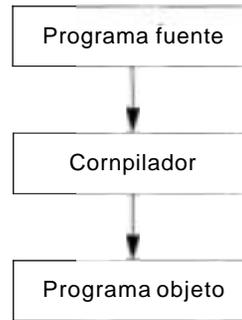


Figura 1.20. La compilación de programas.

1.5.5.2. Compiladores

Un *compilador* es un programa que traduce los programas fuente escritos en lenguaje de alto nivel —C, FORTRAN...— a lenguaje máquina.

Los programas escritos en lenguaje de alto nivel se llaman *programas fuente* y el programa traducido *programa objeto* o *código objeto*. El compilador traduce —sentencia a sentencia— el programa fuente. Los lenguajes compiladores típicos son: C, C++, Pascal, Java y COBOL.

1.5.6. La compilación y sus fases

La *compilación* es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina.

Para conseguir el programa máquina real se debe utilizar un programa llamado *montador* o *enlazador* (*linker*). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable (Fig. 1.21).

El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos:

1. Escritura del *programa fuente* con un *editor* (programa que permite a una computadora actuar de modo similar a una máquina de escribir electrónica) y guardarlo en un dispositivo de almacenamiento (por ejemplo, un disco).

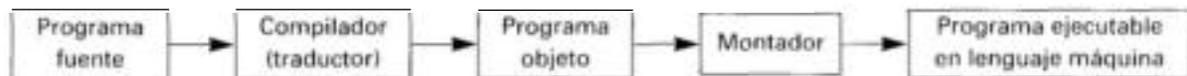


Figura 1.21. Fases de la compilación.

2. Introducir el programa fuente en memoria.
3. *Compilar* el programa con el compilador C.
4. *Verificar y corregir errores de compilación* (listado de errores).
5. Obtención del programa *objeto*.
6. El enlazador (*linker*) obtiene el *programa ejecutable*.
7. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programa.

El proceso de ejecución sería el mostrado en las Figuras 1.22 y 1.23. En el Capítulo 3 se describirá en detalle el proceso completo y específico de ejecución de programas en lenguaje C.

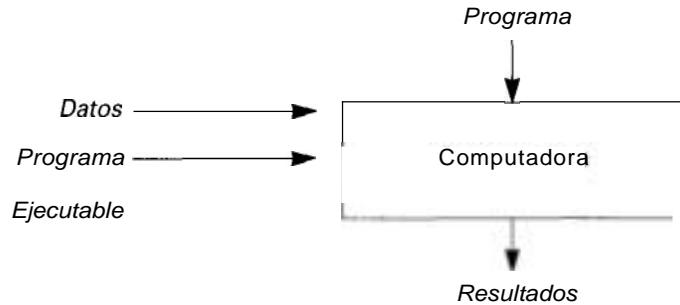


Figura 1.22. Ejecución de un programa.

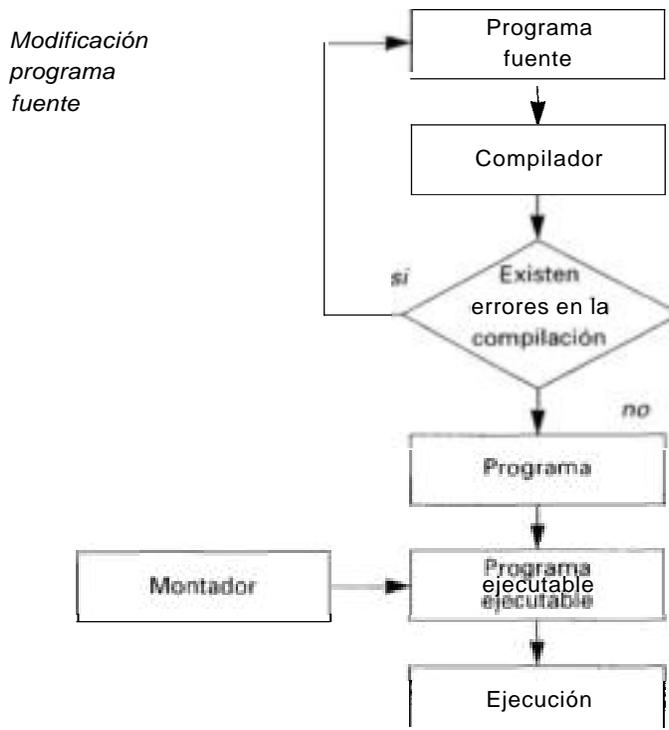


Figura 1.23. Fases de ejecución de un programa.



1.6. EL LENGUAJE C: HISTORIA Y CARACTERÍSTICAS

C es el lenguaje de programación de propósito general asociado, de modo universal, al sistema operativo UNIX. Sin embargo, la popularidad, eficacia y potencia de C, se ha producido porque este lenguaje no está prácticamente asociado a ningún sistema operativo, ni a ninguna máquina, en especial. Ésta es la razón fundamental, por la cual C, es conocido como el *lenguaje de programación de sistemas, por excelencia*.

C es una evolución de los lenguajes BCPL — desarrollado por Martin Richards— y B — desarrollado por Ken Thompson en 1970— para el primitivo UNIX de la computadora DEC PDP-7.

C nació realmente en 1978, con la publicación de The C Programming Language, por Brian Kernighan y Dennis Ritchie (Prentice Hall, 1978). Desde su nacimiento, C fue creciendo en popularidad y los sucesivos cambios en el lenguaje a lo largo de los años junto a la creación de compiladores por grupos no involucrados en su diseño, hicieron necesario pensar en la estandarización de la definición del lenguaje C.

Así, en 1983, el American National Estándar Institute (ANSI), una organización internacional de estandarización, creó un comité (el denominado X3J11) cuya tarea fundamental consistía en hacer «una definición no ambigua del lenguaje C, e independiente de la máquina». Había nacido el estándar ANSI del lenguaje C. Con esta definición de C se asegura que cualquier fabricante de software que vende un compilador ANSI C incorpora todas las características del lenguaje, especificadas por el estándar. Esto significa también que los programadores que escriban programas en C estándar tendrán la seguridad de que correrán sus modificaciones en cualquier sistema que tenga un compilador C.

C es un *lenguaje de alto nivel*, que permite programar con instrucciones de lenguaje de propósito general. También, C se define como un lenguaje de programación estructurado de propósito general; aunque en su diseño también primó el hecho de que fuera especificado como un lenguaje de programación de Sistemas, lo que proporciona una enorme cantidad de potencia y flexibilidad.

El estándar ANSI C formaliza construcciones no propuestas en la primera versión de C, en especial, asignación de estructuras y enumeraciones. Entre otras aportaciones, se definió esencialmente, una nueva forma de declaración de funciones (prototipos). Pero, es esencialmente la biblioteca estándar de funciones, otra de las grandes aportaciones.

Hoy, en el siglo XXI, C sigue siendo uno de los lenguajes de programación más utilizados en la industria del software, así como en institutos tecnológicos, escuelas de ingeniería y universidades. Prácticamente todos los fabricantes de sistemas operativos, UNIX, LINUX, MacOS, SOLARIS, ... soportan diferentes tipos de compiladores de lenguaje C.

1.6.1. Ventajas de C

El lenguaje C tiene una gran cantidad de ventajas sobre otros lenguajes, y son, precisamente la razón fundamental de que después de casi dos décadas de uso, C siga siendo uno de los lenguajes más populares y utilizados en empresas, organizaciones y fábricas de software de todo el mundo.

Algunas ventajas que justifican el uso todavía creciente del lenguaje C en la programación de computadoras son:

- El lenguaje C es poderoso y flexible, con órdenes, operaciones y funciones de biblioteca que se pueden utilizar para escribir la mayoría de los programas que corren en la computadora.
- C se utiliza por programadores profesionales para desarrollar software en la mayoría de los modernos sistemas de computadora.
- Se puede utilizar C para desarrollar sistemas operativos, compiladores, sistemas de tiempo real y aplicaciones de comunicaciones.
- Un programa C puede ser escrito para un tipo de computadora y trasladarse a otra computadora con pocas o ninguna modificación — propiedad conocida como portabilidad —. El hecho de que C sea portable es importante ya que la mayoría de los modernos computadores tienen un compi-

lador C, una vez que se aprende C no tiene que aprenderse un nuevo lenguaje cuando se escriba un programa para otro tipo de computadora. No es necesario reescribir un problema para ejecutarse en otra computadora.

C se caracteriza por su velocidad de ejecución. En los primeros días de la informática, los problemas de tiempo de ejecución se resolvían escribiendo todo o parte de una aplicación en lenguaje ensamblador (lenguaje muy cercano al lenguaje máquina).

Debido a que existen muchos programas escritos en C, se han creado numerosas bibliotecas C para programadores profesionales que soportan gran variedad de aplicaciones. Existen bibliotecas del lenguaje C que soportan aplicaciones de bases de datos, gráficos, edición de texto, comunicaciones, etc.

1.6.2. Características técnicas de C

Hay numerosas características que diferencian a C de otros lenguajes y lo hacen eficiente y potente a la vez.

- Una nueva sintaxis para declarar funciones. Una declaración de función puede añadir una descripción de los argumentos de la función. Esta información adicional sirve para que los compiladores detecten más fácilmente los errores causados por argumentos que no coinciden.
- Asignación de estructuras (registros) y enumeraciones.
- Preprocesador más sofisticado.
- Una nueva definición de la biblioteca que acompaña a C. Entre otras funciones se incluyen: acceso al sistema operativo (por ejemplo, lectura y escritura de archivos), entrada y salida con formato, asignación dinámica de memoria, manejo de cadenas de caracteres.
- Una colección de cabeceras estándar que proporciona acceso uniforme a las declaraciones de funciones y tipos de datos.

1.6.3. Versiones actuales de C

En la actualidad son muchos los fabricantes de compiladores C, aunque los más populares entre los fabricantes de software son: Microsoft, Imprise, etc.

Una evolución de C, el lenguaje C++ (C con clases) que contiene entre otras, todas las características de ANSI C. Los compiladores más empleados Visual C++ de Microsoft. Builder C++ de Imprise-antigua Borland, C++ bajo UNIX y LINUX.

En el verano del 2000, Microsoft patentó una nueva versión de C++, que es C#, una evolución del C++ estándar, con propiedades de Java y diseñado para aplicaciones en línea, *Internet (on line)* y fuera de línea.

1.7. RESUMEN

Una computadora **es** una máquina para procesar información y obtener resultados en función de unos datos de entrada.

Hardware: parte física de una computadora (dispositivos electrónicos).

Software: parte lógica de una computadora (programas).

Las computadoras se componen de:

- Dispositivos de Entrada/Salida (E/S).
- Unidad Central de Proceso (Unidad de Control y Unidad Lógica y Aritmética).
- Memoria central.
- Dispositivos de almacenamiento masivo de información (memoria auxiliar o externa).

El *software del sistema* comprende, entre otros, el sistema operativo **MSDOS, UNIX, Linux...** en

computadoras personales y los lenguajes de programación.

Los lenguajes de programación se clasifican en:

- **alto nivel:** Pascal, FORTRAN, VISUAL, BASIC, C, Ada, Modula-2, C++, Java, Delphi, C, etc.
- **bajo nivel:** Ensamblador.
- **máquina:** Código máquina.

Los programas traductores de lenguajes son:

- **compiladores.**
- **intérpretes.**

C es un lenguaje de programación que contiene excelentes características como lenguaje para aprendizaje de programación y lenguaje profesional de propósito general; básicamente es un *entorno de programación* con editor y compilador incorporado.

CAPITULO 2

FUNDAMENTOS DE PROGRAMACIÓN

CONTENIDO

- 2.1. Fases en la resolución de problemas.
- 2.2. Programación modular.
- 2.3. Programación estructurada.
- 2.4. Representación gráfica de algoritmos.
- 2.6. Diagrama de *Nassi Schneiderman*.
- 2.6. El ciclo de vida del *software*.
- 2.7. Métodos formales de verificación de programas.
- 2.8. Factores de calidad del *software*.
- 2.9. *Resumen*.
- 2.10. *Ejercicios*.
- 2.11. *Ejercicios resueltos*.

INTRODUCCIÓN

Este capítulo le introduce a la metodología a seguir para la resolución de problemas con computadoras y con un lenguaje de programación como C.

La resolución de un problema con una computadora se hace escribiendo un programa, que exige al menos los siguientes pasos:

1. Definición o análisis del problema.
2. Diseño del algoritmo.
3. Transformación del algoritmo en un programa.
4. Ejecución y validación del programa.

Uno de los objetivos fundamentales de este libro es el aprendizaje y diseño de los *algoritmos*. Este capítulo introduce al lector en el concepto de algoritmo y de programa, **así** como las herramientas que permiten «dialogar» al usuario con la máquina: los lenguajes de programación.

CONCEPTUS CLAVE

- Algoritmo.
- Ciclo de vida.
- Diagrama *Nassi Schneiderman*.
- Diagramas de flujo.
- Métodos formales.
- *Postcondiciones*.
- *Precondiciones*.
- Programación modular.
- Diseño.
- Programación estructurada.
- Diseño descendente.
- Pruebas,
- Dominio del problema.
- *Pseudocódigo*.
- Factores de calidad.
- invariantes.
- Verificación.

2.1. FASES EN LA RESOLUCIÓN DE PROBLEMAS

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es —esencialmente— un proceso creativo, se puede considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores.

Las fases de resolución de un problema con computadora son:

- *Análisis del problema.*
- *Diseño del algoritmo.*
- *Codificación.*
- *Compilación y ejecución.*
- *Verificación.*
- *Depuración.*
- *Mantenimiento.*
- *Documentación.*

Constituyen el ciclo de vida del software y las fases o etapas usuales son:

- **Análisis.** El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.
- **Diseño.** Una vez analizado el problema, se diseña una solución que conducirá a un *algoritmo* que resuelva el problema.
- **Codificación (implementación).** La solución se escribe en la sintaxis del lenguaje de alto nivel (por ejemplo, C) y se obtiene un programa.
- **Ejecución, verificación y depuración.** El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados «*bugs*», en inglés) que puedan aparecer.
- **Mantenimiento.** El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.
- **Documentación.** Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera etapa (*codificación*) se *implementa* el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

La fase de *compilación y ejecución* traduce y ejecuta el programa. En las fases de *verificación y depuración* el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la *documentación del programa*.

Antes de conocer las tareas a realizar en cada fase, vamos a considerar el concepto y significado de la palabra **algoritmo**. La palabra *algoritmo* se deriva de la traducción al latín de la palabra Alkhôwarîzmi¹, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un **algoritmo** es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

¹ En la última edición (21.ª) del **DRAE** (Diccionario de la Real Academia Española) se ha aceptado el término *implementar*: (Informática) «Poner en funcionamiento, aplicar métodos, medidas, etc. para llevar algo a cabo».

² Escribió un tratado matemático famoso sobre manipulación de números y ecuaciones titulado *Kitab al-jabr w'almugabala*. La palabra álgebra se derivó, por su semejanza sonora, de *al-jabr*.

Características de un algoritmo

- *preciso* (indicar el orden de realización en cada *paso*),
- *definido* (si se sigue dos veces, obtiene **el mismo** resultado cada vez),
- *finito* (tiene **fin**; un número determinado de pasos).

Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan *métodos algorítmicos*, en oposición a los métodos que implican algún juicio o interpretación que se denominan *métodos heurísticos*. Los métodos algorítmicos se pueden *implementar* en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la *implementación* del proceso heurístico en computadoras.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etc. Los algoritmos se pueden expresar por *fórmulas*, *diagramas de flujo* o **N-S** y *pseudocódigos*. Esta última representación es la más utilizada en lenguajes estructurados como C.

2.1.1. Análisis del problema

La primera fase de la resolución de un problema con computadora es el *análisis del problema*. Esta fase requiere una clara definición, donde se contemple exactamente lo que debe hacer el programa y el resultado o solución deseada.

Dado que se busca una solución por computadora, se precisan especificaciones detalladas de entrada y salida. La Figura 2.1 muestra los requisitos que se deben definir en el análisis.

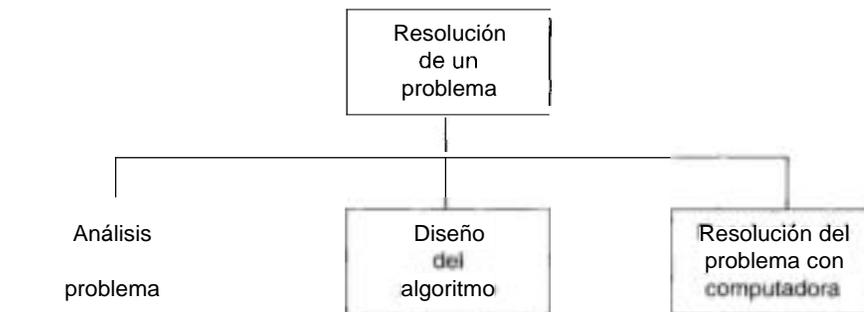


Figura 2.1. Análisis del problema.

Para poder definir bien un problema es conveniente responder a las siguientes preguntas:

- ¿Qué entradas se requieren? (tipo y cantidad).
- ¿Cuál es la salida deseada? (tipo y cantidad).
- ¿Qué método produce la salida deseada?

Problema 2.1

Se desea obtener una tabla con las depreciaciones acumuladas y los valores reales de cada año, de un automóvil comprado en 1.800.000 pesetas en el año 1996, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 120.000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

$$D = \frac{\text{coste} - \text{valor de recuperación}}{\text{vida útil}}$$

$$D = \frac{1.800.000 - 120.000}{6} = \frac{1.680.000}{6} = 280.000$$

Entrada	{	coste original vida útil valor de recuperación
Salida		depreciación anual por año depreciación acumulada en cada año valor del automóvil en cada año
Proceso		depreciación acumulada cálculo de la depreciación acumulada cada año cálculo del valor del automóvil en cada año

La tabla siguiente muestra la salida solicitada

Año	Depreciación	Depreciación acumulada	Valor anual
1 (1996)	280.000	280.000	1.520.000
2 (1997)	280.000	560.000	1.240.000
3 (1998)	280.000	840.000	960.000
4 (1999)	280.000	1.120.000	680.000
5 (2000)	280.000	1.400.000	400.000
6 (2001)	280.000	2.180.000	120.000

2.1.2. Diseño del algoritmo

En la etapa de análisis del proceso de programación se determina *qué* hace el programa. En la etapa de diseño se determina *como* hace el programa la tarea solicitada. Los métodos más eficaces para el proceso de diseño se basan en el conocido por *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividir estos subproblemas en otros de nivel más bajo, hasta que pueda ser *implementada* una solución en la computadora. Este método se conoce técnicamente como **diseño descendente** (*top-down*) o **modular**. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina *refinamiento sucesivo*.

Cada subprograma es resuelto mediante un **módulo** (*subprograma*) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. Los programas estructurados de esta forma se dice que tienen un *diseño modular* y el método de romper el programa en módulos más pequeños se llama *programación modular*. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.

3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina **diseño del algoritmo**.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

2.1.3. Herramientas de programación

Las dos herramientas más utilizadas comúnmente para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*.

Diagramas de flujo

Un **diagrama de flujo** (*flowchart*) es una representación gráfica de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normalización (ANSI), y los más frecuentemente empleados se muestran en la Figura 2.2, junto con una plantilla utilizada para el dibujo de los diagramas de flujo (Fig. 2.3). En la Figura 2.4 se representa el diagrama de flujo que resuelve el Problema 2.1.

Pseudocódigo

El **pseudocódigo** es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como *un lenguaje de especificaciones de algoritmos*.

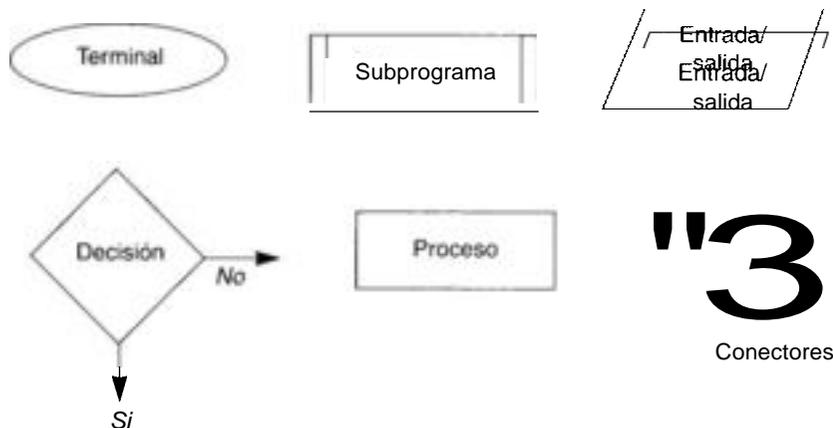


Figura 2.2. Símbolos más utilizados en los diagramas de flujo.

Aunque no existen reglas para escritura del pseudocódigo en español, se ha recogido una notación estándar que se utilizará en el libro y que ya es muy empleada en los libros de programación en espa-

ñol³. Las palabras reservadas básicas se representarán en letras negritas minúsculas. estas palabras son traducción libre de palabras reservadas de lenguajes como C, Pascal, etc. Más adelante se indicarán los pseudocódigos fundamentales a utilizar en esta obra.

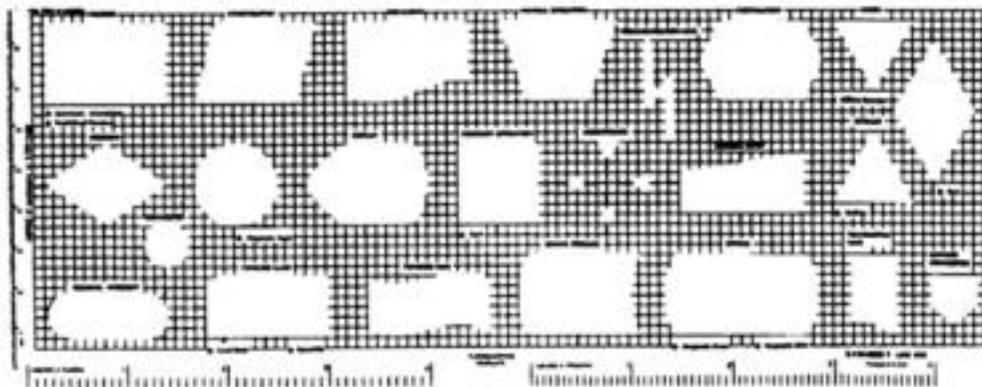


Figura 2.3. Plantilla para dibujo de diagramas de flujo.

El pseudocódigo que resuelve el Problema 2.1 es:

```
Previsiones de depreciacion
Introducir coste
    vida util
    valor final de rescate (recuperacion)
imprimir cabeceras
Establecer el valor inicial del Año
Calcular depreciacion
mientras valor año =< vida util hacer
    calcular depreciacion acumulada
    calcular valor actual
    imprimir una linea en la tabla
    incrementar el valor del año
fin de mientras
```

Ejemplo 2.1

Calcular la paga neta de un trabajador conociendo el número de horas trabajadas, la tarifa horaria y la tasa de impuestos.

Algoritmo

1. Leer Horas, Tarifa, tasa
 2. Calcular PagaBruta = Horas * Tarifa
 3. Calcular impuestos = PagaBruta * Tasa
 4. Calcular PagaNeta = PagaBruta - Impuestos
 5. Visualizar PagaBruta, Impuestos, PagaNeta
-

³ Para mayor ampliación sobre el pseudocódigo, puede consultar, entre otras, algunas de estas obras: *Fundamentos de programación*, Luis Joyanes, 2.ª edición, 1997; *Metodología de la programación*, Luis Joyanes, 1986; *Problemas de Metodología de la programación*, Luis Joyanes, 1991 (todas ellas publicadas en McGraw-Hill, Madrid), así como *Introducción a la programación*, de Clavel y Biondi, Barcelona: Masson, 1987, o bien *Introducción a la programación y a las estructuras de datos*, de Braunstein y Groia, Buenos Aires: Editorial Eudeba, 1986. Para una formación práctica puede consultar: *Fundamentos de programación: Libro de problemas* de Luis Joyanes, Luis Rodríguez y Matilde Fernández en McGraw-Hill (Madrid, 1998).

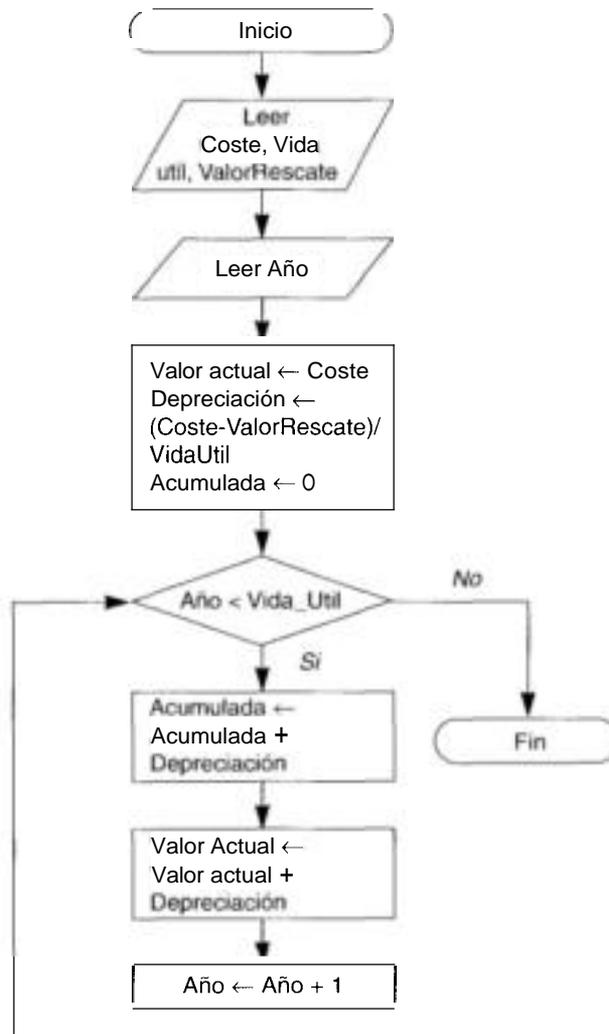


Figura 2.4. Diagrama de flujo (Ejemplo 2.1).

Ejemplo 2.2

Calcular el valor de la suma $1+2+3+\dots+100$.

Algoritmo

Se utiliza una variable *Contador* como un contador que genere los sucesivos números enteros, y *Suma* para almacenar las sumas parciales $1, 1+2, 1+2+3, \dots$

1. Establecer *Contador* a 1
2. Establecer *Suma* a 0
3. **mientras** *Contador* < = 100 **hacer**
 Sumar *Contador* a *Suma*
 Incrementar *Contador* en 1
fin-mientras
4. Visualizar *Suma*

2.1.4. Codificación de un programa

Codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la conversión del algoritmo en programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural expresarlas en el lenguaje de programación correspondiente.

```

/*
 Este programa obtiene una tabla de depreciaciones acumuladas y
 valores reales de cada año de un determinado producto
*/
#include <stdio.h>
void main()
{
    double Coste, Depreciacion,
           Valor_Recuperacion,
           Valor-actual,
           Acumulado,
           Valor_Anual;
    int Anio, Vida_util;
    puts("Introduzca coste, valor recuperación y vida Útil");
    scanf("%lf %lf %lf",&Coste,&Valor_Recuperacion,&Vida_Util);
    puts("Introduzca año actual");
    scanf("%d",&Anio);
    Valor_Actual = Coste;
    Depreciación = (Coste-Valor_Recuperacion)/Vida_Util;
    Acumulado = 0;
    puts("Año Depreciación Dep. Acumulada");
    while (Anio < Vida_Util)
    {
        Acumulado = Acumulado + Depreciacion;
        Valor_Actual = ValorActual - Depreciacion;
        printf("Año: %d, Depreciacion:%.2lf, %.2lf Acumulada",
              Anio,Depreciacion,Acumulado);
        Anio = Anio + 1;
    }
}

```

Documentación interna

Como se verá más tarde, la documentación de un programa se clasifica en *interna* y *externa*. La *documentación interna* es la que se incluye dentro del código del programa fuente mediante comentarios que ayudan a la comprensión del código. Todas las líneas de programas que comiencen con un símbolo */** son *comentarios*. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios sólo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios.

Debido a que las máquinas actuales soportan grandes memorias (64 Mb o 128Mb de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que incluya el mayor número de comentarios posibles, pero, eso sí, que sean significativos.

2.1.5. Compilación y ejecución de un programa

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un disco. Esta operación se realiza con un programa editor, posteriormente el programa fuente se convierte en un *archivo de programa* que se guarda (graba) en disco.

El **programa fuente** debe ser traducido a lenguaje máquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (*errores de compilación*) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el **programa objeto** que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de **montaje o enlace** (*link*), carga, del programa objeto con las librerías del programa del compilador. El proceso de montaje produce un **programa ejecutable**. La Figura 2.5 describe el proceso completo de compilación/ejecución de un programa.

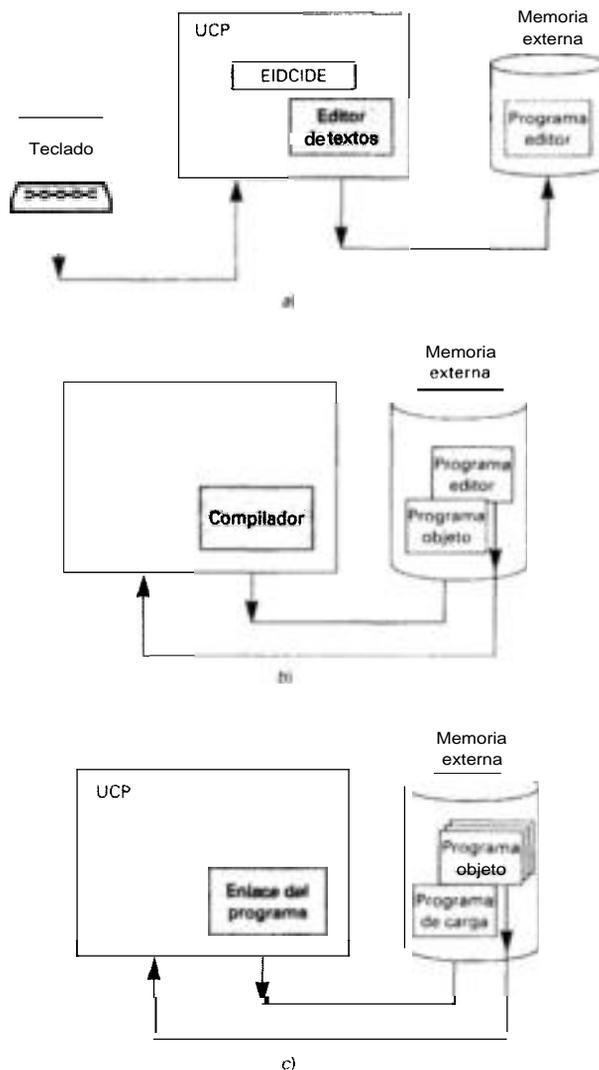


Figura 2.5. Fases de la compilación/ejecución de un programa: a) edición; b) compilación; c) montaje o enlace.

Cuando el programa ejecutable se ha creado, se puede ya ejecutar (correr o rodar) desde el sistema operativo con sólo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados **errores en tiempo de ejecución**), se obtendrá la salida de resultados del programa.

Las instrucciones u Órdenes para compilar y ejecutar un programa en C puede variar según el tipo de compilador. Así el proceso de Visual C++ 6 es diferente de C bajo UNIX o bajo Linux.

2.1.6. Verificación y depuración de un programa

La *verificación* o *compilación* de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados *datos de test* o *prueba*, que determinarán si el programa tiene errores («*bugs*»). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.

Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

1. *Errores de compilación*. Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.
2. *Errores de ejecución*. Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.
3. *Errores lógicos*. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertir el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

2.1.7. Documentación y mantenimiento

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de un problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser *interna* y *externa*. La *documentación interna* es la contenida en líneas de comentarios. La *documentación externa* incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan *mantenimiento del programa*. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas **1.0, 1.1, 2.0, 2.1**, etc. (Si los cambios introducidos son importantes, se varía el primer dígito [**1.0, 2.0,...**], en caso de pequeños cambios sólo se varía el segundo dígito [**2.0, 2.1...**].)

2.2. PROGRAMACIÓN MODULAR

La *programación modular* es uno de los métodos de diseño más flexible y potentes para mejorar la productividad de un programa. En programación modular el programa se divide en *módulos* (partes independientes), cada una de las cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analizan, codifican y ponen a punto por separado.

Cada programa contiene un módulo denominado *programa principal* que controla todo lo que sucede; se transfiere el control a *submódulos* (posteriormente se denominarán *subprogramas*), de modo que ellos puedan ejecutar sus funciones; sin embargo, cada submódulo devuelve el control al módulo principal cuando se haya completado su tarea. Si la tarea asignada a cada submódulo es demasiado compleja, éste deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica que ejecutar. Esta tarea puede ser *entrada, salida, manipulación de datos, control de otros módulos o alguna combinación de éstos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe eventualmente devolver el control al módulo del cual se recibe originalmente el control.

Los módulos son independientes en el sentido en que ningún módulo puede tener acceso directo a cualquier otro módulo excepto el módulo al que llama y sus propios submódulos. Sin embargo, los resultados producidos por un módulo pueden ser utilizados por cualquier otro módulo cuando se transfiera a ellos el control.

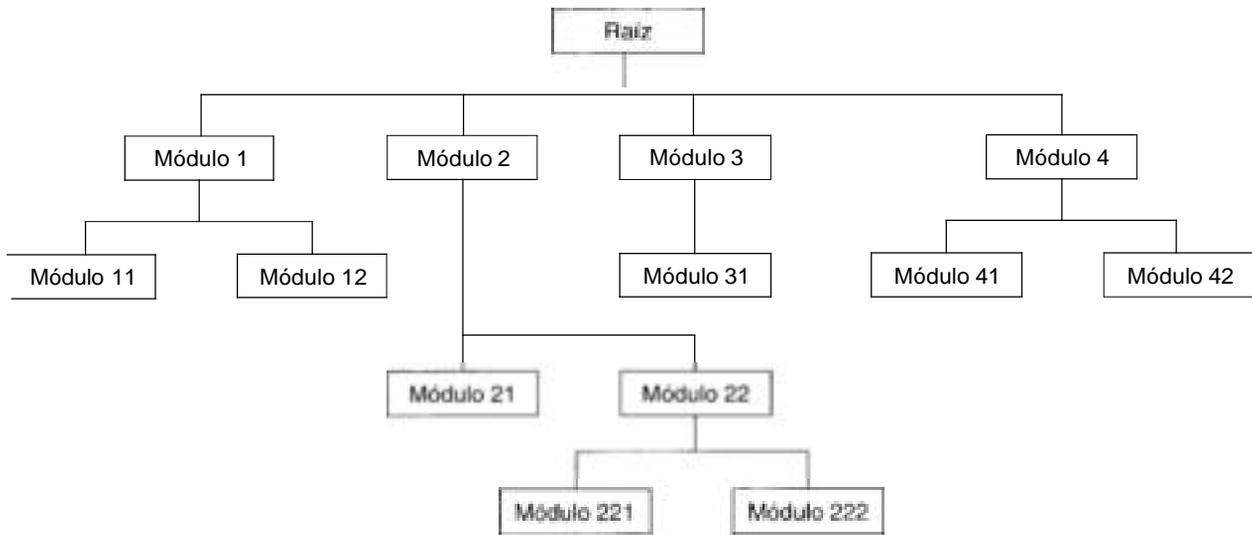


Figura 2.6. Programación modular.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en diferentes partes del mismo programa. Esto reducirá el tiempo del diseño del algoritmo y posterior codificación del programa. Además, un módulo se puede modificar radicalmente sin afectar a otros módulos, incluso sin alterar su función principal.

La descomposición de un programa en módulos independientes más simples se conoce también como el método de «**divide y vencerás**» (*divide and conquer*). Se diseña cada módulo con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

2.3. PROGRAMACIÓN ESTRUCTURADA

Los términos *programación modular*, *programación descendente* y *programación estructurada* se introdujeron en la segunda mitad de la década de los sesenta y a menudo sus términos se utilizan como sinónimos aunque no significan lo mismo. La programación modular y descendente ya se ha examinado anteriormente. La *programación estructurada* significa escribir un programa de acuerdo a las siguientes reglas:

- El programa tiene un diseño modular.
- Los módulos son diseñados de modo descendente.
- Cada módulo se codifica utilizando las tres estructuras de control básicas: *secuencia*, *selección* y *repetición*.

Si está familiarizado con lenguajes como **BASIC**, Pascal, FORTRAN o C, la programación estructurada significa también *programación sin GOTO* (C no requiere el uso de la sentencia **GOTO**).

El término *programación estructurada* se refiere a un conjunto de técnicas que han ido evolucionando desde los primeros trabajos de Edgar Dijkstra. Estas técnicas aumentan considerablemente la productividad del programa reduciendo en elevado grado el tiempo requerido para escribir, verificar, depurar y mantener los programas. La programación estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los programas y, por consiguiente, reducen los errores; hace los programas más fáciles de escribir, verificar, leer y mantener. Los programas deben estar dotados de una estructura.

La **programación estructurada** es el conjunto de técnicas que incorporan:

- *recursos abstractos*,
- *diseño descendente (top-down)*,
- *estructuras básicas*.

2.3.1. Recursos abstractos

La programación estructurada se auxilia de los recursos abstractos en lugar de los recursos concretos de que dispone un determinado lenguaje de programación.

Descomponer un programa en términos de recursos abstractos —según Dijkstra— consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples capaces de ejecutarlas o que constituyan instrucciones de computadoras disponibles.

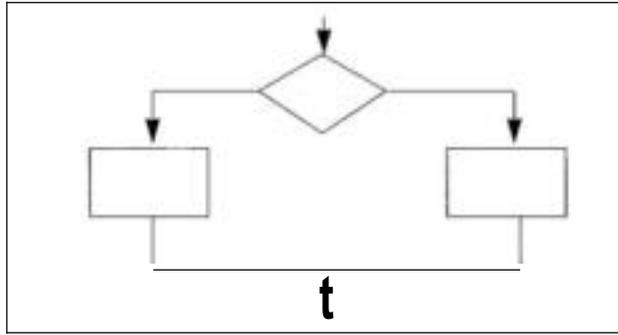
2.3.2. Diseño descendente (*top-down*)

El **diseño descendente** (*top-down*) es el proceso mediante el cual un problema se descompone en una serie de niveles o pasos sucesivos de refinamiento (*stepwise*). La metodología descendente consiste en efectuar una relación entre las sucesivas etapas de estructuración de modo que se relacionasen unas con otras mediante entradas y salidas de información. Es decir, se descompone el problema en etapas o estructuras jerárquicas, de forma que se puede considerar cada estructura desde dos puntos de vista: *¿qué hace?* y *¿cómo lo hace?*

Si se considera un nivel n de refinamiento, las estructuras se consideran de la siguiente manera:



Nivel n : desde el exterior
«¿lo que hace?»



Nivel $n + 7$: Vista desde el interior
«¿cómo lo hace?»

El diseño descendente se puede ver en la Figura 2.7

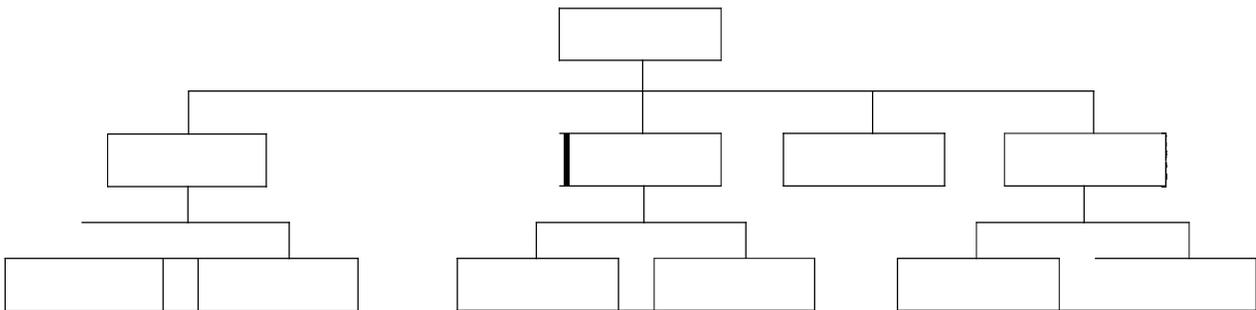


Figura 2.7. Diseño descendente.

2.3.3. Estructuras de control

Las *estructuras de control* de un lenguaje de programación son métodos de especificar el orden en que las instrucciones de un algoritmo se ejecutarán. El orden de ejecución de las sentencias (lenguaje) o instrucciones determinan el *flujo de control*. Estas estructuras de control son, por consiguiente, fundamentales en los lenguajes de programación y en los diseños de algoritmos especialmente los pseudocódigos.

Las tres estructuras de control básico son:

- *secuencia*
- *selección*
- *repetición*

y se estudian en los Capítulos 5 y 6.

La programación estructurada hace los programas más fáciles de escribir, verificar, leer y mantener; utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas.

2.3.4. Teorema de la programación estructurada: estructuras básicas

En mayo de 1966, Bohm y Jacopini demostraron que *un programa propio* puede ser escrito utilizando solamente tres tipos de estructuras de control.

- *secuenciales,*
- *selectivas,*
- *repetitivas.*

Un programa se define como **propio** si cumple las siguientes características:

- *Posee un solo punto de entrada y uno de salida o fin para control del programa.*
- *Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas las partes del programa.*
- *Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos (sinfin).*

Los Capítulos 5 y 6 se dedican al estudio de las estructuras de control selectivas y repetitivas

La programación estructurada significa:

- El programa completo tiene un diseño modular.
 - **Los** módulos se diseñan con metodología descendente (puede hacerse también ascendente).
 - Cada módulo se codifica utilizando las tres estructuras de control básicas: secuenciales, selectivas y repetitivas (ausencia total de sentencias **GOTO**).
- **Estructuración y modularidad** son conceptos complementarios (se solapan).

2.4. REPRESENTACIÓN GRÁFICA DE LOS ALGORITMOS

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráficamente o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, *su codificación*.

Los métodos usuales para representar un algoritmo son:

1. *diagrama de flujo,*
2. *diagrama N-S (Nassi-Schneiderman),*
3. *lenguaje de especificación de algoritmos: pseudocódigo,*
4. *lenguaje español, inglés...*
5. *fórmulas.*

Los métodos 4 y 5 no suelen ser fáciles de transformar en programas. Una descripción en *español narrativo* no es satisfactoria, ya que es demasiado prolija y generalmente ambigua. Una *fórmula*, sin embargo, es buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado) es un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

$$x1 = (-b + \sqrt{b^2 - 4ac}) / 2a \quad x2 = (-b - \sqrt{b^2 - 4ac}) / 2a$$

y significa lo siguiente:

1. *Eleve al cuadrado b.*
2. *Toma a; multiplicar por c; multiplicar por 4.*
3. *Restar el resultado obtenido de 2 del resultado de 1, etc.*

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

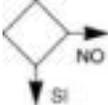
2.4.1. Diagramas de flujo

Un **diagrama de flujo** (*flowchart*) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) estándar mostrados en la Tabla 2.1 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas *líneas de flujo*, que indican la secuencia en que se debe ejecutar.

La Figura 2.8 es un diagrama de flujo básico. Este diagrama representa la resolución de un programa que deduce el salario neto de un trabajador a partir de la lectura del nombre, horas trabajadas, precio de la hora, y sabiendo que los impuestos aplicados son el 25 por 100 sobre el salario bruto.

Los símbolos estándar normalizados por ANSI (abreviatura de *American National Standards Institute*) son muy variados. En la Figura 2.9 se representa una plantilla de dibujo típica donde se contemplan la mayoría de los símbolos utilizados en el diagrama; sin embargo, los símbolos más utilizados representan:

Tabla 2.1. Símbolos de diagrama de flujo

Símbolos principales	Función
	Terminal (representa el comienzo, «inicio», y el final, «fin» de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa).
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos. «entrada», o registro de la información procesada en un periférico. «salida»).
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.).
	Decisión (indica operaciones lógicas o de comparación entre datos — normalmente dos — y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas — respuestas SI o NO — pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector (sirve para enlazar dos partes cualesquiera de un organograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama).
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organograma situado en páginas diferentes).
	Llamada subrutina o a un proceso predeterminado (una subrutina es un módulo independiente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).

(Continúa)

(Continuación)

Símbolos secundarios	Función
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

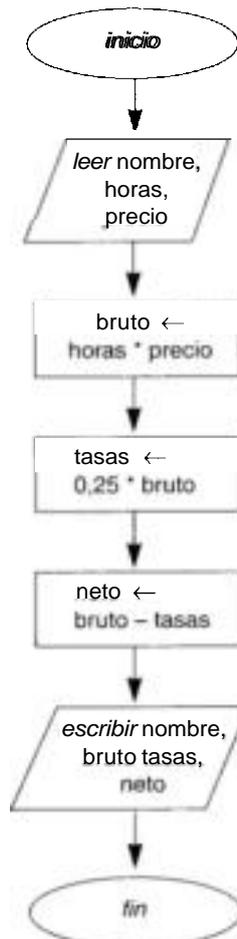


Figura 2.8. Diagrama de flujo.

- proceso,
- decisión,
- conectores,
- fin,
- entrada/salida,
- dirección del flujo.

Se resume en la Figura 2.8 en un diagrama de flujo:

- existe una caja etiquetada "inicio", que es de tipo elíptico,
- existe una caja etiquetada "fin" de igual forma que la anterior,
- si existen otras cajas, normalmente son rectangulares, tipo rombo o paralelogramo (el resto de las figuras se utilizan sólo en diagramas de flujo generales o de detalle y no siempre son imprescindibles).

Se puede escribir más de un paso del algoritmo en una sola caja rectangular. El uso de flechas significa que la caja no necesita ser escrita debajo de su predecesora. Sin embargo, abusar demasiado de esta flexibilidad conduce a diagramas de flujo complicados e ininteligibles.

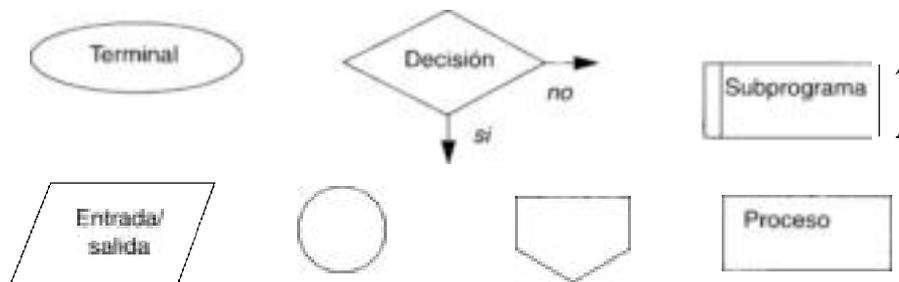


Figura 2.9. Plantilla típica para diagramas de flujo.

Ejemplo 2.3

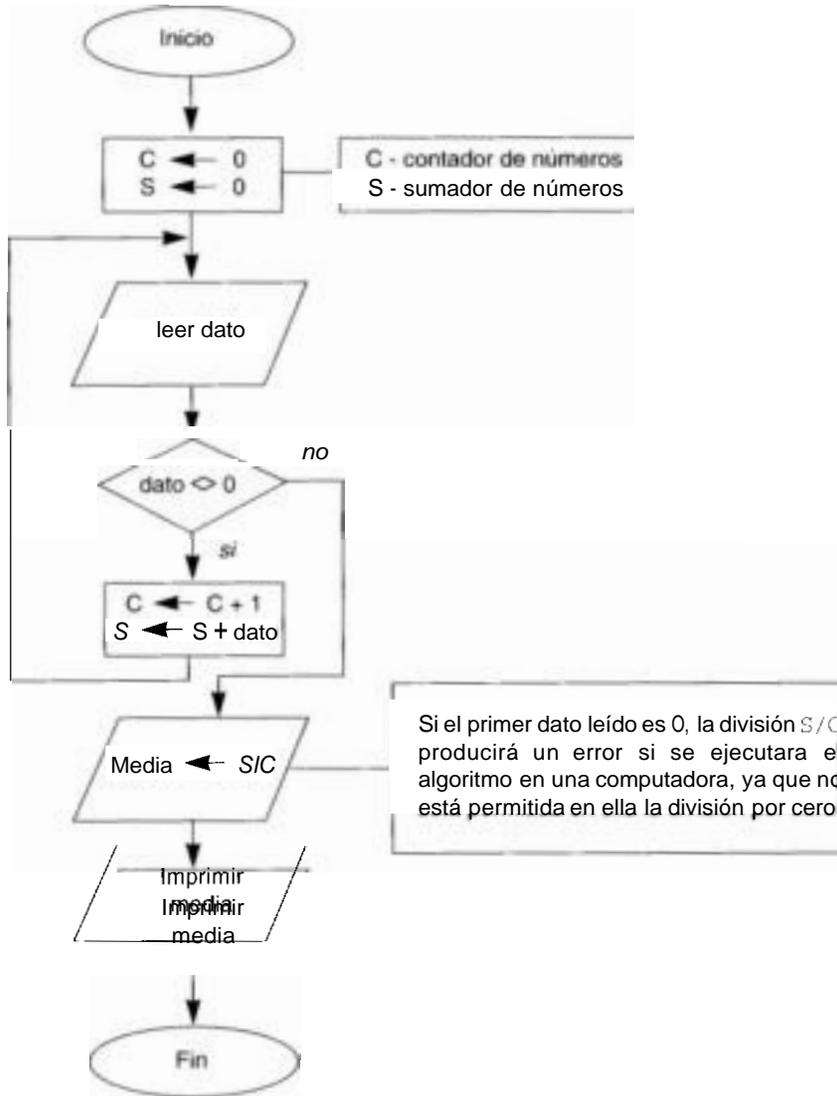
Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero — como entrada — indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, nuestro algoritmo en forma descriptiva sería:

1. inicializar contador de numeros C y variable suma S.
2. Leer un numero
3. Si el numero leído es cero :
 - calcular la media ;
 - imprimir la media ;
 - fin del proceso.
 Si el numero leído no es cero :
 - calcular la suma ;
 - incrementar en uno el contador de números ;
 - ir al paso 2.
4. Fin.

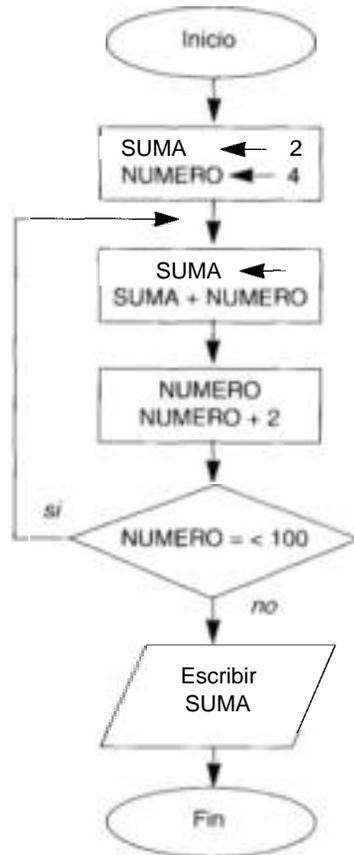
El refinamiento del algoritmo conduce a los pasos sucesivos necesarios para realizar las operaciones de lectura, verificación del último dato, suma y media de los datos.

Si el primer dato leído es 0, la división S/C produciría un error si se ejecutara el algoritmo en una computadora, ya que no está permitida en ella la división por cero.



Ejemplo 2.4

Suma de los números pares comprendidos entre 2 y 100.

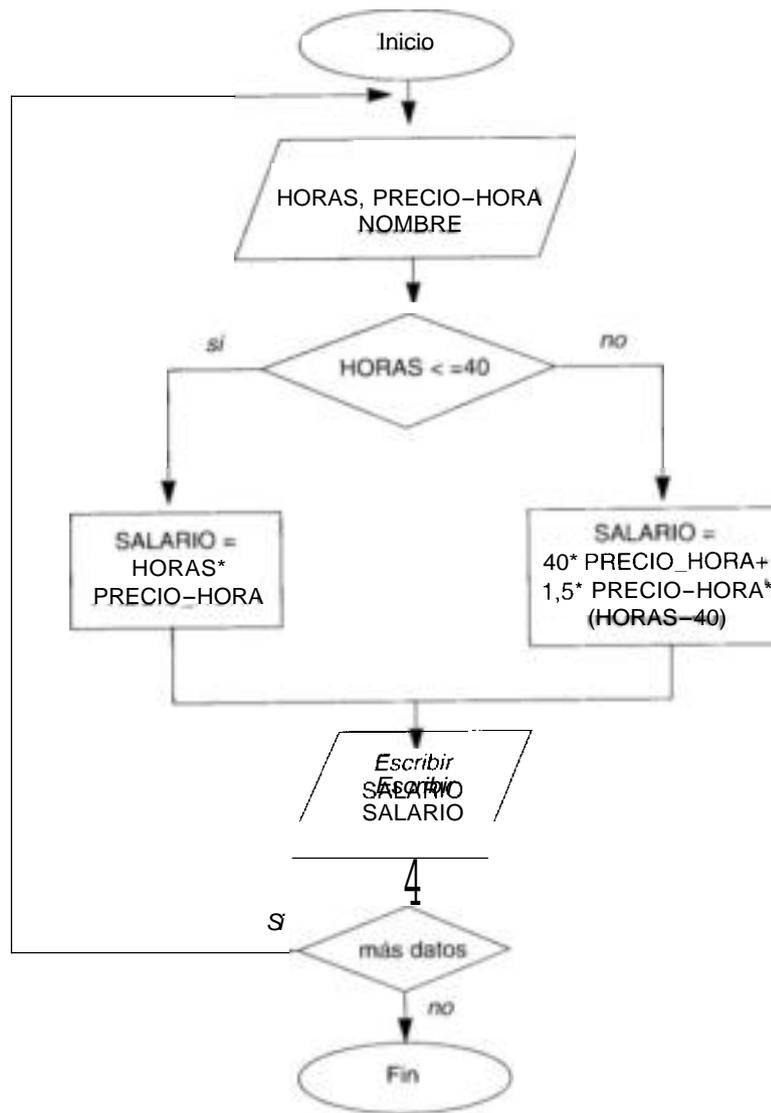
**Ejemplo 2.5**

Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, subiendo que éstos se calculan en base a las horas semanales trabajadas y de acuerdo a un precio especificado por horas. Si se pasan de cuarenta horas semanales, las horas extraordinarias se pagarán a razón de 1.5 veces la hora ordinaria.

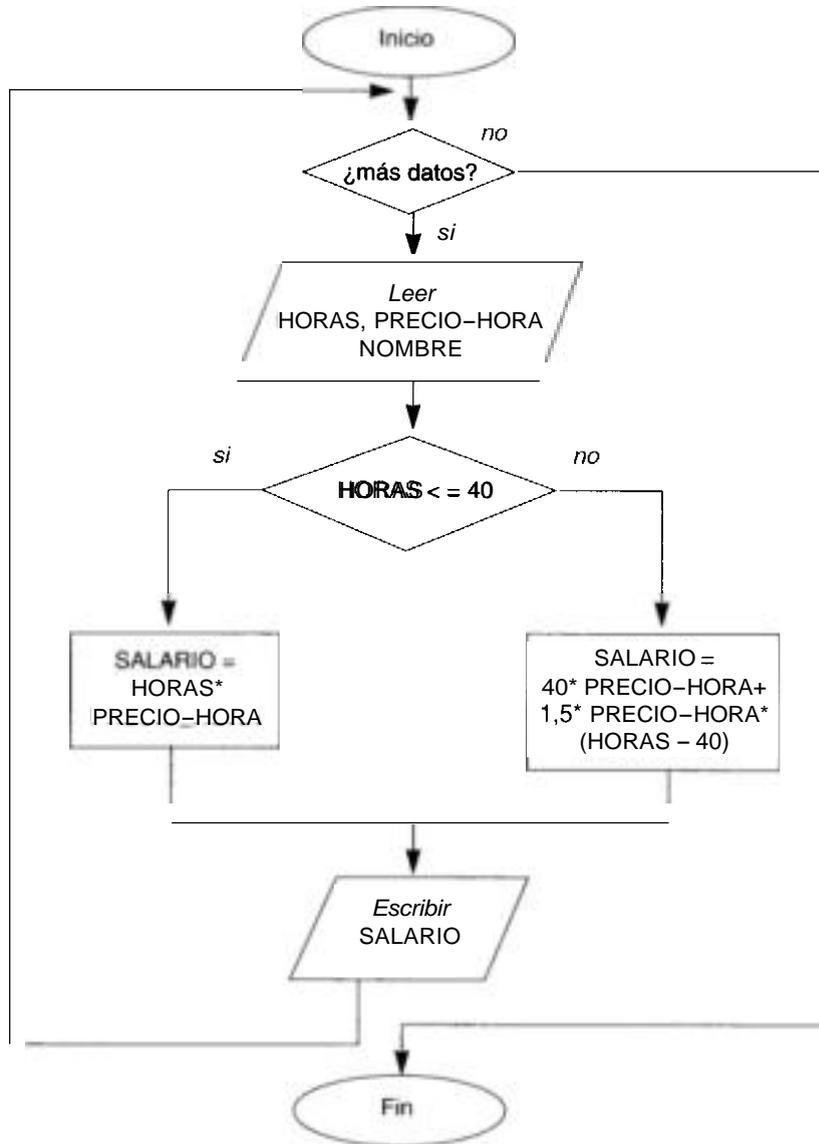
Los cálculos son:

1. Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS, PRECIO-HORA, NOMBRE).
2. Si HORAS \leq 40, entonces SALARIO es el producto de horas por PRECIO-HORA.
3. Si HORAS $>$ 40, entonces SALARIO es la suma de 40 veces PRECIO-HORA más 1.5 veces PRECIO-HORA por (HORAS-40).

El diagrama de flujo completo del algoritmo se indica a continuación:



Una variante también válida al diagrama de flujo anterior es:



Ejemplo 2.6

La escritura de algoritmos para realizar operaciones sencillas de conteo es una de las primeras cosas que un ordenador puede aprender:

Supongamos que se proporciona una secuencia de números, tales como

5 3 0 2 4 4 0 0 2 3 6 0 2

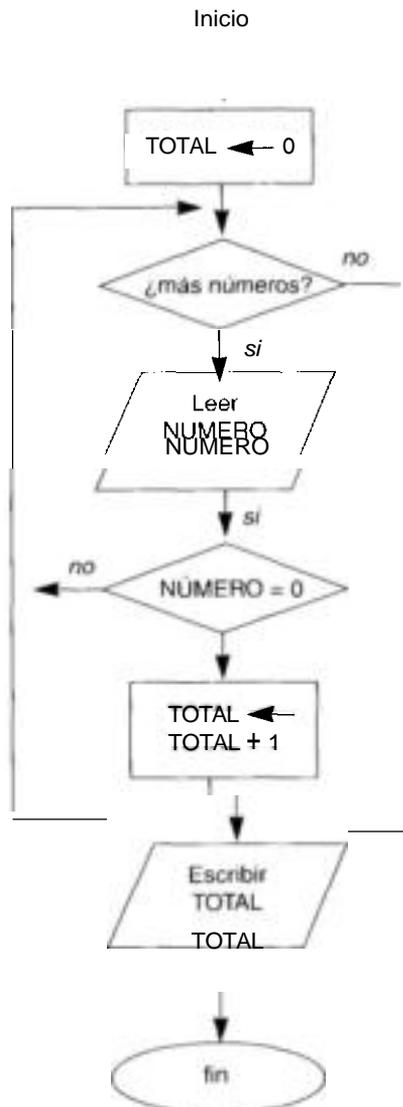
y desea contar e imprimir el número de ceros de la secuencia.

El algoritmo es muy sencillo, ya que sólo basta leer los números de izquierda a derecha, mientras se cuentan los ceros. Utiliza como variable la palabra **NUMERO** para los números que se examinan y **TOTAL** para el número de ceros encontrados. Los pasos a seguir son:

1. Establecer **TOTAL** a cero.
2. ¿Quedan más números a examinar?

3. Si no quedan numeros, imprimir el valor de TOTAL y fin.
4. Si existen mas numeros, ejecutar los pasos 5 a 8.
5. Leer el siguiente numero y dar su valor a la variable NUMERO.
6. Si NUMERO = 0, incrementar TOTAL en 1
7. Si NUMERO <> 0, no modificar TOTAL.
8. Retornar al paso 2.

El diagrama de flujo correspondiente es:



Ejemplo 2.7

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición, escribir «Iguales» y, en caso contrario, escribir «Distintas».

En el caso de que los números sean: 3 9 6

la respuesta es "Iguales", ya que $3 + 6 = 9$. Sin embargo, si los números fueran:

2 3 4

el resultado sería 'Distintas'.

Para resolver este problema, se puede comparar la suma de cada pareja con el tercer número. Con tres números solamente existen tres parejas distintas y el algoritmo de resolución del problema será fácil.

1. Leer los tres valores, A, B y C.
2. Si $A + B = C$ escribir "Iguales" y parar.
3. Si $A + C = B$ escribir "Iguales" y parar.
4. Si $B + C = A$ escribir "Iguales" y parar.
5. Escribir 'Distintas' y parar.

El diagrama de flujo correspondiente es la Figura 2.10.

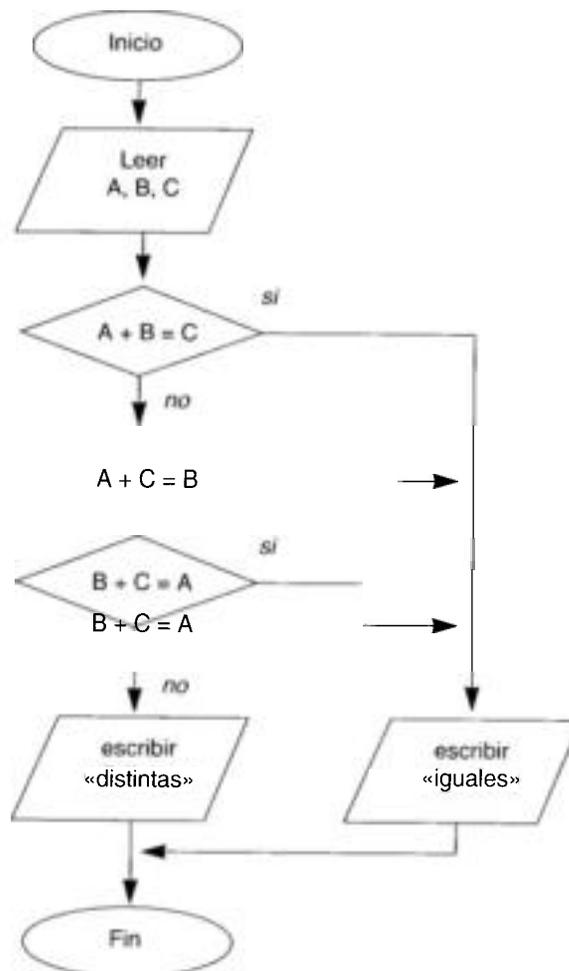


Figura 2.10. Diagrama de flujo (Ejemplo 2.7).

2.5. DIAGRAMAS DE NASSI-SCHNEIDERMAN (N-S)

El diagrama N-S de Nassi Schneiderman — también conocido como diagrama de Chapin— es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben en cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.

Un algoritmo se representa con un rectángulo en el que cada banda es una acción a realizar:

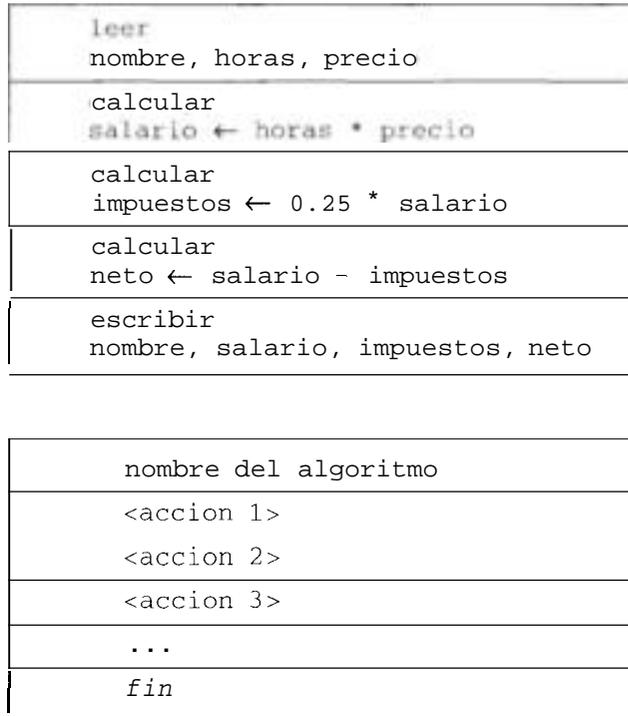


Figura 2.11. Representación gráfica N-S de un algoritmo.

Otro ejemplo es la representación de la estructura condicional (Fig. 2.12).

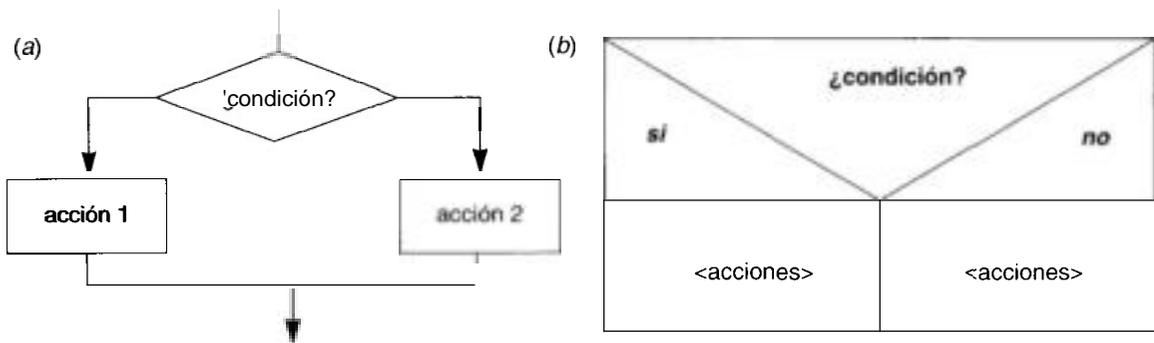


Figura 2.12. Estructura condicional o selectiva: (a)diagrama de flujo; (b) diagrama N-S.

Ejemplo 2.8

Se desea calcular el salario neto semanal de un trabajador en función del número de horas trabajadas y la tasa de impuestos:

- las primeras 35 horas se pagan a tarifa normal,
- las horas que pasen de 35 se pagan a 1,5 veces la tarifa normal,
- las tasas de impuestos son:
 - a) las primeras 60.000 pesetas son libres de impuestos,
 - b) las siguientes 40.000 pesetas tienen un 25 por 100 de impuesto,
 - c) las restantes, un 45 por 100 de impuestos,
- la tarifa horaria es 800 pesetas.

También se desea escribir el nombre, salario bruto, tasas y salario neto (*este ejemplo se deja como ejercicio al alumno*).

2.6. EL CICLO DE VIDA DEL SOFTWARE

Existen dos niveles en la construcción de programas: aquéllos relativos a pequeños programas (los que normalmente realizan programadores individuales) y aquellos que se refieren a sistemas de desarrollo de programas grandes (*proyectos de software*) y que, generalmente, requieren un equipo de programadores en lugar de personas individuales. El primer nivel se denomina *programación a pequeña escala*; el segundo nivel se denomina *programación a gran escala*.

La programación en pequeña escala se preocupa de los conceptos que ayudan a crear pequeños programas — aquellos que varían en longitud desde unas pocas líneas a unas pocas páginas —. En estos programas se suele requerir claridad y precisión mental y técnica. En realidad, el interés mayor desde el punto de vista del futuro programador profesional está en los programas de gran escala que requiere de unos principios sólidos y firmes de lo que se conoce como *ingeniería de software* y que constituye un conjunto de técnicas para facilitar el desarrollo de programas de computadora. Estos programas o mejor proyectos de software están realizados por equipos de personas dirigidos por un director de proyectos (analista o ingeniero de software) y los programas pueden tener más de 100.000 líneas de código.

El desarrollo de un buen sistema de software se realiza durante el *ciclo de vida* que es el período de tiempo que se extiende desde la concepción inicial del sistema hasta su eventual retirada de la comercialización o uso del mismo. Las actividades humanas relacionadas con el ciclo de vida implican **pro**cesos tales como análisis de requisitos, diseño, implementación, codificación, pruebas, verificación, documentación, mantenimiento y evolución del sistema y obsolescencia. En esencia el ciclo de vida del software comienza con una idea inicial, incluye la escritura y depuración de programas, y continúa durante años con correcciones y mejoras al software original⁴.

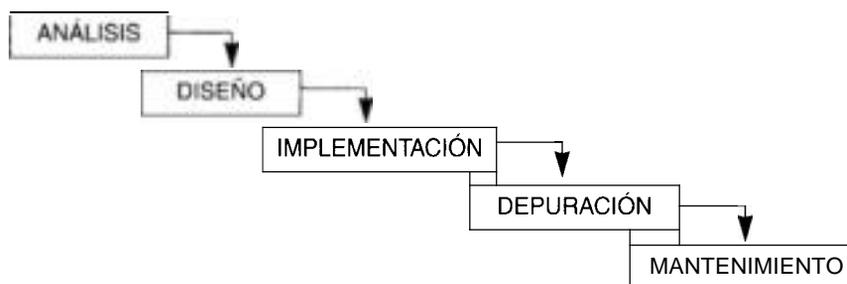


Figura 2.13. Ciclo de vida del software.

⁴ Carrano, Hellman y Verof: *Data structures and problem solving with Turbo Pascal*, The Benjamin/Cummings Publishing, 1993, pág. 210.

El ciclo de vida del software es un proceso iterativo, de modo que se modificarán las sucesivas etapas en función de la modificación de las especificaciones de los requisitos producidos en la fase de diseño o implementación, o bien una vez que el sistema se ha implementado, y probado, pueden aparecer errores que será necesario corregir y depurar, y que requieren la repetición de etapas anteriores.

La Figura 2.13 muestra el ciclo de vida de software y la disposición típica de sus diferentes etapas en el sistema conocido como *ciclo de vida en cascada*, que supone que la salida de cada etapa es la entrada de la etapa siguiente.

2.6.1. Análisis

La primera etapa en la producción de un sistema de software es decidir exactamente *qué* se supone ha de hacer el sistema. Esta etapa se conoce también como *análisis de requisitos o especificaciones* y por esta circunstancia muchos tratadistas suelen subdividir la etapa en otras dos:

- *Análisis y definición del problema.*
- *Especificación de requisitos.*

La parte más difícil en la tarea de crear un sistema de software es definir cuál es el problema, y a continuación especificar lo que se necesita para resolverlo. Normalmente la definición del problema comienza analizando los requisitos del usuario, pero estos requisitos, con frecuencia, suelen ser imprecisos y difíciles de describir. Se deben especificar todos los aspectos del problema, pero con frecuencia las personas que describen el problema no son programadores y eso hace imprecisa la definición. La fase de especificación requiere normalmente la comunicación entre los programadores y los futuros usuarios del sistema e iterar la especificación, hasta que tanto el especificador como los usuarios estén satisfechos de las especificaciones y hayan resuelto el problema normalmente.

En la etapa de especificaciones puede ser muy útil para mejorar la comunicación entre las diferentes partes implicadas construir un prototipo o modelo sencillo del sistema final; es decir, escribir un programa prototipo que simule el comportamiento de las partes del producto software deseado. Por ejemplo, un programa sencillo —incluso ineficiente— puede demostrar al usuario la interfaz propuesta por el analista. Es mejor descubrir cualquier dificultad o cambiar su idea original ahora que después de que la programación se encuentre en estado avanzado o, incluso, terminada. El modelado de datos es una herramienta muy importante en la etapa de definición del problema. Esta herramienta es muy utilizada en el diseño y construcción de bases de datos.

Tenga presente que el usuario final, normalmente, no conoce exactamente lo que desea que haga el sistema. Por consiguiente, el analista de software o programador, en su caso, debe interactuar con el usuario para encontrar lo que el usuario *deseará* que haga el sistema. En esta etapa se debe responder a preguntas tales como:

- ¿Cuáles son los datos de entrada?
- ¿Qué datos son válidos y qué datos no son válidos?
- ¿Quién utilizará el sistema: especialistas cualificados o usuarios cualesquiera (sin formación)?
- ¿Qué interfaces de usuario se utilizarán?
- ¿Cuáles son los mensajes de error y de detección de errores deseables? ¿Cómo debe actuar el sistema cuando el usuario cometa un error en la entrada?
- ¿Qué hipótesis son posibles?
- ¿Existen casos especiales?
- ¿Cuál es el formato de la salida?
- ¿Qué documentación es necesaria?
- ¿Qué mejoras se introducirán —probablemente— al programa en el futuro?
- ¿Cómo debe ser de rápido el sistema?
- ¿Cada cuanto tiempo ha de cambiarse el sistema después que se haya entregado?

El resultado final de la fase de análisis es una *especificación de los requisitos del software*.

- Descripción del problema previa y detalladamente.
 - Prototipos de programas que pueden ayudar a resolver el problema.

2.6.2. Diseño

La especificación de un sistema indica *lo que* el sistema debe *hacer*. La etapa de diseño del sistema indica *cómo* ha de hacerse. Para un sistema pequeño, la etapa de diseño puede ser tan sencilla como escribir un algoritmo en pseudocódigo. Para un sistema grande, esta etapa incluye también la fase de diseño de algoritmos, pero incluye el diseño e interacción de un número de algoritmos diferentes, con frecuencia sólo bosquejados, así como una estrategia para cumplir todos los detalles y producir el código correspondiente.

Es preciso determinar si se pueden utilizar programas o subprogramas que ya existen o es preciso construirlos totalmente. El proyecto se ha de dividir en módulos utilizando los principios de diseño descendente. A continuación, se debe indicar la interacción entre módulos; un diagrama de estructuras proporciona un esquema claro de estas relaciones⁷.

En este punto, es importante especificar claramente no sólo el propósito de cada módulo, sino también el *flujo de datos* entre módulos. Por ejemplo, se debe responder a las siguientes preguntas: ¿Qué datos están disponibles al módulo antes de su ejecución? ¿Qué supone el módulo? ¿Qué hacen los datos después de que se ejecuta el módulo? Por consiguiente, se deben especificar en detalle las hipótesis, entrada y salida para cada módulo. Un medio para realizar estas especificaciones es escribir una *precondición*, que es una descripción de las condiciones que deben cumplirse al principio del módulo y una *postcondición*, que es una descripción de las condiciones al final de un módulo. Por ejemplo, se puede describir un subprograma que ordena una lista (un array) de la forma siguiente:

```
subprograma ordenar (A, n)
  {Ordena una lista en orden ascendente}
  precondición: A es un array de n enteros, 1 <= n <= Max.
  postcondición: A[1] <= A[2] <...<= A[n], n es inalterable}
```

Por último, se puede utilizar pseudocódigo⁸ para especificar los detalles del algoritmo. Es importante que se emplee bastante tiempo en la fase de diseño de sus programas. El resultado final de diseño descendente es una solución que sea fácil de traducir en estructuras de control y estructuras de datos de un lenguaje de programación específico —en nuestro caso, C—

El gasto de tiempo en la fase de **diseño** será ahorro de tiempo cuando se escriba y depura su programa.

2.6.3. Implementación (codificación)

La etapa de *implementación (codificación)* traduce los algoritmos del diseño en un programa escrito en un lenguaje de programación. Los algoritmos y las estructuras de datos realizadas en pseudocódigo

⁷ Para ampliar sobre este tema de diagramas de estructuras, puede consultar estas obras nuestras: *Fundamentos de programación*, 2.ª edición, McGraw-Hill, 1992; *Problemas de metodología de la programación*, McGraw-Hill, 1992 o bien la obra *Psycal y Turbo Psycal. Un enfoque práctico* de Joyanes, Zabonero y Hermoso en McGraw-Hill, 1995.

⁸ Para consultar el tema del pseudocódigo, véase las obras: *Fundamentos de programación. Algoritmos y estructuras de datos*, 2.ª edición, McGraw-Hill, 1996 de Luis Joyanes y *Fundamentos de programación. Libro de problemas*, McGraw-Hill, 1996 de Luis Joyanes, Luis Rodríguez y Matilde Fernández.

han de traducirse codificados en un lenguaje que entiende la computadora: PASCAL, FORTRAN, COBOL, C, C++, C# o Java.

La codificación cuando un problema se divide en subproblemas, los algoritmos que resuelven cada subproblema (tarea o módulo) deben ser codificados, depurados y probados independientemente.

Es relativamente fácil encontrar un error en un procedimiento pequeño. Es casi imposible encontrar todos los errores de un programa grande, que se codificó y comprobó como una sola unidad en lugar de como una colección de módulos (procedimientos) bien definidos.

Las reglas del sangrado (indentación) y buenos comentarios facilitan la escritura del código. El pseudocódigo es una herramienta excelente que facilita notablemente la codificación.

2.6.4. Pruebas e integración

Cuando los diferentes componentes de un programa se han implementado y comprobado individualmente, el sistema completo se ensambla y se integra.

La etapa de pruebas sirve para mostrar que un programa es correcto. Las pruebas nunca son fáciles. Edgar Dijkstra ha escrito que mientras que las pruebas realmente muestran la presencia de errores, nunca puede mostrar su ausencia. Una prueba con «éxito» en la ejecución significa sólo que no se han descubierto errores en esas circunstancias específicas, pero no se dice nada de otras circunstancias. En teoría el Único modo que una prueba puede mostrar que un programa es correcto si todos los casos posibles se han intentado y comprobado (es lo que se conoce como prueba exhaustiva); es una situación técnicamente imposible incluso para los programas más sencillos. Supongamos, por ejemplo, que se ha escrito un programa que calcule la nota media de un examen. Una prueba exhaustiva requerirá todas las combinaciones posibles de marcas y tamaños de clases; puede llevar muchos años completar la prueba.

La fase de pruebas es una parte esencial de un proyecto de programación. Durante la fase de pruebas se necesita eliminar tantos errores lógicos como pueda. En primer lugar, se debe probar el programa con datos de entrada válidos que conducen a una solución conocida. Si ciertos datos deben estar dentro de un rango, se deben incluir los valores en los extremos finales del rango. Por ejemplo, si el valor de entrada de n cae en el rango de 1 a 10, se ha de asegurar incluir casos de prueba en los que n esté entre 1 y 10. También se deben incluir datos no válidos para comprobar la capacidad de detección de errores del programa. Se han de probar también algunos datos aleatorios y, por último, intentar algunos datos reales.

2.6.5. Verificación

La etapa de pruebas ha de comenzar tan pronto como sea posible en la fase de diseño y continuará a lo largo de la implementación del sistema. Incluso aunque las pruebas son herramientas extremadamente válidas para proporcionar la evidencia de que un programa es correcto y cumple sus especificaciones, es difícil conocer si las pruebas realizadas son suficientes. Por ejemplo, ¿cómo se puede conocer que son suficientes los diferentes conjuntos de datos de prueba o que se han ejecutado todos los caminos posibles a través del programa?

Por esas razones se ha desarrollado un segundo método para demostrar la corrección o exactitud de un programa. Este método, denominado *verificación formal* implica la construcción de pruebas matemáticas que ayudan a determinar si los programas hacen lo que se supone han de hacer. La verificación formal implica la aplicación de reglas formales para mostrar que un programa cumple su especificación: la verificación. La verificación formal funciona bien en programas pequeños, pero es compleja cuando se utiliza en programas grandes. La teoría de la verificación requiere conocimientos matemáticos avanzados y, por otra parte, se sale fuera de los objetivos de este libro; por esta razón sólo hemos constatado la importancia de esta etapa.

La prueba de que un algoritmo es correcto es como probar un teorema matemático. Por ejemplo, probar que un módulo es exacto (correcto) comienza con las precondiciones (axiomas e hipótesis en

matemáticas) y muestra que las etapas del algoritmo conducen a las postcondiciones. La verificación trata de probar con medios matemáticos que los algoritmos son correctos.

Si se descubre un error durante el proceso de verificación, se debe corregir su algoritmo y posiblemente se han de modificar las especificaciones del problema. Un método es utilizar *invariantes* (una condición que siempre es verdadera en un punto específico de un algoritmo) lo que probablemente hará que su algoritmo contenga pocos errores *antes* de que comience la codificación. Como resultado se gastará menos tiempo en la depuración de su programa.

2.6.6. Mantenimiento

Cuando el producto software (el programa) se ha terminado, se distribuye entre los posibles usuarios, se instala en las computadoras y se utiliza (*producción*). Sin embargo, y aunque, *a priori*, el programa funcione correctamente, el software debe ser mantenido y actualizado. De hecho, el coste típico del mantenimiento excede, con creces, el coste de producción del sistema original.

Un sistema de software producirá errores que serán detectados, casi con seguridad, por los usuarios del sistema y que no se descubrieron durante la fase de prueba. La corrección de estos errores es parte del mantenimiento del software. Otro aspecto de la fase de mantenimiento es la mejora del software añadiendo más características o modificando partes existentes que se adapten mejor a los usuarios.

Otras causas que obligarán a revisar el sistema de software en la etapa de mantenimiento son las siguientes: 1) Cuando un nuevo *hardware* se introduce, el sistema puede ser modificado para ejecutarlo en un nuevo entorno; 2) Si cambian las necesidades del usuario, suele ser menos caro y más rápido, modificar el sistema existente que producir un sistema totalmente nuevo. La mayor parte del tiempo de los programadores de un sistema se gasta en el mantenimiento de los sistemas existentes y no en el diseño de sistemas totalmente nuevos. Por esta causa, entre otras, se ha de tratar siempre de diseñar programas de modo que sean fáciles de comprender y entender (legibles) y fáciles de cambiar.

2.6.7. La *obsolescencia*: programas obsoletos

La última etapa en el ciclo de vida del software es la evolución del mismo, pasando por su vida útil hasta su *obsolescencia* o fase en la que el software se queda anticuado y es preciso actualizarlo o escribir un nuevo programa sustitutorio del antiguo.

La decisión de dar de baja un software por obsoleto no es una decisión fácil. Un sistema grande representa una inversión enorme de capital que parece, a primera vista, más barato modificar el sistema existente, en vez de construir un sistema totalmente nuevo. Este criterio suele ser, normalmente, correcto y por esta causa los sistemas grandes se diseñan para ser modificados. Un sistema puede ser productivamente revisado muchas veces. Sin embargo, incluso los programas grandes se quedan obsoletos por caducidad de tiempo al pasar una fecha límite determinada. A menos que un programa grande esté bien escrito y adecuado a la tarea a realizar, como en el caso de programas pequeños, suele ser más eficiente escribir un nuevo programa que corregir el programa antiguo.

2.6.8. Iteración y evolución del software

Las etapas de vida del software suelen formar parte de un ciclo o bucle, como su nombre sugiere y no son simplemente una lista lineal. Es probable, por ejemplo, que durante la fase de mantenimiento tenga que volver a las especificaciones del problema para verificarlas o modificarlas.

Obsérvese en la Figura 2.14 las diferentes etapas que rodean al núcleo: documentación. La documentación no es una etapa independiente como se puede esperar sino que está integrada en todas las etapas del ciclo de vida del software.



Figura 2.14. Etapas del ciclo de vida del software cuyo núcleo aglutinador es la documentación.

2.7. MÉTODOS FORMALES DE VERIFICACIÓN DE PROGRAMAS

Aunque la verificación formal de programas se sale fuera del ámbito de este libro, por su importancia vamos a considerar dos conceptos clave, *asertos* (afirmaciones) y *precondiciones/postcondiciones invariantes* que ayuden a documentar, corregir y clarificar el diseño de módulos y de programas.

2.7.1. Aserciones⁷

Una parte importante de una verificación formal es la documentación de un programa a través de *asertos* o *afirmaciones* —sentencias lógicas acerca del programa que se declaran «verdaderas»—. Un aserto se escribe como un comentario y describe lo que se supone sea verdadero sobre las variables del programa en ese punto.

Un aserto *es* una frase sobre una condición específica en un cierto punto de un **algoritmo** o **programa**.

Ejemplo 2.9

El siguiente fragmento de programa contiene una secuencia de sentencias de asignación, seguidas por un aserto.

```
A = 10;           { aserto: A es 10 }
X = A;           { aserto: X es 10 }
Y = X + A;       { aserto: Y es 20 }
```

⁷ Este término se suele traducir también por *afirmaciones* o *declaraciones*. El término *aserto* está extendido en la jerga informática pero no es aceptado por el DKAE.

La verdad de la primera afirmación {A es 10}, sigue a la ejecución de la primera sentencia con el conocimiento de que 10 es una constante. La verdad de la segunda afirmación { X es 10}, sigue de la ejecución de $X = A$ con el conocimiento de que $A = 10$. La verdad de la tercera afirmación {Y es 20} sigue de la ejecución $Y = X + A$ con el conocimiento de que $X = 10$ y $A = 10$. En este segmento del programa se utilizan afirmaciones como comentarios para documentar el cambio en una variable de programa después que se ejecuta cada sentencia de afirmación.

La tarea de utilizar verificación formal es probar que un segmento de programa cumple su especificación. La afirmación final se llama *postcondición* (en este caso, {Y es 20} y sigue a la presunción inicial o *precondición* (en este caso {10 es una constante}), después que se ejecuta el segmento de programa.

2.7.2. Precondiciones y postcondiciones

Las precondiciones y postcondiciones son afirmaciones sencillas sobre condiciones al principio y al final de los módulos. Una precondición de un procedimiento es una afirmación lógica sobre sus parámetros de entrada; se supone que es verdadera cuando se llama al procedimiento. Una postcondición de un procedimiento puede ser una afirmación lógica que describe el cambio en el estado *del programa* producido por la ejecución del procedimiento; la postcondición describe el efecto de llamar al procedimiento. En otras palabras, la postcondición indica que será verdadera después que se ejecute el procedimiento.

Ejemplo 2.10

```
{Precondiciones y postcondiciones del subprograma LeerEnteros}
subprograma LeerEnteros (Min, Max: Entero; var N: Entero);
{
  Lectura de un entero entre Min y Max en N
  Pre : Min y Max son valores asignados
  Post: devuelve en N el primer valor del dato entre Min y Max
        si Min <= Max es verdadero; en caso contrario
        N no esta definido.
```

La precondición indica que los parámetros de entrada Min y Max se definen antes de que comience la ejecución del procedimiento. La postcondición indica que la ejecución del procedimiento asigna el primer dato entre Min y Max al parámetro de salida siempre que $Min \leq Max$ sea verdadero.

Las precondiciones y postcondiciones son más que un método para resumir acciones de un procedimiento. La declaración de estas condiciones debe ser la primera etapa en el diseño y escritura de un procedimiento. Es conveniente en la escritura de algoritmos de procedimientos, se escriba la cabecera del procedimiento que muestra los parámetros afectados por el procedimiento así como unos comentarios de cabecera que contienen las precondiciones y postcondiciones.

Precondición: Predicado lógico que debe cumplirse al comenzar la ejecución de una operación.

Postcondición: Predicado lógico que debe cumplirse al acabar la ejecución de una operación; siempre que se haya cumplido previamente la precondición correspondiente.

2.7.3. Reglas para prueba de programas

Un medio útil para probar que un programa P hace lo que realmente ha de hacer es proporcionar *aserciones* que expresen las condiciones antes y después de que P sea ejecutada. En realidad las aserciones son como sentencias o declaraciones que pueden ser o bien *verdaderas* o bien *falsas*.

La primera aserción, la *precondición* describe las condiciones que han de ser verdaderas antes de ejecutar P . La segunda aserción, la *postcondición*, describe las condiciones que han de ser verdaderas después de que P se ha ejecutado (suponiendo que la precondición fue verdadera antes). El modelo general es:

```
{precondición} {= condiciones logicas que son verdaderas antes de que P
                 se ejecute}
(postcondición) {= condiciones logicas que son verdaderas
                 despues de que P se ejecute}
```

Ejemplo 2.1 1

El procedimiento *OrdenarSeleccion* (A, m, n) ordena a los elementos del array. $A[m..n]$ en orden descendente. El modelo correspondiente puede escribirse así:

```
{m ≤ n} {precondicion: A ha de tener al menos 1 elemento}
OrdenarSeleccion (A,m,n) {programa de ordenacion a ejecutar}
{A[m] ≥ A[m+1] ≥ ... ≥ A[n]} {postcondicion: elementos de A en orden
                              descendente}
```

Problema 2.2

Encontrar la posición del elemento mayor de una lista con indicación de precondiciones y postcondiciones.

```
int EncontrarMax (int* A,int m,int n)
{
  /* precondicion : m < n
   postcondicion : devuelve posicion elemento mayor en A[m..n] */
  int i, j;

  i = m;
  j = n;      {asercion}
  /* (i = m)^(j = m)^(m < n) */ {^, operador and}
  do {
    i = i + 1;
    if (A[i] > A[j])
      j = i;
  }while (i<n);
  return j;   /*devuelve j como elemento mayor*/
}
```

2.7.4. Invariantes de bucles

Una *invariante de bucle* es una condición que es verdadera antes y después de la ejecución de un bucle. Las invariantes de bucles se utilizan para demostrar la corrección (exactitud) de algoritmos iterativos. Utilizando invariantes, se pueden detectar errores antes de comenzar la codificación y por esa razón reducir tiempo de depuración y prueba.

Ejemplo 2.12

Un bucle que calcula la suma de los n primeros elementos del array (lista)A:

```
{
```

Un invariante es un predicado que cumple tanto antes como después de cada iteración (vuelta) y que describe la misión del bucle.

Invariantes de bucle como herramientas de diseño

Otra aplicación de los invariantes de bucle es la especificación del bucle: iniciación, condición de repetición y cuerpo del bucle.

Ejemplo 2.13

Si la invariante de un bucle es:

```
{invariante : i <= n y Suma es la suma de todos los números leídos del
teclado}
```

Se puede deducir que:

```
Suma = 0.0;           {iniciacion}
i = 0;
i < n                 {condicion/prueba del bucle}
scanf("%d",&Item);
Suma = Suma + Item;  {cuerpo del bucle}
i = i + 1;
```

Con toda esta información es una tarea fácil escribir el bucle de suma

```
Suma = 0.0;
i = 0;
while (i < n) /*i, toma los valores 0,1,2,3,..n-1*/
{
    scanf("%d",&Item);
    Suma = Suma + Item;
    i = i + 1;
}
```

Ejemplo 2.14

En los bucles `for` es posible declarar también invariantes, pero teniendo presente la particularidad de esta sentencia: la variable de control del bucle es indefinida después que se sale del bucle, por lo que

para definir su invariante se ha de considerar que dicha variable de control se incrementa antes de salir del bucle y mantiene su valor final.

```

/*precondicion n >= 1*/
Suma = 0;
for (i=1; i<=n; i=i+1)
{
    /*invariante : i <= n+1 y Suma es 1+2+...i-1*/
    Suma = Suma + i;
}
/*postcondicion: Suma es 1+2+3+...n-1+n*/

```

Problema 2.3

Escribir un bucle controlado por centinela que calcule el producto de un conjunto de datos.

```

/*Calcular el producto de una serie de datos*/
/*precondicion : centinela es constante*/
Producto = 1;
printf ("Para terminar, introduzca %d", Centinela);
puts ("Introduzca numero:");
scanf ("%d", &Numero);
while (Numero != Centinela)
{
    /*invariante: Producto es el producto de todos los valores
    leidos en Numero y ninguno era el Centinela*/
    Producto = Producto * Numero;
    puts ('Introduzca numero siguiente:');
    scanf ("%d", &Numero);
}
/*postcondicion: Producto es el producto de todos los numeros leidos en
Numero antes del centinela*/

```

2.7.5. Etapas a establecer la exactitud (corrección) de un programa

Se pueden utilizar invariantes para establecer la corrección de un algoritmo iterativo. Supongamos el algoritmo ya estudiado anteriormente.

```

/*calcular la suma de A[0], A[2], ...A[n-1]*/
Suma = 0;
j = 0;
while (j <= n-1)
{
    Suma = Suma + A[j];
    j = j+1;
}
/*invariante: Suma es la suma de los elementos A[0] a A[j-1]*/

```

Los siguientes cuatro puntos han de ser verdaderos^x:

1. **El invariante debe ser inicialmente verdadero**, antes de que comience la ejecución por primera vez del bucle. En el ejemplo anterior, Suma es 0 y j es 0 inicialmente. En este caso, el invariante significa que Suma contiene la suma de los elementos A[0] a A[j-1], que es verdad ya que no hay elementos en este rango.

^x Carrasco, Helnian y Verof, *op. cit.*, pág. 15.

2. **Una ejecución del bucle debe mantener el invariante.** Esto es si el invariante es verdadero antes de cualquier iteración del bucle, entonces se debe demostrar que es verdadero después de la iteración. En el ejemplo, el bucle añade $A[j]$ a Suma y a continuación incrementa j en 1. Por consiguiente, después de una ejecución del bucle, el elemento añadido más recientemente a Suma es $A[j-1]$; esto es el invariante que es verdadero después de la iteración.
3. **El invariante debe capturar la exactitud del algoritmo.** Esto es, debe demostrar que si el invariante es verdadero cuando termina el bucle, el algoritmo es correcto. Cuando el bucle del ejemplo termina, j contiene n y el invariante es verdadero: Suma contiene la suma de los elementos $A[0]$ a $A[j-1]$, que es la suma que se trata de calcular.
4. **El bucle debe terminar.** Esto es, se debe demostrar que el bucle termina después de un número finito de iteraciones. En el ejemplo, j comienza en 0 y a continuación se incrementa en 1 en cada ejecución del bucle. Por consiguiente, j eventualmente excederá a n con independencia del valor de n . Este hecho y la característica fundamental de *while* garantizan que el bucle terminará.

La identificación de invariantes de bucles, ayuda a escribir bucles correctos. Se representa el invariante como un comentario que precede a cada bucle. En el ejemplo anterior

```
{Invariante: 0 <= j < N Y Suma = A[0]+...+A[j-1]}
while j <= n-1 do
```

2.7.6. Programación segura contra fallos

Un programa es seguro contra fallos cuando se ejecuta razonablemente por cualquiera que lo utilice. Para conseguir este objetivo se han de comprobar *los errores en datos de entrada* y en la *lógica del programa*.

Supongamos un programa que espera leer datos enteros positivos pero lee -25. Un mensaje típico a visualizar ante este error suele ser:

```
Error de rango
```

Sin embargo, es más útil un mensaje tal como este:

```
-25 no es un número válido de años
Por favor vuelva a introducir el número
```

Otras reglas prácticas a considerar son:

- Comprobar datos de entrada no válidos

```
scanf("%f %d", Grupo, Numero) ;
...
if (Numero >= 0)
    agregar Numero a total
else manejar el error
```

- Cada subprograma debe comprobar los valores de sus parámetros. Así, en el caso de la función SumaIntervalo que suma todos los enteros comprendidos entre m y n .

```
int SumaIntervalo (int m, int n)
```

```
precondicion : m y n son enteros tales que m <= n
postcondicion: Devuelve SumaIntervalo = m+(m+1)+...+n
                m y n son inalterables
```

```

int Suma,Indice;
Suma = 0;
for (Indice= m; Indice<=n ;Indice++)
    Suma = Suma + Indice;
return Suma;
}

```

2.8. FACTORES EN LA CALIDAD DEL SOFTWARE

La construcción de software requiere el cumplimiento de numerosas características. Entre ellas se destacan las siguientes:

Eficiencia

La eficiencia de un software es su capacidad para hacer un buen uso de los recursos que manipula.

Transportabilidad (portabilidad)

La *transportabilidad o portabilidad* es la facilidad con la que un software puede ser transportado sobre diferentes sistemas físicos o lógicos.

Verificabilidad

La verificabilidad —facilidad de verificación de un software— es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.

Integridad

La integridad es la capacidad de un software a proteger sus propios componentes contra los procesos que no tenga el derecho de acceder.

Fácil de utilizar

Un *software* es fácil de utilizar si se puede comunicar consigo de manera cómoda.

Corrección

Capacidad de los productos *software* de realizar exactamente las tareas definidas por su especificación.

Robustez

Capacidad de los productos software de funcionar incluso en situaciones anormales.

Extensibilidad

Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esta característica:

- diseño simple;
- descentralización.

Reutilización

Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

Compatibilidad

Facilidad de los productos para ser combinados con otros.

2.9. RESUMEN

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. *Análisis del programa.*
2. *Diseño del algoritmo.*
3. *Codificación.*
4. *Compilación y ejecución.*
5. *Verificación y mantenimiento.*
6. *Documentación y mantenimiento.*

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento suce-

sivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

1. **Secuenciales:** las instrucciones se ejecutan sucesivamente una después de otra.
2. **Repetitivas:** una serie de instrucciones se repiten una y otra vez hasta que se cumple una cierta condición.
3. **Selectivas:** permite elegir entre dos alternativas (dos conjuntos de instrucciones) dependiendo de una condición determinada.

2.10. EJERCICIOS

21. Diseñar una solución para resolver cada uno de los siguientes problemas y trate de refinar sus soluciones mediante algoritmos adecuados:
 - a) Realizar una llamada telefónica desde un teléfono público.
 - b) Cocinar una tortilla.
 - c) Arreglar un pinchazo de una bicicleta.
 - d) Freír un huevo.
22. Escribir un algoritmo para:
 - a) Sumar dos números enteros.
 - b) Restar dos números enteros.
 - c) Multiplicar dos números enteros.
 - d) Dividir un número entero por otro.
23. Escribir un algoritmo para determinar el máximo común divisor de dos números enteros (MCD) por el algoritmo de Euclides:
 - Dividir el mayor de los dos enteros positivos por el más pequeño.
 - A continuación dividir el divisor por el resto.
 - Continuar el proceso de dividir el último divisor por el último resto hasta que la división sea exacta.
 - El último divisor es el mcd.
24. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Visualizar el número de valores leídos.
25. Diseñar un algoritmo que imprima y sume la serie de números 3, 6, 9, 12, ..., 99.
26. Escribir un algoritmo que lea cuatro números y a continuación imprima el mayor de los cuatro.
27. Diseñar un algoritmo que lea tres números y encuentre si uno de ellos es la suma de los otros dos.
28. Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada consistirá en parejas de números (*minutos, segundos*) que dan el tiempo del corredor; por cada corredor, el algoritmo debe imprimir el tiempo en minutos y segundos así como la velocidad media.

Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.
29. Diseñar un algoritmo para determinar si un número N es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad.)
- 2.10. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura ($S = 1/2 \text{ Base} \times \text{Altura}$).
- 2.11. Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

- 2.12. Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.
- 2.13. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como «*radar*».
- 2.14. Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. For ejemplo, «*Mortimer*» contiene dos «*m*», una «*o*», dos «*r*», una «*y*», una «*t*» y una «*e*».
- 2.15. Muchos bancos y cajas de **ahorro** calculan los intereses de las cantidades depositadas por los clientes diariamente en base a las siguientes premisas. Un capital de **1.000** pesetas, con una tasa de interés del 6 por 100, renta un interés en un día de 0,06 multiplicado por 1.000 y dividido por **365**. Esta operación producirá 0,16 pesetas de interés y el capital acumulado será **1.000,16**. El interés para el segundo día se calculará multiplicando 0,06 por 1.000 y dividiendo el resultado por **365**. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcule el capital total acumulado al final del período de tiempo especificado.

2.11. EJERCICIOS RESUELTOS

Desarrolle los algoritmos que resuelvan los siguientes problemas:

2.1. Ir al cine.

Análisis del problema

DATOS DE SALIDA: Ver la película.
 DATOS DE ENTRADA: Nombre de la película, dirección de la sala, hora de proyección.
 DATOS AUXILIARES: Entrada, número de asiento.

Para solucionar el problema, se debe seleccionar una película de la cartelera del periódico, ir a la sala y comprar la entrada para, finalmente, poder ver la película.

Diseño del algoritmo

```

inicio
    < seleccionar la película >
    tomar el periódico
    mientras no lleguemos a la cartelera
        pasar la hoja
    mientras no se acabe la cartelera
        leer la película
        si nos gusta, recordarla
        elegir una de las películas seleccionadas
        leer la dirección de la sala y la hora de proyección
    
```

```

    < comprar la entrada >
    trasladarse a la sala
    si no hay entradas, ir a fin
    si hay cola
        ponerse el último
        mientras no lleguemos a la taquilla
            avanzar
            si no hay entradas, ir a fin
        comprar la entrada
    < ver la película >
    leer el número de asiento de la entrada
    buscar el asiento
    sentarse
    ver la película
    fin.
    
```

2.2. Comprar una entrada para ir a los toros.

Análisis del problema

DATOS DE SALIDA: La entrada.
 DATOS DE ENTRADA: Tipo de entrada (sol, sombra, tendido, andanada...).
 DATOS AUXILIARES: Disponibilidad de la entrada.

Hay que ir a la taquilla y elegir la entrada deseada. Si hay entradas se compra (en taquilla o a los reventas). Si no la hay, se puede seleccionar otro tipo de entrada o desistir, repitiendo esta acción hasta que se ha conseguido la entrada o el posible comprador ha desistido.

Diseño del algoritmo

```

inicio
  ir a la taquilla
  si no hay entradas en taquilla
    si nos interesa comprarla en la
      reventa
        ir a comprar la entrada
    si no ir a fin
  < comprar la entrada >
  seleccionar sol o sombra
  seleccionar barrera, tendido,
    andanada o palco
  seleccionar número de asiento
  solicitar la entrada
  si la tienen disponible
    adquirir la entrada
    si no
      si queremos otro tipo de
        entrada
        ir a comprar la entrada
  fin.

```

23. Hacer una taza de té.

DATOS DE SALIDA: taza de té.
DATOS DE ENTRADA: bolsa de té, agua.
DATOS AUXILIARES: pitido de la tetera, aspecto de la infusión.

Después de echar agua en la tetera, se pone al fuego y se espera a que el agua hierva (hasta que suena el pitido de la tetera). Introducimos el té y se deja un tiempo hasta que esté hecho.

Diseño del algoritmo

```

inicio
  tomar la tetera
  llenarla de agua
  encender el fuego
  poner la tetera en el fuego
  mientras no hierva el agua
    esperar
  tomar la bolsa de té
  introducirla en la tetera
  mientras no está hecho el té
    esperar
  echar el té en la taza
fin.

```

24. Hacer una llamada telefónica. Considerar los casos: a) llamada manual con operador; b) llamada automática; c) llamada a cobro revertido.

Análisis del problema

Para decidir el tipo de llamada que se efectuará, primero se debe considerarse si se dispone de efectivo o no

para realizar la llamada a cobro revertido. Si hay efectivo se debe ver si el lugar a donde vamos a llamar está conectado a la red automática o no.

Para una llamada con operadora hay que llamar a la centralita y solicitar la llamada, esperando hasta que se establezca la comunicación. Para una llamada automática se leen los prefijos del país y provincia si fuera necesario, y se realiza la llamada, esperando hasta que cojan el teléfono. Para llamar a cobro revertido se debe llamar a centralita, solicitar la llamada y esperar a que el abonado del teléfono d que se llama dé su autorización, con lo que establecerá la comunicación.

Como datos de entrada tendríamos las variables que nos van a condicionar el tipo de llamada, el número de teléfono y, en caso de llamada automática, los prefijos si los hubiera. Como dato auxiliar se podría considerar en los casos a) y c) el contacto con la centralita.

Diseño del algoritmo

```

inicio
  si tenemos dinero
  si podemos hacer una llamada
    automática
    Leer el prefijo de país y local-
      lidad
    marcar el número
  si no
    < llamada manual >
    llamar a la centralita
    solicitar la comunicación
  mientras no contesten
    esperar
  establecer comunicación
  si no
    < realizar una llamada a cobro
    revertido >
    llamar a la centralita
    solicitar la llamada
    esperar hasta tener la autori-
      zación
    establecer comunicación
fin.

```

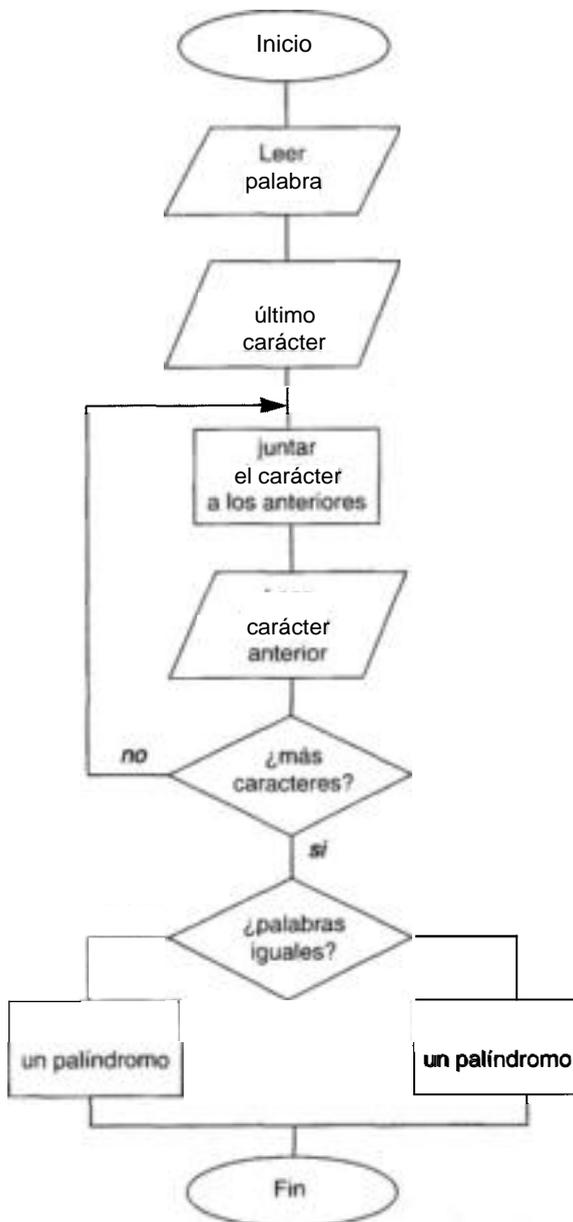
2.5. Averiguar si una palabra es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, como, por ejemplo, «radar»

Análisis del problema

DATOS DE SALIDA: el mensaje que nos dice si es o no un palíndromo.
DATOS DE ENTRADA: palabra.
DATOS AUXILIARES: cada carácter de la palabra, palabra al revés.

Para comprobar si una palabra es un palíndromo, se puede ir formando una palabra con los *caracteres* invertidos con respecto a la original y comprobar si la palabra al revés es igual a la original. Para obtener esa palabra al revés, *se leerán* en sentido inverso los *caracteres* de la palabra inicial *y se irán* juntando sucesivamente *hasta* llegar al *primer* carácter.

Diseño del algoritmo



26. Diseñar un algoritmo **para** calcular la velocidad (en metros/segundo) de los corredores de una carrera de 1.500 metros. La entrada serán parejas de números (minutos, segundos) que darán el tiempo de cada corredor. Por cada corredor **se** imprimirá el tiempo en minutos y segundos, así como la velocidad media. El bucle **se** ejecutará hasta que demos una entrada de 0,0 que será la marca de fin de entrada de datos.

Análisis del problema

DATOS DE SALIDA: v (velocidad media).
DATOS DE ENTRADA: mm, ss (minutos y segundos).

DATOS AUXILIARES: distancia (distancia recorrida, que en el ejemplo es de 1.500 metros) y tiempo (los minutos y los segundos que ha tardado en recorrerla).

Se debe efectuar un bucle hasta que mm sea 0 y ss sea 0. Dentro del bucle se calcula el tiempo en segundos con la fórmula $\text{tiempo} = \text{ss} + \text{mm} * 60$. La velocidad **se** hallará con la fórmula $\text{velocidad} = \text{distancia} / \text{tiempo}$.

Diseño del algoritmo

```

inicio
  distancia ← 1500
  leer (mm, ss)
  mientras mm = 0 y ss = 0 hacer
    tiempo ← ss + mm * 60
    v ← distancia / tiempo
    escribir (mm, ss, v)
    leer (mm, ss)
  fin
  
```

27. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura.

Análisis del problema

DATOS DE SALIDA: s (superficie).
DATOS DE ENTRADA: b (base) a (altura).

Para calcular la superficie se aplica la fórmula $S = \text{base} * \text{altura} / 2$.

Diseño del algoritmo

```

inicio
  leer (b, a)
  s = b * a / 2
  escribir (s)
fin
  
```

2.8. Realizar un algoritmo que calcule la suma de los enteros entre 1 y 10, es decir, $1+2+3+\dots+10$.

Análisis del problema

DATOS DE SALIDA: suma (contiene la suma requerida).

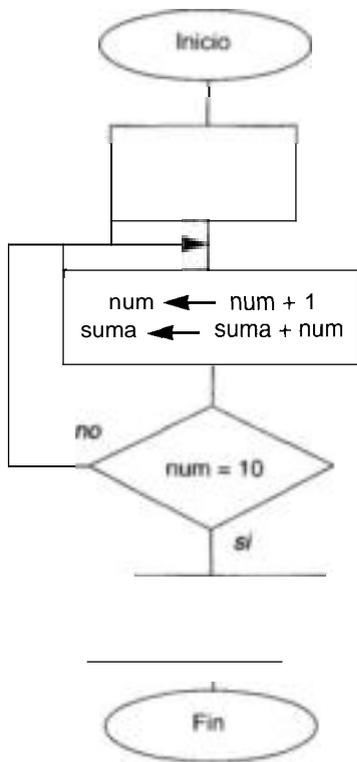
DATOS AUXILIARES: num (será una variable que vaya tomando valores entre 1 y 10 y se acumulará en suma).

Hay que ejecutar un bucle que se realice 10 veces. En él se irá incrementando en 1 la variable num, y se acumulará su valor en la variable suma. Una vez salgamos del bucle se visualizará el valor de la variable suma.

Diseño del algoritmo

TABLA DE VARIABLES

entero: suma, num



2.9. Realizar un algoritmo que calcule y visualice las potencias de 2 entre 0 y 10.

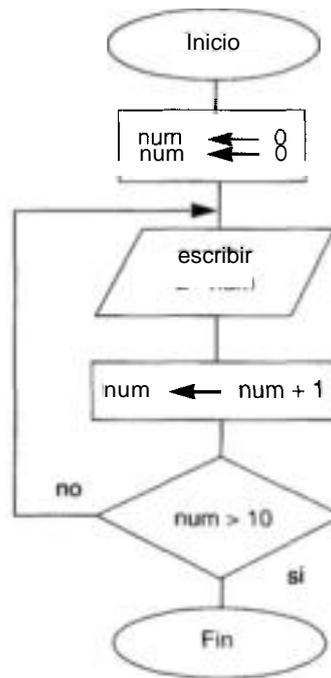
Análisis del problema

Hay que implementar un bucle que se ejecute once veces y dentro de él ir incrementando una variable que tome valores entre 0 y 10 y que se llamará num. También dentro de él se visualizará el resultado de la operación 2^{num} .

Diseño del algoritmo

TABLA DE VARIABLES:

entero: num



CAPÍTULO 3

EL LENGUAJE C: ELEMENTOS BÁSICOS

CONTENIDO

- 3.1. Estructura general de un programa en C.
- 3.2. Creación de un programa.
- 3.3. El proceso de ejecución de un programa en C.
- 3.4. Depuración de un programa en C.
- 3.b. Pruebas.
- 3.6. Los elementos de un programa en C.
- 3.7. Tipos de datos en C.
- 3.8. El tipo de dato lógico.
- 3.9. Constantes.
- 3.10. Variables.
- 3.11. Duración de una variable.
- 3.12. Entradas y salidas.
- 3.13. *Resumen.*
- 3.14. *Ejercicios.*

INTRODUCCIÓN

Una vez que se le ha enseñado a crear sus propios programas, vamos a analizar los fundamentos del lenguaje de programación C. Este capítulo comienza con un repaso de los conceptos teóricos y prácticos relativos a la estructura de un programa enunciados en capítulos anteriores, dada su gran importancia en el desarrollo de aplicaciones, incluyendo además los siguientes temas:

- creación de un programa;
- elementos básicos que componen un programa;
- tipos de datos en C y cómo se declaran;
- concepto de constantes y su declaración;
- concepto y declaración de variables;
- tiempo de vida o duración de variables;
- operaciones básicas de entrada/salida.

CONCEPTOS CLAVE

- Archivo de cabecera.
- *Código* ejecutable.
- Código fuente.
- Código objeto.
- Comentarios.
- Constantes.
- `char`.
- Directiva `#include`.
- `Float/double`.
- Flujos.
- Función `main()`.
- Identificador.
- `int`.
- Preprocesador.
- `printf()`.
- `scanf()`.
- Variables.

3.1. ESTRUCTURA GENERAL DE UN PROGRAMA EN C

En esta sección repasamos los elementos constituyentes de un programa escrito en C, fijando ideas y describiendo ideas nuevas relativas a la mencionada estructura de un programa en C.

Un programa en C se compone de una o más funciones. Una de las funciones debe ser obligatoriamente **main**. Una función en C es un grupo de instrucciones que realizan una o más acciones. Asimismo, un programa contendrá una serie de directivas **#include** que permitirán incluir en el mismo archivos de cabecera que a su vez constarán de funciones y datos predefinidos en ellos.

```
#include <stdio.h>           ← archivo de cabecera  stdio.h

int main()                  ← cabecera defunción
{
    ██████████             ← nombre de la función
    ...                    ← sentencias
}
```

#include	<i>Directivas del preprocesador</i>
#define	Macros del procesador

<i>Declaraciones globales</i>
o prototipos de funciones
o variables

<i>Función principal main</i>
main()
{
<i>declaraciones locales</i>
<i>sentencias</i>
}

<i>Definiciones de otras funciones</i>
tipo1 func1(...)
{
...
}

Figura 3.1. Estructura típica de un programa C.

De un modo más explícito, un programa C puede incluir:

- directivas de preprocesador;
- declaraciones globales;
- la función `main()`;
- funciones definidas por el usuario;
- comentarios del programa (utilizados en su totalidad).

La estructura típica completa de un programa C se muestra en la Figura 3.1. Un ejemplo de un programa sencillo en C.

```
/*Listado DEMO_UNO.C. Programa de saludo */

#include <stdio.h>
/* Este programa imprime: Bienvenido a la programación en C */
int main()
{
    printf("Bienvenido a la programación en C\n");
    return 0;
}
```

La directiva `#include` de la primera línea es necesaria para que el programa tenga salida. *Se* refiere a un archivo externo denominado `stdio.h` en el que se proporciona la información relativa a la función `printf()`. Obsérvese que los ángulos `<` y `>` no son parte del nombre del archivo; se utilizan para indicar que el archivo es un archivo de la biblioteca estándar C.

La segunda línea es un *comentario*, identificado por los caracteres `/*` y `*/`. Los comentarios se incluyen en programas que proporcionan explicaciones a los lectores de los mismos. Son ignorados por el compilador.

La tercera línea contiene la cabecera de la función `main()`, obligatoria en cada programa C. Indica el comienzo del programa y requieren los paréntesis `()` a continuación de `main()`.

La cuarta y séptima línea contienen sólo las llaves `{` y `}` que encierran el cuerpo de la función `main()` y son necesarias en todos los programas C.

La quinta línea contiene la sentencia

```
printf("Bienvenido a la programación en C\n");
```

que indica al sistema que escriba el mensaje "Bienvenido a la programación en C\n".

`printf()` es la función más utilizada para dar salida de datos por el dispositivo estándar, la pantalla. La salida será

```
Bienvenido a la programación en C
```

El símbolo `\n` es el símbolo de *nueva línea*. Poniendo este símbolo al final de la cadena entre comillas, indica al sistema que comience una nueva línea después de imprimir los caracteres precedentes, terminando, por consiguiente, la línea actual.

La sexta línea contiene la sentencia `return 0`. Esta sentencia termina la ejecución del programa y devuelve el control al sistema operativo de la computadora. El número 0 se utiliza para señalar que el programa ha terminado correctamente (*con éxito*).

Obsérvese el punto y coma (`;`) al final de la quinta y sexta línea. C requiere que cada sentencia termine con un punto y coma. No es necesario que esté al final de una línea. Se pueden poner varias sentencias en la misma línea y se puede hacer que una sentencia se extienda sobre varias líneas.

Advertencia

- El programa más corto de C es el «programavació» que no hace nada.
- La sentencia `return 0`; no es obligatoria en la mayoría de los compiladores, aunque algunos emiten un mensaje de advertencia si se omite.

3.1.1. Directivas del preprocesador

El *preprocesador* en un programa C se puede considerar como un editor de texto inteligente que consta de *directivas* (instrucciones al compilador antes de que se compile el programa principal). Las dos directivas más usuales son `#include` y `#define`.

Todas las directivas del preprocesador comienzan con el signo de libro o «*almohadilla*» (`#`), que indica al compilador que lea las directivas antes de compilar la parte (función) principal del programa.

Las *directivas* son instrucciones al compilador. Las directivas no son generalmente sentencias —obsérvese que su línea no termina en punto y coma—, sino instrucciones que se dan al compilador antes de que el programa se compile. Aunque las directivas pueden definir macros, nombres de constantes, archivos fuente adicionales, etc., su uso más frecuente en C es la inclusión de archivos de cabecera.

Existen archivos de cabecera estándar que se utilizan ampliamente, tales como `STDIO.H`, `STDLIB.H`, `MATH.H`, `STRING.H` y se utilizarán otros archivos de cabecera definidos por el usuario para diseño estructurado.

La directiva `#include` indica al compilador que lea el archivo fuente que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva. Estos archivos se denominan *archivos de cubecera o archivos de inclusión*.

Los *archivos de cabecera* (archivos con extensión `.h` contienen código fuente C) se sitúan en un programa C mediante la directiva del preprocesador `#include` con una instrucción que tiene el siguiente formato :

```
#include <nombrearch.h>    o bien    #include "nombrearch.h"
```

`nombrearch` debe ser un archivo de texto **ASCII** (su archivo fuente) que reside en su disco. En realidad, la directiva del preprocesador mezcla un archivo de disco en su programa fuente.

La mayoría de los programadores C sitúan las directivas del preprocesador al principio del programa, aunque esta posición no es obligatoria.

Además de los archivos de código fuente diseñados por el usuario, `#include` se utiliza para incluir archivos de sistemas especiales (también denominados archivos de cabecera) que residen en su compilador C. Cuando se instala el compilador, estos archivos de cabecera se almacenarán automáticamente en su disco, en el directorio de inclusión (`include`) del sistema. Sus nombres de archivo siempre tienen la extensión `.h`.

El archivo de cabecera más frecuente es `STDIO.H`. Este archivo proporciona al compilador C la información necesaria sobre las funciones de biblioteca que realizan operaciones de entrada y salida.

Como casi todos los programas que escriba imprimirán información en pantalla y leerán datos de teclado, necesitarán incluir `scanf()` y `printf()` en los mismos.

Para ello será preciso que cada programa contenga la línea siguiente:

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

De igual modo es muy frecuente el uso de funciones de cadena, especialmente `strcpy()`; por esta razón, se requiere el uso del archivo de cabecera denominado `string.h`. Por consiguiente, será muy usual que deba incluir en sus programas las líneas:

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

El orden de sus archivos de inclusión no importan con tal que se incluyan antes de que se utilicen las funciones correspondientes. La mayoría de los programas C incluyen todos los archivos de cabecera necesarios antes de la primera función del archivo.

La directiva `#include` puede adoptar uno de los siguientes formatos:

```
#include <nombre del archivo>
#include "nombre del archivo"
```

Dos ejemplos típicos son:

```
(a) #include <stdio.h>
(b) #include "pruebas.h"
```

El formato (a) (el nombre del archivo entre ángulos) significa que los archivos se encuentran en el directorio por defecto `include`. El formato (b) significa que el archivo está en el directorio *actual*. Los dos métodos no son excluyentes y pueden existir en el mismo programa archivos de cabecera estándar utilizando ángulos y otros archivos de cabecera utilizando comillas. Si desea utilizar un archivo de cabecera que se creó y no está en el directorio por defecto, se encierra el archivo de cabecera y el camino entre comillas, tal como

```
#include "D:\MIPROG\CABEZA.H"
```

#define. La directiva `#define` indica al preprocesador que defina un ítem de datos u operación para el programa C. Por ejemplo, la directiva

```
#define TAM_LINEA 65
```

sustituirá `TAM-LINEA` por el valor 65 cada vez que aparezca en el programa.

3.1.2. Declaraciones globales

Las *declaraciones globales* indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones de su programa. Las declaraciones globales se sitúan antes de la función `main()`. Si se declara global una variable `Grado-clase` del tipo

```
int Grado-clase;
```

cualquier función de su programa, incluyendo `main()`, puede acceder a la variable `Grado-clase`.

La zona de declaraciones globales de un programa puede incluir declaraciones de variables además de declaraciones de función. Las declaraciones de función se denominan *prototipos*

```
int media(int a, int b);
```

El siguiente programa es una estructura modelo que incluye declaraciones globales.

```
/* Programa demo.C */
#include <stdio.h>

/* Definición de macros */
#define MICONST1 0.50
#define MICONS2 0.75

/* Declaraciones globales */
int Calificaciones ;

main()
{
    ...
}
```

3.1.3. Función `main()`

Cada programa C tiene una función `main()` que es el punto de entrada al programa. Su estructura es:

```

main()
{
    ... ← bloque de sentencias
}

```

Las sentencias incluidas entre las llaves { ... } se denominan *bloque*. Un programa debe tener sólo una función `main()`. Si se intenta hacer dos funciones `main()` se produce un error. Además de la función `main()`, un programa C consta de una colección de funciones.

Una función C es un subprograma que devuelve un único valor, un conjunto de valores o realiza alguna tarea específica tal como E/S.

En un programa corto, el programa completo puede incluirse totalmente en la función `main()`. Un programa largo, sin embargo, tiene demasiados códigos para incluirlo en esta función. La función `main()` en un programa largo consta prácticamente de llamadas a las funciones definidas por el usuario. El programa siguiente se compone de tres funciones: `obtenerdatos()`, `alfabetizar()` y `verpalabras()` que se invocan sucesivamente.

```

int main()

    obtenerdatos();

    alfabetizar();

    verpalabras();

    return 0;
}

```

Las variables y constantes *globales* se declaran y definen fuera de la definición de las funciones, generalmente en la cabecera del programa, antes de `main()`, mientras que las variables y constantes *locales* se declaran y definen en la cabecera del cuerpo o bloque de la función principal, o en la cabecera de cualquier bloque. Las sentencias situadas en el interior del cuerpo de la función `main()`, o cualquier otra función, deben terminar en punto y coma.

3.1.4. Funciones definidas por el usuario

Un programa C es una colección de funciones. Todos los programas se construyen a partir de una o más funciones que se integran para crear una aplicación. Todas las funciones contienen una o más sentencias C y se crean generalmente para realizar una única tarea, tales como imprimir la pantalla, escribir un archivo o cambiar el color de la pantalla. Se pueden declarar y ejecutar un número de funciones casi ilimitado en un programa C.

Las funciones definidas por el usuario se invocan por su nombre y los parámetros opcionales que puedan tener. Después de que la función es llamada, el código asociado con la función se ejecuta y, a continuación, se retorna a la función llamadora.

Todas las funciones tienen nombre y una lista de valores que reciben. Se puede asignar cualquier nombre a su función, pero normalmente se procura que dicho nombre describa el propósito de la función. En C, las funciones requieren una *declaración o prototipo* en el programa:

```

void trazarcuerva();

```

Una **declaración de función** indica al compilador el nombre de la función por el que ésta será invocada en el programa. Si la función no se define, el compilador informa de un error. La palabra reservada `void` significa que la función no devuelve un valor.

```
void contarvocales(char caracter);
```

La definición de una función es la estructura de la misma:

```
tipo-retorno nombre-función(lista_de_parámetros) principio de la función
{
    sentencias                                 cuerpo de la función
    return;                                     retorno de la función
}
```

tipo-retorno Es el tipo de valor, o `void`, devuelto por la función
nombre-función Nombre de la función
lista_de_parámetros Lista de **parámetros**, o `void`, pasados a la función. Se conoce también como **argumentos de la función** o argumentos formales.

C proporciona también funciones predefinidas que se denominan **funciones de biblioteca**. Las funciones de biblioteca son funciones listas para ejecutar que vienen con el lenguaje C. Requieren la inclusión del archivo de cabecera estándar, tal como `STDIO.H`, `MATH.H`, etc. Existen centenares de funciones definidas en diversos archivos de cabecera.

```
/* ejemplo funciones definidas por el usuario */

#include <stdio.h>

void visualizar();
#int main()
{
    visualizar(j);
    return 0;
}

void visualizar()
{
    printf( "primeros pasos en C\n");
}
```

Los programas C constan de un conjunto de funciones que normalmente están controladas por la función `main()`.

```
main()
{
    ...
}

obtenerdatos()
{
    ...
}

alfabetizar()
{
    ...
}
```

3.1.5. Comentarios

Un *comentario* es cualquier información que se añade a su archivo fuente para proporcionar documentación de cualquier tipo. El compilador ignora los comentarios, no realiza ninguna tarea concreta. El uso de comentarios es totalmente opcional, aunque dicho uso es muy recomendable.

Generalmente, se considera buena práctica de programación comentar su archivo fuente tanto como sea posible, al objeto de que usted mismo y otros programadores puedan leer fácilmente el programa con el paso de tiempo. Es buena práctica de programación comentar su programa en la parte superior de cada archivo fuente. La información que se suele incluir es el nombre del archivo, el nombre del programador, una breve descripción, la fecha en que se creó la versión y la información de la revisión.

Los comentarios en C estándar comienzan con la secuencia `/*` y terminan con la secuencia `*/`. Todo el texto situado entre las dos secuencias es un comentario ignorado por el compilador.

```
/* PRUEBA1.C - Primer programa C */
```

Si se necesitan varias líneas de programa se puede hacer lo siguiente:

```
/*
Programa           : PRUEBA1.C
Programador        : Pepe Mortimer
Descripción        : Primer programa C
Fecha creación     : Septiembre 2000
Revisión           : Ninguna
*/
```

También se pueden situar comentarios de la forma siguiente:

```
scanf("%d",&x);          /* sentencia de entrada de un valor entero*/
```

Ejemplo 3.1

Supongamos que se ha de imprimir su nombre y dirección muchas veces en su programa C. El sistema normal es teclear las líneas de texto cuantas veces sea necesario; sin embargo, el método más rápido y eficiente sería escribir el código fuente correspondiente una vez y a continuación grabar un archivo *MIDIREC.C*, de modo que para incluir el código sólo necesitará incluir en su programa la línea

```
#include "midirec.c"
```

Es decir, teclee las siguientes líneas y grábelas en un archivo denominado *MIDIREC.C*

```
/* archivo midirec.c */
printf( "Luis Joyanes Aguilar\n");
printf( "Avda de Andalucía, 48\n");
printf( "Carchelejo, JAEN\n");
printf( "Andalucía, ESPAÑA\n");
```

El programa siguiente:

```
/* nombre del archivo demoincl.c,
   ilustra el uso de #include
*/

#include <stdio.h>

int main()
{
    #include "midirec.c"
```

```

    return 0;
}
equivale a
/* nombre del archivo demoincl.c
   ilustra el uso de #include
*/

#include <stdio.h>

int main()

{

    printf( "Luis Joyanes Aguilar\n" );
    printf( "Avda de Andalucía, 48\n" );
    printf( 'Carchelejo,JAEN\n' );
    printf( "Andalucía, ESPAÑA\n" );
    return 0;
}

```

Ejemplo 3.2

El siguiente programa copia un mensaje en un array de caracteres y lo imprime en la pantalla. Ya que `printf()` y `strcpy()` (una función de cadena) se utilizan, se necesitan sus archivos de cabecera específicos.

```

/* nombre del archivo demoinc2.c
   utiliza dos archivos de cabecera
*/
#include <stdio.h>
#include <string.h>

int main0
{
    char mensaje[20];
    strcpy (mensaje, "Atapuerca\n");
    /* Las dos líneas anteriores también se pueden sustituir por
       char mensaje[20] = "Atapuerca\n";
    */
    printf(mensaje);
    return 0;
}

```

Los archivos de cabecera en C tienen normalmente una extensión `.h` y los archivos fuente, la extensión `.c`.

3.2. CREACIÓN DE UN PROGRAMA

Una vez creado un programa en C como el anterior, se debe ejecutar. ¿Cómo realizar esta tarea? Los pasos a dar dependerán del compilador C que utilice. Sin embargo, serán similares a los mostrados en la Figura 3.2. En general, los pasos serían:

- Utilizar un editor de texto para escribir el programa y grabarlo en un archivo. Este archivo constituye el código fuente de un programa.
- Compilar el código fuente. Se traduce el código fuente en un *código objeto* (extensión .obj) (lenguaje máquina entendible por la computadora). Un *archivo objeto* contiene instrucciones en lenguaje máquina que se pueden ejecutar por una computadora. Los archivos estándar C y los de cabecera definidos por el usuario son incluidos (`#include`) en su código fuente por el preprocesador. Los archivos de cabecera contienen información necesaria para la compilación, como es el caso de *stdio.h* que contiene información `scanf()` y de `printf()`.
- Enlazar el código objeto con las *bibliotecas* correspondientes. Una biblioteca C contiene código objeto de una colección de rutinas o *funciones* que realizan tareas, como visualizar informaciones en la pantalla o calcular la raíz cuadrada de un número. El enlace del código objeto del programa con el objeto de las funciones utilizadas y cualquier otro código empleado en el enlace, producirá un código *ejecutable*. Un programa C consta de un número diferente de archivos objeto y archivos biblioteca.

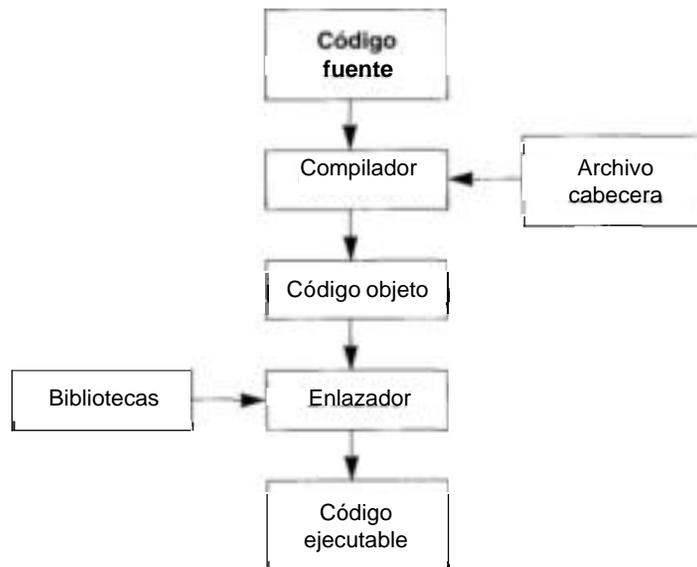


Figura 3.2. Etapas de creación de un programa

Para crear un programa se utilizan las siguientes etapas:

1. Definir su programa.
2. Definir directivas del preprocesador.
3. Definición de declaraciones globales.
4. Crear `main()`.
5. Crear el cuerpo del programa.
6. Crear sus propias funciones definidas por el usuario.
7. Compilar, enlazar, ejecutar y comprobar su programa.
8. Utilizar comentarios.

3.3. EL PROCESO DE EJECUCIÓN DE UN PROGRAMA EN C

Un programa de computadora escrito en un lenguaje de programación (por ejemplo, C) tiene forma de un texto ordinario. Se escribe el programa en una hoja de papel y a este programa se le denomina *programa texto* o *código fuente*. Considérese el ejemplo sencillo:

```
#include <stdio.h>
int main()
{
    printf("Longitud de circunferencia de radio 5: %f",2*3.1416*5);
    return 0;
}
```

La primera operación en el proceso de ejecución de un programa es introducir las sentencias (instrucciones) del programa con un editor de texto. El editor almacena el texto y debe proporcionarle un nombre tal como `area.c`. Si la ventana del editor le muestra un nombre tal como `noname.c`, es conveniente cambiar dicho nombre (por ejemplo, por `area.c`). A continuación se debe guardar el texto en disco para su conservación y uso posterior, ya que en caso contrario el editor sólo almacena el texto en memoria central (RAM) y cuando se apague la computadora, o bien ocurra alguna anomalía, se perderá el texto de su programa. Sin embargo, si el texto del programa se almacena en un disquete, en un disco duro, o bien en un CD-ROM, el programa se guardará de modo permanente, incluso después de apagar la computadora y siempre que ésta se vuelva a arrancar.

La Figura 3.3 muestra el método de edición de un programa y la creación del programa en un disco, en un archivo que se denomina *archivo de texto* (*archivofuente*). Con la ayuda de un editor de texto se puede editar el texto fácilmente, es decir, cambiar, mover, cortar, pegar, borrar texto. Se puede ver, normalmente, una parte del texto en la pantalla y se puede marcar partes del texto a editar con ayuda de un ratón o el teclado. El modo de funcionamiento de un editor de texto y las órdenes de edición asociadas varían de un sistema a otro.

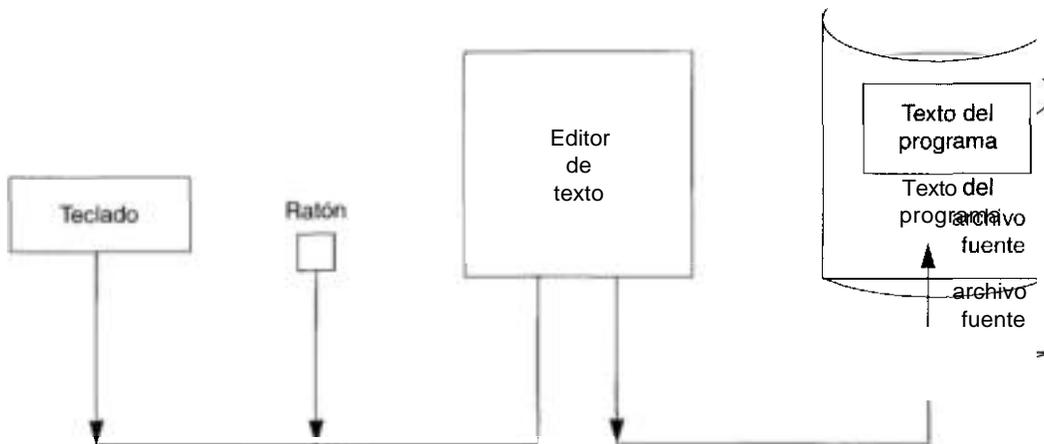


Figura 3.3. Proceso de edición de un archivo fuente.

Una vez editado un programa, se le proporciona un nombre. Se suele dar una extensión al nombre (normalmente `.c`, aunque en algunos sistemas puede tener otros sufijos).

La siguiente etapa es la de compilación. En ella se traduce el código fuente escrito en lenguaje C a código máquina (entendible por la computadora). El programa que realiza esta traducción se llama *compilador*. Cada compilador se construye para un determinado lenguaje de programación (por ejemplo C); un compilador puede ser un programa independiente (como suele ser el caso de sistemas operativos como VMS, UNIX, etc.) o bien formar parte de un programa entorno integrado de desarrollo (**EID**). Los programas EID contienen todos los recursos que se necesitan para desarrollar y ejecutar un programa, por ejemplo, editores de texto, compiladores, enlazadores, navegadores y depuradores.

Cada lenguaje de programación tiene unas reglas especiales para la construcción de programas que se denomina *sintaxis*. El compilador lee el programa del archivo de texto creado anteriormente y comprueba que el programa sigue las reglas de sintaxis del lenguaje de programación. Cuando se

compila su programa, el compilador traduce el código fuente C (las sentencias del programa) en un código máquina (*código objeto*). El código objeto consta de instrucciones máquina e información de cómo cargar el programa en memoria antes de su ejecución. Si el compilador encuentra errores, los presentará en la pantalla. Una vez corregidos los errores con ayuda del editor se vuelve a compilar sucesivamente hasta que no se produzcan errores.

El código objeto así obtenido se almacena en un archivo independiente, normalmente con extensión `.obj` o bien `.o`. Por ejemplo, el programa `area` anterior, se puede almacenar con el nombre `area.obj`.

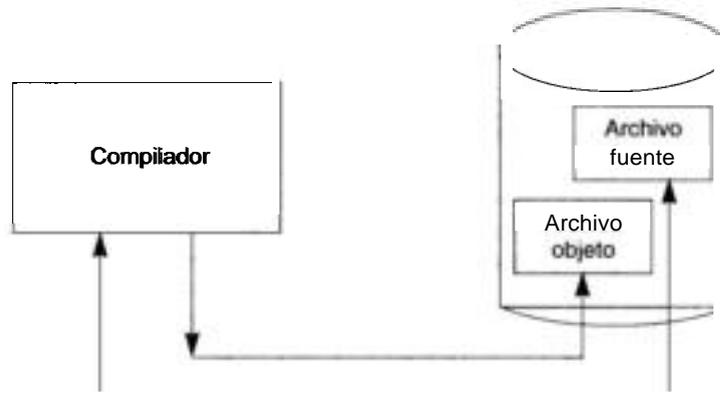


Figura 3.4. Proceso de edición de un archivo fuente.

El archivo objeto contiene sólo la traducción del código fuente. Esto no es suficiente para ejecutar realmente el programa. Es necesario incluir los archivos de biblioteca (por ejemplo, en el programa `area.c`, `stdio.h`). Una biblioteca es una colección de código que ha sido programada y traducida y lista para utilizar en su programa.

Normalmente un programa consta de diferentes unidades o partes de programa que se han compilado independientemente. Por consiguiente, puede haber varios archivos objetos. Un programa especial llamado *enlazador* toma el archivo objeto y las partes necesarias de la biblioteca del sistema y construye un *archivo ejecutable*. Los archivos ejecutables tienen un nombre con la extensión `.exe` (en el ejemplo, `area.exe` o simplemente `area` según sea su computadora). Este archivo ejecutable contiene todo el código máquinas necesario para ejecutar el programa. Se puede ejecutar el programa escribiendo `area` en el indicador de órdenes o haciendo clic en el icono del archivo.

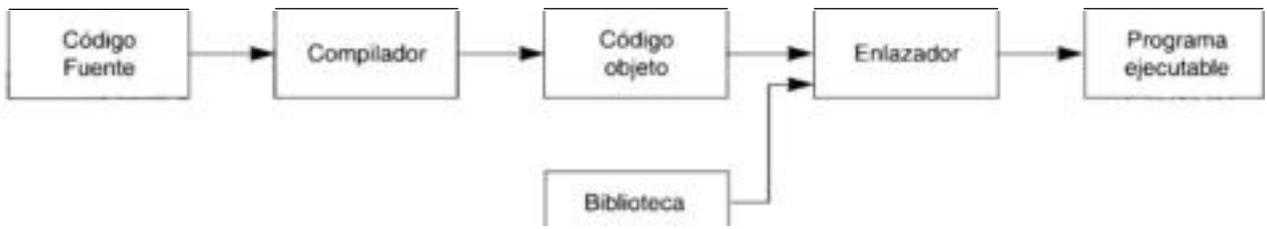


Figura 3.5. Proceso de conversión de código fuente a código ejecutable.

Se puede poner ese archivo en un disquete o en un CD-ROM, de modo que esté disponible después de salir del entorno del compilador a cualquier usuario que no tenga un compilador C o que puede no conocer lo que hace.

El proceso de ejecución de un programa no suele funcionar a la primera vez; es decir, casi siempre hay errores de sintaxis o errores en tiempo de ejecución. El proceso de detectar y corregir errores se denomina *depuración o puesta a punto* de un programa.

La Figura 3.6 muestra el proceso completo de puesta a punto de un programa.

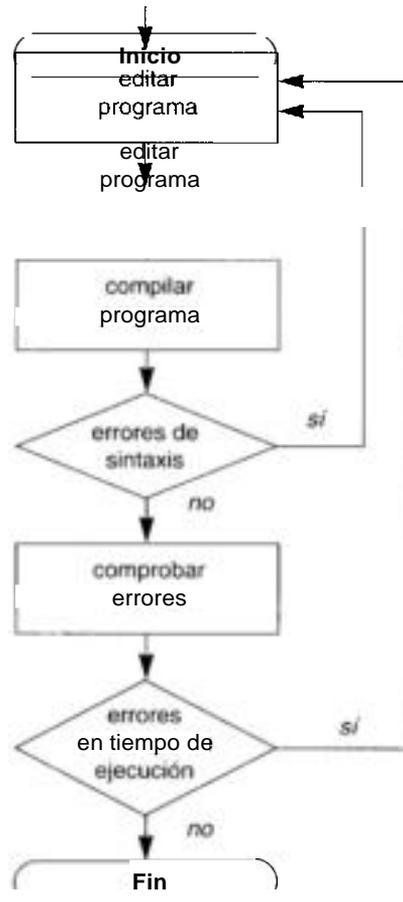


Figura 3.6. Proceso completo de depuración de un programa.

Se comienza escribiendo el archivo fuente con el editor. Se compila el archivo fuente y se comprueban mensajes de errores. Se retorna al editor y se fijan los errores de sintaxis. Cuando el compilador tiene éxito, el enlazador construye el archivo ejecutable. Se ejecuta el archivo ejecutable. Si se encuentra un error, se puede activar el depurador para ejecutar sentencia a sentencia. Una vez que se encuentra la **causa** del error, se vuelve al editor y se repite la compilación. El proceso de compilar, enlazar y ejecutar el programa se repetirá hasta que no se produzcan errores.

Etapas del proceso

- o El código fuente (archivo del programa) se crea con la ayuda del editor de texto.
- o El compilador traduce el archivo texto en un archivo objeto.
- El enlazador pone juntos a diferentes archivos objetos para poner un archivo ejecutable.
- o El sistema operativo pone el archivo ejecutable en la memoria central y se ejecuta el programa.

3.4. DEPURACIÓN DE UN PROGRAMA EN C

Rara vez los programas funcionan bien la primera vez que se ejecutan. Los errores que se producen en los programas han de ser detectados, aislados (fijados) y corregidos. El proceso de encontrar errores se denomina **depuración** del programa. La corrección del error es probablemente la etapa más fácil, siendo la detección y aislamiento del error las tareas más difíciles.

Existen diferentes situaciones en las cuales se suelen introducir errores en un programa. Dos de las más frecuentes son:

1. Violación (no cumplimiento) de las reglas gramaticales del lenguaje de alto nivel en el que se escribe el programa.
2. Los errores en el diseño del algoritmo en el que está basado el programa.

Cuando el compilador detecta un error, visualiza un *mensaje de error* indicando que se ha cometido un error y posible causa del error. Desgraciadamente los mensajes de error son difíciles de interpretar y a veces se llegan a conclusiones erróneas. También varían de un compilador a otro compilador. A medida que se gana en experiencia, el proceso de puesta a punto de un programa se mejora considerablemente. Nuestro objetivo en cada capítulo es describir los errores que ocurren más frecuentemente y sugerir posibles causas de error, junto con reglas de estilo de escritura de programas. Desde el punto de vista conceptual existen tres tipos de errores: *sintaxis*, *lógicos* y de *regresión*.

3.4.1. Errores de sintaxis

Los **errores de sintaxis** son aquellos que se producen cuando el programa viola la sintaxis, es decir, las reglas de gramática del lenguaje. Errores de sintaxis típicos son: escritura incorrecta de palabras reservadas, omisión de signos de puntuación (comillas, punto y coma...). Los errores de sintaxis son los más fáciles de fijar, ya que ellos son detectados y aislados por el compilador.

Estos errores se suelen detectar por el compilador durante el proceso de compilación. A medida que se produce el proceso de traducción del código fuente (por ejemplo, programa escrito en C) a lenguaje máquina de la computadora, el compilador verifica si el programa que se está traduciendo cumple las reglas de sintaxis del lenguaje. Si el programa viola alguna de estas reglas, el compilador genera un *mensaje de error (odiagnóstico)* que explica el problema (aparente). Algunos errores típicos (ya citados anteriormente):

- Punto y coma después de la cabecera `main()`.
- Omisión de punto y coma al final de una sentencia.
- Olvido de la secuencia `*/` para finalizar un comentario.
- Olvido de las dobles comillas al cerrar una cadena.
- Etc.

Si una sentencia tiene un error de sintaxis no se traducirá completamente y el programa no se ejecutará. Así, por ejemplo, si una línea de programa es

```
double radio
```

se producirá un error ya que falta el punto y coma (;) después de la letra última "o". Posteriormente se explicará el proceso de corrección por parte del programador.

3.4.2. Errores lógicos

Un segundo tipo de error importante es el **error lógico**, ya que tal error representa errores del programador en el diseño del algoritmo y posterior programa. Los errores lógicos son más difíciles de encontrar y aislar ya que no suelen ser detectados por el compilador.

Suponga, por ejemplo, que una línea de un programa contiene la sentencia

```
double peso = densidad * 5.25 * PI * pow(longitud,5)/4.0
```

pero resulta que el tercer asterisco (operador de multiplicación) es en realidad un signo $+$ (operador suma). El compilador no produce ningún mensaje de error de sintaxis ya que no se ha violado ninguna regla de sintaxis y, por tanto, *el compilador no detecta error* y el programa se compilará y ejecutará bien, aunque producirá resultados de valores incorrectos ya que la fórmula utilizada para calcular el peso contiene un error lógico.

Una vez que se ha determinado que un programa contiene un error lógico —si es que se encuentra en la primera ejecución y no pasa desapercibida al programador— encontrar el error es una de las tareas más difíciles de la programación. El **depurador** (*debugger*) un programa de software diseñado específicamente para la detección, verificación y corrección de errores, ayudará en las tareas de depuración.

Los errores lógicos ocurren cuando un programa es la implementación de un algoritmo defectuoso. Dado que los errores lógicos normalmente no producen errores en tiempo de ejecución y no visualizan mensajes de error; son más difíciles de detectar porque el programa parece ejecutarse sin contratiempos. El único signo de un error lógico puede ser la salida incorrecta de un programa. La sentencia

```
total-grados-centigrados = fahrenheit_a_centigrados * temperatura_cen;
```

es una sentencia perfectamente legal en C, pero la ecuación no responde a ningún cálculo válido para obtener el total de grados centígrados en una sala.

Se pueden detectar errores lógicos comprobando el programa en su totalidad, comprobando su salida con los resultados previstos. Se pueden prevenir errores lógicos con un estudio minucioso y detallado del algoritmo antes de que el programa se ejecute, pero resultará fácil cometer errores lógicos y es el conocimiento de C, de las técnicas algorítmicas y la experiencia lo que permitirá la detección de los errores lógicos.

3.4.3. Errores de regresión

Los **errores de regresión** son aquellos que se crean accidentalmente cuando se intenta corregir un error lógico. Siempre que se corrige un error se debe comprobar totalmente la exactitud (corrección) para asegurarse que se fija el error que se está tratando y no produce otro error. Los errores de regresión son comunes, pero son fáciles de leer y corregir. Una ley no escrita es que: «un error se ha producido, probablemente, por el último código modificado».

3.4.4. Mensajes de error

Los compiladores emiten mensajes de error o de advertencia durante las fases de compilación, de enlace o de ejecución de un programa.

Los mensajes de error producidos durante la compilación se suelen producir, normalmente, por errores de sintaxis y suele variar según los compiladores; pero, en general, se agrupan en tres grandes bloques:

- **Errores fatales.** Son raros. Algunos de ellos indican un error interno del compilador. Cuando ocurre un error fatal, la compilación se detiene inmediatamente, se debe tomar la acción apropiada y a continuación se vuelve a iniciar la compilación.
- **Errores de sintaxis.** Son los errores típicos de sintaxis, errores de línea de órdenes y errores de acceso a memoria o disco. El compilador terminará la fase actual de compilación y se detiene.
- **Advertencias** (*warning*). No impiden la compilación. Indican condiciones que son sospechosas, pero son legítimas como parte del lenguaje.

3.4.5. Errores en tiempo de ejecución

Existen dos tipos de *errores en tiempo de ejecución*: aquellos que son detectados por el sistema en tiempo de ejecución de C y aquellos que permiten la terminación del programa pero producen resultados incorrectos.

Un error en tiempo de ejecución puede ocurrir como resultado de que el programa obliga a la computadora a realizar una operación ilegal, tal como dividir un número por cero, **raíz** cuadrada de un número negativo o manipular datos no válidos o no definidos. Cuando ocurre este tipo de error, la computadora detendrá la ejecución de su programa y emitirá (visualizará) un mensaje de diagnóstico tal como:

```
Divide error, line number ***
```

Si se intenta manipular datos no válidos o indefinidos su salida puede contener resultados extraños. Por ejemplo, se puede producir un *desbordamiento aritmético* cuando un programa intenta almacenar un número que es mayor que el tamaño máximo que puede manipular su computadora.

El programa `depurar.c` se compila con éxito; pero no contiene ninguna sentencia que asigne un valor a la variable `x` que pueda sumarse a `y` y para producir un valor `z`, por lo tanto al ejecutarse la sentencia de asignación

```
z = x + y;
```

se produce un error en tiempo de ejecución, un error de lógica.

```
1: /* archivo depurar
2:   prueba de errores en tiempo de ejecución
3: */
4: #include <stdio.h>
5:
6: void main()
7: {
8:   /* Variables locales */
9:   float x, y, z;
10:
11:  y = 10.0
12:  z = x + y;           /* valor inesperado: error de ejecución */
13:  printf("El valor de z es = %f\n", z);
14: }
```

El programa anterior, sin embargo, podría terminar su ejecución, aunque produciría resultados incorrectos. Dado que no se asigna ningún valor a `x`, contendrá un valor impredecible y el resultado de la suma será también impredecible. Muchos compiladores inicializan las variables automáticamente a cero, haciendo en este caso más difícil de detectar la omisión, sobre todo cuando el programa se transfiere a otro compilador que no asigna ningún valor definido.

Otra fuente de errores en tiempo de ejecución se suele producir por errores en la entrada de datos producidos por la lectura del dato incorrecto en una variable de entrada.

3.5. PRUEBAS

Los **errores de ejecución** ocurren después que el programa se ha compilado con éxito y aún se está ejecutando. Existen ciertos errores que la computadora *sólo* puede detectar cuando se ejecuta el programa. La mayoría de los sistemas informáticos detectarán ciertos errores en tiempo de ejecución y presentarán un mensaje de error apropiado. Muchos errores en tiempo de ejecución tienen que ver con los cálculos numéricos. Por ejemplo, si la computadora intenta dividir un número por cero o leer un archivo no creado, se produce un error en tiempo de ejecución.

Es preciso tener presente que el compilador puede no emitir ningún mensaje de error durante la ejecución y eso no garantiza que el programa sea correcto. Recuerde *que el compilador sólo le indica si se escribió bien sintácticamente un programa en C. No indica si el programa hace lo que realmente desea que haga*. Los errores lógicos pueden aparecer —y de hecho aparecerán— por un mal diseño del algoritmo y posterior programa.

Para determinar si un programa contiene un error lógico, se debe ejecutar utilizando datos de muestra y comprobar la salida verificando su exactitud. Esta **prueba** (*testing*) se debe hacer varias veces utilizando diferentes entradas, preparadas —en el caso ideal—, por personas diferentes al programador, que puedan indicar suposiciones no evidentes en la elección de los datos de prueba. Si *cualquier* combinación de entradas produce salida incorrecta, entonces el programa contiene un error lógico.

Una vez que se ha determinado que un programa contiene un error lógico, la localización del error es una de las partes más difíciles de la programación. La ejecución se debe realizar paso a paso (seguir la *traza*) hasta el punto en que se observe que un valor calculado difiere del valor esperado. Para simplificar este *seguimiento* o *traza*, la mayoría de los compiladores de C proporcionan un depurador integrado¹ incorporado con el editor, y todos ellos en un mismo paquete de software, que permiten al programador ejecutar realmente un programa, línea a línea, observando los efectos de la ejecución de cada línea en los valores de los objetos del programa. Una vez que se ha localizado el error, se utilizará el editor de texto para corregir dicho error.

Es preciso hacer constar que casi nunca será posible comprobar un programa para todos los posibles conjuntos de datos de prueba. Existen casos en desarrollos profesionales en los que, aparentemente, los programas han estado siendo utilizados sin problemas durante años, hasta que se utilizó una combinación específica de entradas y ésta produjo una salida incorrecta debida a un error lógico. El conjunto de datos específicos que produjo el error nunca se había introducido.

A medida que los programas crecen en tamaño y complejidad, el problema de las pruebas se convierte en un problema de dificultad cada vez más creciente. No importa cuantas pruebas se hagan: «las pruebas nunca se terminan, sólo se detienen y no existen garantías de que se han encontrado y corregido todos los errores de un programa». Dijkstra ya predijo a principios de los setenta una máxima que siempre se ha de tener presente en la construcción de un programa: «*Las pruebas sólo muestran la presencia de errores, no su ausencia. No se puede probar que un programa es correcto (exacto) sólo se puede mostrar que es incorrecto*».

3.6. LOS ELEMENTOS DE UN PROGRAMA EN C

Un programa C consta de uno o más archivos. Un archivo es traducido en diferentes fases. La primera fase es el preprocesado, que realiza la inclusión de archivos y la sustitución de macros. El preprocesador se controla por directivas introducidas por líneas que contienen # como primer carácter. El resultado del preprocesado es una secuencia de *tokens*.

3.6.1. Tokens (elementos léxicos de los programas)

Existen cinco clases de *tokens*: identificadores, palabras reservadas, literales, operadores y otros separadores.

¹ Éste es el caso de Borland C++, Builder C++ de Borland/Imprise, Visual C++ de Microsoft o los compiladores bajo UNIX y Lyiiux. Suelen tener un menú *Debug* o bien una opción *Debug* en el menú *Run*.

3.6.2. Identificadores

Un *identificador* es una secuencia de caracteres, letras, dígitos y subrayados (_). El primer carácter debe ser una letra (algún compilador admite carácter de subrayado). Las letras mayúsculas y minúsculas son diferentes.

nombre-clase	Indice	Dia_Mes_Anyo
elemento_mayor	Cantidad-Total	Fecha-Compra-Casa
a	Habitacion120	1

En Borland C/C++ el identificador puede ser de cualquier longitud; sin embargo, el compilador ignora cualquier carácter fuera de los 32 primeros.

C es *sensible a las mayúsculas*. Por consiguiente, C reconoce como distintos los identificadores ALFA, alfa y ALFa. (Le recomendamos que utilice siempre el mismo estilo al escribir sus identificadores.) Un consejo que puede servir de posible regla puede ser:

1. Escribir identificadores de variables en letras minúsculas.
2. Constantes en mayúsculas.
3. Funciones con tipo de letra mixto: mayúscula/minúscula.

Reglas básicas de formación de identificadores

1. Secuencia de letras o dígitos; el primer carácter puede ser una letra o un subrayado (compiladores de Borland, entre otros).
2. Los identificadores son sensibles a las mayúsculas:

minun **es distinto** de MiNum
3. Los identificadores pueden tener cualquier longitud, pero sólo son significativos los 32 primeros (ése es el caso de Borland y Microsoft).
4. Los identificadores no pueden ser palabras reservadas, tales como `if`, `switch` o `else`.

3.6.3. Palabras reservadas

Una palabra reservada (*keyword* o *resewed word*), tal como `void` es una característica del lenguaje C asociada con algún significado especial. Una palabra reservada no se puede utilizar como nombre de identificador o función

```
void void()      /* error */
...
int char;      /* error */
...
}
```

Los siguientes identificadores están reservados para utilizarlos como *palabras reservadas*, y no se deben emplear para otros propósitos.

<code>asm</code>	<code>enum</code>	<code>signed</code>
<code>auto</code>	<code>extern</code>	<code>sizeof</code>
<code>break</code>	<code>float</code>	<code>static</code>
<code>case</code>	<code>for</code>	<code>struct</code>
<code>char</code>	<code>goto</code>	<code>switch</code>
<code>const</code>	<code>if</code>	<code>typedef</code>

<code>continue</code>	<code>int</code>	<code>union</code>
<code>default</code>	<code>long</code>	<code>unsigned</code>
<code>do</code>	<code>register</code>	<code>void</code>
<code>double</code>	<code>return</code>	<code>volatile</code>
<code>else</code>	<code>short</code>	<code>while</code>

3.6.4. Comentarios

Ya se ha expuesto antes que los comentarios en C tienen el formato:

```
/*...*/
```

Los comentarios se encierran entre `/*` y `*/` pueden extenderse a lo largo de varias líneas.

```
/* Título: Demo-uno por Mr. Martinez */
```

Otra forma, el comentario en dos líneas:

```
/* Cabecera del programa text-uno
   Autor: J.R. Mazinger */
```

3.6.5. Signos de puntuación y separadores

Todas las sentencias deben terminar con un punto y coma. Otros signos de puntuación son:

```
[ % ^ & * O - + = } -
[ ] \ ; ' : < > ? , . / "
```

Los separadores son espacios en blanco, tabulaciones, retornos de carro y avances de línea.

3.6.6. Archivos de cabecera

Un archivo de cabecera es un archivo especial que contiene declaraciones de elementos y funciones de la biblioteca. Para utilizar macros, constantes, tipos y funciones almacenadas en una biblioteca, un programa debe utilizar la directiva `#include` para insertar el archivo de cabecera correspondiente. Por ejemplo, si un programa utiliza la función `pow` que se almacena en la biblioteca matemática `math.h`, debe contener la directiva

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

para hacer que el contenido de la biblioteca matemática esté disponible a un programa. La mayoría de los programas contienen líneas como ésta al principio, que se incluyen en el momento de compilación.

```
#include <stdio.h>
/* o bien */
#include "stdio.h"
```

3.7. TIPOS DE DATOS EN C

C no soporta un gran número de tipos de datos predefinidos, pero tiene la capacidad para crear sus propios tipos de datos. Todos los tipos de datos simples o básicos de C son, esencialmente, números. Los tres tipos de datos básicos son:

- enteros;
- números de coma flotante (*reales*);
- caracteres.

La Tabla 3.1 recoge los principales tipos de datos básicos, sus tamaños en bytes y el rango de valores que puede almacenar.

Tabla 3.1. Tipos de datos simples de C.

Tipo	Ejemplo	Tamaño en bytes	Rango Mínimo..Máximo
char	'C'	1	0..255
short	-15	2	-128..127
int	1024	2	-32768..32767
unsigned int	42325	2	0..65535
long	262144	4	-2147483648..2147483637
float	10.5	4	$3.4 \cdot (10^{-38}) \dots 3.4 \cdot (10^38)$
double	0.00045	8	$1.7 \cdot (10^{-308}) \dots 1.7 \cdot (10^{308})$
long double	1e-8	8	<i>igual que</i> double

Los tipos de datos fundamentales en C son:

- **enteros:** (números completos y sus negativos), de tipo `int`.
- **variantes de enteros:** tipos `short`, `long` y `unsigned`.
- **reales:** números decimales, tipos `float`, `double` o `long double`.
- **caracteres:** letras, dígitos, símbolos y signos de puntuación, tipo `char`.

`char`, `int`, `float` y `double` son palabras reservadas, o más específicamente, *especificadores de tipos*. Cada tipo de dato tiene su propia *lista de atributos* que definen las características del tipo y pueden variar de una máquina a otra. Los tipos `char`, `int` y `double` tienen variaciones o *modificadores de tipos de datos*, tales como `short`, `long`, `signed` y `unsigned`, para permitir un uso más eficiente de los tipos de datos.

Existe el tipo adicional `enum` (constante de enumeración (Capítulo 9)).

3.7.1. Enteros (`int`)

Probablemente el tipo de dato más familiar es el entero, o tipo `int`. Los enteros son adecuados para aplicaciones que trabajen con datos numéricos. Los tipos enteros se almacenan internamente en 2 bytes (o 16bits) de memoria. La Tabla 3.2 resume los tres tipos enteros básicos, junto con el rango de valores y el tamaño en bytes usual, dependiendo de cada máquina.

Tabla 3.2. Tipos de datos enteros.

Tipo C	Rango de valores	Uso recomendado
<code>int</code>	-32.768 .. +32.767	Aritmética de enteros, bucles <code>for</code> , conteo.
<code>unsigned int</code>	0 .. 65.535	Conteo, bucles <code>for</code> , índices.
<code>short int</code>	-128 .. +127	Aritmética de enteros, bucles <code>for</code> , conteo.

Declaración de variables

La forma más simple de una declaración de variable en C es poner primero el tipo de dato y a continuación el nombre de la variable. Si se desea dar un valor inicial a la variable, éste se pone a continuación. El formato de la declaración es:

```
<tipo de dato> <nombre de variable> = <valor inicial>
```

Se pueden también declarar múltiples variables en la misma línea:

```
<tipo_de_dato> <nom-variz, <nom_var2> ... <nom-varn>
```

Así, por ejemplo,

```
int longitud; int valor = 99;
int valor1, valor2;
int num_parte = 1141, num_items = 45;
```

Los tres modificadores (`unsigned`, `short`, `int`) que funcionan con `int` (Tabla 3.3) varían el rango de los enteros.

En aplicaciones generales, las constantes enteras se escriben en *decimal* o *base 10*; por ejemplo, 100, 200 o 450. Para escribir una constante sin signo, se añade la letra `u` (o bien `U`). Por ejemplo, para escribir 40.000, escriba 40000U.

Si se utiliza C para desarrollar software para sistemas operativos o para hardware de computadora, será útil escribir constantes enteras en *octal* (base 8) o *hexadecimal* (base 16). Una constante octal es cualquier número que comienza con un 0 y contiene dígitos en el rango de 1 a 7. Por ejemplo, 0377 es un número octal. Una constante hexadecimal comienza con `0x` y va seguida de los dígitos 0 a 9 o las letras `A` a `F` (o bien `a` a `f`). Por ejemplo, `0xFF16` es una constante hexadecimal.

La Tabla 3.3 muestra ejemplos de constantes enteras representadas en sus notaciones (bases) decimal, hexadecimal y octal.

Cuando el rango de los tipos enteros básicos no es suficientemente grande para sus necesidades, se consideran tipos enteros largos. La Tabla 3.4 muestra los dos tipos de datos enteros largos. Ambos tipos requieren 4 bytes de memoria (32 bits) de almacenamiento. Un ejemplo de uso de enteros largos es:

```
long medida_milimetros;

unsigned long distancia_media;
```

Tabla 3.3. Constantes enteras en tres bases diferentes.

Base 10 Decimal	Base 16 Hexadecimal (Hex)	Base 8 Octal
8	0x08	010
10	0x0A	012
16	0x10	020
65536	0x10000	0200000
24	0x18	030
17	0x11	021

Si se desea forzar al compilador para tratar sus constantes como `long`, añada la letra `L` (o bien `l`) a su constante. Por ejemplo,

```
long numeros_grandes = 40000L;
```

Tabla 3.4. Tipos de datos enteros largos.

Tipo C	Rango de valores
long	-2147483648 .. 2147483647
unsigned long	0 .. +4294967295

3.7.2. Tipos de coma flotante (float/double)

Los tipos de datos de coma (*punto*)flotante representan números reales que contienen una coma (un punto) decimal, tal como 3.14159, o números muy grandes, tales como 1.85×10^{15} .

La declaración de las variables de coma flotante es igual que la de variables enteras. Así, un ejemplo es el siguiente:

```
float valor;           /* declara una variable real */
float valor1, valor2; /* declara varias variables de coma flotante */
float valor = 99.99;  /* asigna el valor 99.99 a la variable valor */
```

C soporta tres formatos de coma flotante (Tabla 3.5). El tipo float requiere 4 bytes de memoria, double requiere 8 bytes y long double requiere 10 bytes (Borland C).

Tabla 3.5. Tipos de datos en coma flotante (Borland C).

Tipo C	Rango de valores	Precisión
float	$3.4 \times 10^{38} \dots$	3.4×10^{38} 7 dígitos
double	1.7×10^{410} ...	1.7×10^H 15 dígitos
long double	3.4×10^I ...	1.1×10^{410} 19 dígitos

Ejemplos

```
float f;           /* definición de la variable f */
f = 1.65;         /* asignación a f */
printf("f: %f\n", f); /* visualización de f:5.65 */
double h;        /* definición de la variable de tipo double h */
h = 0.0;         /* asignación de 0.0 a h */
```

3.7.3. Caracteres (char)

Un *carácter* es cualquier elemento de un conjunto de caracteres predefinidos o alfabeto. La mayoría de las computadoras utilizan el conjunto de caracteres **ASCII**.

C procesa datos carácter (tales como texto) utilizando el tipo de dato char. En unión con la estructura array, que se verá posteriormente, se puede utilizar para almacenar *cadena de caracteres* (grupos de caracteres). Se puede definir una variable carácter escribiendo:

```
char dato-car;
char letra = 'A';
char respuesta = 'S';
```

Internamente, los caracteres se almacenan como números. La letra A, por ejemplo, se almacena internamente como el número 65, la letra B es 66, la letra c es 67, etc. El tipo char representa valores en el rango -128 a +127 y se asocian con el código **ASCII**.

Dado que el tipo `char` almacena valores en el rango de -128 a $+127$, C proporciona el tipo `unsigned char` para representar valores de 0 a 255 y así representar todos los caracteres ASCII.

Puesto que los caracteres se almacenan internamente como números, se pueden realizar operaciones aritméticas con datos tipo `char`. Por ejemplo, se puede convertir una letra minúscula *a* a una letra mayúscula *A*, restando 32 del código ASCII. Las sentencias para realizar la conversión:

```
char car-uno = 'a';

car-uno = car-uno - 32;
```

Esto convierte a (código ASCII 97) a A (código ASCII 65). De modo similar, añadiendo 32 convierte el carácter de letra mayúscula a minúscula:

```
car-uno = car-uno + 32;
```

Como los tipos `char` son subconjuntos de los tipos enteros, se puede asignar un tipo `char` a un entero. Por ejemplo,

```
int suma = 0;
char valor;
...
scanf("%c",&valor); /* función estándar de entrada */
suma = suma + valor; /* operador... */
```

Existen caracteres que tienen un propósito especial y no se pueden escribir utilizando el método normal. C proporciona **secuencias de escape**. Por ejemplo, el literal carácter de un apóstrofe se puede escribir como

```
'\''
```

y el carácter nueva línea

```
\n
```

La Tabla 3.7 enumera las diferentes secuencias de escape de C.

3.8. EL TIPO DE DATO LÓGICO

Los compiladores de C que siguen la norma ANSI no incorporan el tipo de dato **lógico** cuyos valores son «verdadero» (*true*) y «falso» (*false*). El lenguaje C simula este tipo de dato tan importante en la estructuras de control (`if`, `while`, `.`). Para ello utiliza el tipo de dato `int`. C interpreta todo valor distinto de 0 como «verdadero» y el valor 0 como «falso». De esta forma se pueden escribir expresiones lógicas de igual forma que en otros lenguajes de programación se utiliza *true* y *false*. Una expresión lógica que se evalúa a «0» se considera falsa; una expresión lógica que se evalúa a 1 (o valor entero distinto de 0) se considera verdadera.

Ejemplo

```
int bisiesto;
bisiesto = 1;

int encontrado, bandera;
```

Dadas estas declaraciones, las siguientes sentencias son todas válidas

```
if (encontrado) ... /* sentencia de selección */
```

```

indicador = 0;           /* indicador toma el valor falso */
indicador = suma > 10; /* indicador toma el valor 1(true) si suma es
                        mayor que 10, en caso contrario, 0 */

```

Valor distinto de cero representa *true* (verdadero)
0 representa *false* (falso)

En C, se puede definir un tipo que asocia valores enteros constantes con identificadores, es el tipo enumerado. Para representar los datos lógicos en C, el sistema usual es definir un tipo enumerado Boolean con dos identificadores *false* (valor 0) y *true* (valor 1) de la forma siguiente:

```
enum Boolean { FALSE, TRUE };
```

Esta declaración hace a Boolean un tipo definido por el usuario con literales o identificadores (valores constantes) TRUE y FALSE.

Ejercicio 3.1

Si desea simular el tipo lógico pero al estilo de tipo incorporado propio, se podría conseguir construyendo un archivo *boolean.h* (boolean) con constantes con nombre TRUE y FALSE, tal como

```

/* archivo: boolean.h */
#ifndef BOOLEAN-H
#define BOOLEAN-H
typedef int Boolean;
const int TRUE = 1;
const int FALSE = 0;
#endif /* BOOLEAN-H */

```

Entonces, basta con incluir el archivo "boolean.h" y utilizar Boolean como si fuera un tipo de dato incorporado con los literales TRUE y FALSE como literales lógicos o booleanos.

Si desea utilizar las letras minúsculas para definir boolean, true y false, se puede utilizar esta versión del archivo de cabecera boolean.h.

```

/* archivo: boolean.h */
#ifndef BOOLEAN-H
#define BOOLEAN-H
typedef int boolean;
const int true = 1;
const int false = 0;
#endif /* BOOLEAN_H */

```

3.8.1. Escritura de valores lógicos

La mayoría de las expresiones lógicas aparecen en estructuras de control que sirven para determinar la secuencia en que se ejecutan las sentencias C. Raramente se tiene la necesidad de leer valores lógicos como dato de entrada o de visualizar valores lógicos como resultados de programa. Si es necesario, se puede visualizar el valor de la variable lógica utilizando la función para salida printf(). Así, si encontrado es false, la sentencia

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

visualizará

```
El valor de encontrado es 0
```

3.9. CONSTANTES

En C existen cuatro tipos de constantes:

- *constantes literales,*
- *constantes definidas,*
- *constantes enumeradas,*
- *constantes declaradas.*

Las constantes literales son las más usuales; toman valores tales como 45.32564,222 o bien "Introduzca sus datos" que se escriben directamente en el texto del programa. Las constantes definidas son identificadores que se asocian con valores literales constantes y que toman determinados nombres. Las constantes declaradas son como variables: sus valores se almacenan en memoria, pero no se pueden modificar. Las constantes enumeradas permiten asociar un identificador, tal como Color, con una secuencia de otros nombres, tales como Azul, Verde, Rojo y Amarillo.

3.9.1. Constantes literales

Las constantes literales o *constantes*, en general, se clasifican también en cuatro grupos, cada uno de los cuales puede ser de cualquiera de los tipos:

- constantes enteras,
- constantes caracteres,
- constantes de coma flotante,
- constantes de cadena.

Constantes enteras

La escritura de constantes enteras requiere seguir unas determinadas reglas:

- No utilizar nunca comas ni otros signos de puntuación en números enteros o completos.
123456 *en lugar de* 123.456
- Para forzar un valor al tipo long, terminar con una letra L o l. Por ejemplo,
1024 *es un tipo entero* 1024L *es un tipo largo (long)*
- Para forzar un valor al tipo unsigned, terminarlo con una letra mayúscula u. Por ejemplo, 4352U.
- Para representar un entero en octal (base 8), éste debe de estar precedido de 0.

<i>Formato decimal</i>	123	
<i>Formato octal</i>	0777	(están precedidas de la cifra 0)

- Para representar un entero en hexadecimal (base 16), este debe de estar precedido de 0x.

<i>Formato hexadecimal</i>	0XFF3A	(están precedidas de "0x" o bien "0X")
----------------------------	--------	--

Se pueden combinar sufijos L(1), que significa *long* (largo), o bien u(u), que significa *unsigned* (sin signo).

3456UL

Constantes reales

Una constante flotante representa un número real; siempre tienen signo y representan aproximaciones en lugar de valores exactos.

82.347 .63 83. 47e-4 1.25E7 61.e+4

La notación científica se representa con un exponente positivo o negativo.

2.5E4 *equivale a* 25000
 5.435E-3 *equivale a* 0.005435

Existen tres tipos de constantes:

float 4 bytes
 double 8 bytes
 long double 10 bytes

Constantes carácter

Una constante carácter (char) es un carácter del código ASCII encerrado entre apóstrofes.

'A' 'b' 'c'

Además de los caracteres ASCII estándar, una constante carácter soporta caracteres especiales que no se pueden representar utilizando su teclado, como, por ejemplo, los códigos ASCII altos y las secuencias de escape. (El Apéndice B recoge un listado de todos los caracteres ASCII.)

Así, por ejemplo, el carácter sigma (Σ) -código ASCII 228, hex E4— se representa mediante el prefijo \x y el número hexadecimal del código ASCII. Por ejemplo,

char sigma = '\xE4';

Este método se utiliza para almacenar o imprimir cualquier carácter de la tabla ASCII por su número hexadecimal. En el ejemplo anterior, la variable sigma no contiene cuatro caracteres sino únicamente el símbolo sigma.

Tabla 3.6. Caracteres secuencias (códigos) de escape.

Código de escape	Significado	Códigos ASCII	
		Dec	Hex
'\n'	nueva línea	13 10	0D 0A
'\r'	retorno de carro	13	0D
'\t'	tabulación	9	09
'\v'	tabulación vertical	11	0B
'\a'	alerta (pitido sonoro)	7	07
'\b'	retroceso de espacio	8	08
'\f'	avance de página	12	0C
'\'	barra inclinada inversa	92	5 c
'\''	comilla simple	39	27
'\"'	doble comilla	34	22
'\?'	signo de interrogación	63	3F
'\000'	número octal	<i>Todos</i>	<i>Todos</i>
'\xhh'	número hexadecimal	<i>Todos</i>	<i>Todos</i>

Un carácter que se lee utilizando una barra oblicua (\) se llama *secuencia* o *código de escape*. La Tabla 3.6 muestra diferentes secuencias de escape y su significado.

```

/* Programa: Pruebas códigos de escape */
#include <stdio.h>

int main0

    char alarma = '\a';          /* alarma */
    char bs = '\b';             /* retroceso de espacio */
    printf("%c %c", alarma, bs);
    return 0;
}

```

Aritmética con caracteres C

Dada la correspondencia entre un carácter y su código ASCII, es posible realizar operaciones aritméticas sobre datos de caracteres. Observe el siguiente segmento de código:

```

char c;
c = 'T' + 5;          /* suma 5 al carácter ASCII */

```

Realmente lo que sucede es almacenar Y en c. El valor ASCII de la letra T es 84, y al sumarle 5 produce 89, que es el código de la letra Y. A la inversa, se pueden almacenar constantes de carácter en variables enteras. Así,

```
int j = 'p'
```

No pone una letra p en j, sino que asigna el valor 80—código ASCII de p— a la variable j. Observar este pequeño segmento de código:

```
int m;
m = m + 'a' - 'A';
```

Está convirtiendo una letra mayúscula en su correspondiente minúscula. Para lo cual suma el desplazamiento de las letras mayúsculas a las minúsculas ('a' - 'A').

Constantes cadena

Una *constante cadena* (también llamada *literal cadena* o simplemente *cadena*) es una secuencia de caracteres encerrados entre dobles comillas. Algunos ejemplos de constantes de cadena son:

```

"123"
"12 de octubre 1492"
"esto es una cadena"

```

Se puede escribir una cadena en varias líneas, terminando cada línea con “\”

```

"esto es una cadena\
que tiene dos líneas"

```

Se puede concatenar cadenas, escribiendo

```

"ABC" "DEF" "GHI"
"JKL"

```

que equivale a

```
"ABCDEFGHIJKL"
```

En memoria, las cadenas se representan por una serie de caracteres ASCII más un 0 o nulo. El carácter nulo marca el final de la cadena y se inserta automáticamente por el compilador C al final de las constantes de cadenas. Para representar valores nulos, C define el símbolo `NULL` como una constante en diversos archivos de cabecera (normalmente `STDEF.H`, `STDIO.H`, `STDLIB.H` y `STRING.H`). Para utilizar `NULL` en un programa, incluya uno o más de estos archivos en lugar de definir `NULL` con una línea tal como

```
#define NULL 0
```

Recuerde que una constante de caracteres se encierra entre comillas simples (apóstrofe), y las constantes de cadena encierran caracteres entre dobles comillas. Por ejemplo,

```
'z'  "z"
```

El primer `'z'` es una constante carácter simple con una longitud de 1, y el segundo `"z"` es una constante de cadena de caracteres también con la longitud 1. La diferencia es que la constante de cadena incluye un cero (nulo) al final de la cadena, ya que C necesita conocer dónde termina la cadena, y la constante carácter no incluye el nulo ya que se almacena como un entero. Por consiguiente, *no puede mezclar constantes caracteres y cadenas de caracteres en su programa.*

3.9.2. Constantes definidas (simbólicas)

Las constantes pueden recibir nombres simbólicos mediante la directiva `#define`.

```
#define NUEVALINEA \n
#define PI 3.141592
#define VALOR 54
```

C sustituye los valores `\n`, `3.141592` y `54` cuando se encuentra las constantes simbólicas `NUEVALINEA`, `PI` y `VALOR`. Las líneas anteriores no son sentencias y, por ello, no terminan en punto y coma.

```
printf ("El valor es %dNUEVALINEA",VALOR);
```

Escribe en pantalla la constante `VALOR`. Realmente, el compilador lo que hace es sustituir en el programa todas las ocurrencias de `VALOR` por `54`, antes de analizar sintácticamente el programa fuente.

3.9.3. Constantes enumeradas

Las constantes enumeradas permiten crear listas de elementos afines. Un ejemplo típico es una constante enumerada de lista de colores, que se puede declarar como:

```
enum Colores {Rojo, Naranja, Amarillo, Verde, Azul, Violeta};
```

Cuando se procesa esta sentencia, el compilador asigna un valor que comienza en 0 a cada elemento enumerado; así, `Roj o` equivale a 0, `Naranja` es 1, etc. El compilador *enumera* los identificadores por usted. Después de declarar un tipo de dato enumerado, se pueden crear variables de ese tipo, como con cualquier otro tipo de datos. Así, por ejemplo, se puede definir una variable de tipo `enum colores`.

```
enum Colores Colorfavorito = Verde;
```

Otro ejemplo puede ser:

```
enum Boolean { False, True };
```

que asignará al elemento `False` el valor 0 y a `True` el valor 1.

Para crear una variable de tipo lógico declarar:

```
enum Boolean Interruptor = True;
```

Es posible asignar valores distintos de los que les corresponde en su secuencia natural

```
enum LucesTrafico {Verde, Amarillo = 10, Rojo};
```

Al procesar esta sentencia, el compilador asigna el valor 0 al identificador verde, 10 al identificador Amarillo y 11a Rojo.

3.9.4. Constantes declaradas `const` y `volatile`

El cualificador `const` permite dar nombres simbólicos a constantes a modo de otros lenguajes, como Pascal. El formato general para crear una constante es:

```
const tipo nombre = valor;
```

Si se omite **tipo**, C utiliza `int` (entero por defecto)

```
const int Meses=12;      /* Meses es constante simbólica de valor 12 */
```

```
const char CHARACTER='@';
const int OCTAL=0233;
const char CADENA []="Curso de C";
```

C soporta el calificador de tipo variable `const`. Especifica que el valor de una variable no se puede modificar durante el programa. Cualquier intento de modificar el valor de la variable definida con `const` producirá un mensaje de error.

```
const int semana = 7;
const char CADENA []= "Borland C 3.0/3.1 Guía de referencia";
```

La palabra reservada `volatile` actúa como `const`, pero su valor puede ser modificado no sólo por el propio programa, sino también por el *hardware* o por el *software* del sistema. Las variables volátiles, sin embargo, no se pueden guardar en registros, como es el caso de las variables normales.

Diferencias entre `const` y `#define`

Las definiciones `const` especifican tipos de datos, terminan con puntos y coma y se inicializan como las variables. La directiva `#define` no especifica tipos de datos, no utilizan el operador de asignación (=) y no termina con punto y coma.

Ventajas de `const` sobre `#define`

En C casi siempre es recomendable el uso de `const` en lugar de `#define`. Además de las ventajas ya enunciadas se pueden considerar otras:

- El compilador, normalmente, genera código más eficiente con constantes `const`.
- Como las definiciones especifican tipos de datos, el compilador puede comprobar inmediatamente si las constantes literales en las definiciones de `const` están en forma correcta. Con `#define` el compilador no puede realizar pruebas similares hasta que una sentencia utiliza el identificador constante, por lo que se hace más difícil la detección de errores.



Desventaja de `const` sobre `#define`

Los valores de los símbolos de `const` ocupan espacio de datos en tiempo de ejecución, mientras que `#define` sólo existe en el texto del programa y su valor se inserta directamente en el código compilado. Los valores `const` no se pueden utilizar donde el compilador espera un valor constante, por ejemplo en la definición de un array. Por esta razón, algunos programadores de C siguen utilizando `#define` en lugar de `const`.

Sintaxis de `const`

```
const tipoDato nombreConstante = valorConstante;

const unsigned DiasDeSemana = 7;
const HorasDelDia = 24;
```

3.10. VARIABLES

En C una *variable* es una posición con nombre en memoria donde se almacena un valor de un cierto tipo de dato. Las variables pueden almacenar todo tipo de datos: cadenas, números y estructuras. Una *constante*, por el contrario, es una variable cuyo valor no puede ser modificado.

Una variable típicamente tiene un nombre (un identificador) que describe su propósito. Toda variable utilizada en un programa debe ser declarada previamente. La definición en C debe situarse al principio del bloque, antes de toda sentencia ejecutable. Una definición reserva un espacio de almacenamiento en memoria. El procedimiento para definir (*crear*) una variable es escribir el tipo de dato, el identificador o nombre de la variable y, en ocasiones, el valor inicial que tomará. Por ejemplo,

```
char Respuesta;
```

significa que se reserva espacio en memoria para `Respuesta`, en este caso, un carácter ocupa un solo byte.

El nombre de una variable ha de ser un identificador válido. Es frecuente, en la actualidad, utilizar subrayados en los nombres, bien al principio o en su interior, con objeto de obtener mayor legibilidad y una correspondencia mayor con el elemento del mundo real que representa.

```
salario      dias-de-semana      edad-alumno      _fax
```

3.10.1. Declaración

Una *declaración* de una variable es una sentencia que proporciona información de la variable al compilador C. Su sintaxis es:

tipo variable

tipo es el nombre de un tipo de dato conocido por el C
variable es un identificador (nombre) válido en C.

Ejemplo

```
long dNumero;
double HorasAcumuladas;
float HorasPorSemana;
float NotaMedia;
short DiaSemana;
```

Es preciso *declarar* las variables antes de utilizarlas. Se puede declarar una variable al principio de un archivo o bloque de código al principio de una función.

```
#include <stdio.h>          /* variable al principio del archivo */

int MiNumero;

int main()
{
    printf("¿Cuál es su número favorito?");
    scanf("%d",&MiNumero);
    return 0;
}

/*Variable al principio de una función .
Al principio de la función main()*/
...

int main()
{
    int i;
    int j;
}

/*Variable al principio de un bloque.
Al principio de un bloque for*/
...

int main()
{
    int i;

    for (i=0; i<9; i++)
    {
        double suma;
        ...
    }
    ...
}
```

En C las declaraciones se han de situar siempre al principio del bloque. Su ámbito es el bloque en el que están declaradas.

Ejemplo

```
/* Distancia a la luna en kilometros */

#include <stdio.h>

int main()
```

```

{
  const int luna=238857;          /* Distancia en millas */
  float luna-kilo;
  printf("Distancia a la Luna %d millas\n",luna);
  luna-kilo = luna*1.609;        /* una milla = 1.609 kilómetros */
  printf("En kilómetros es %fKm.\n",luna_kilo);
  return 0;
}

```

Ejemplo 3.3

Este ejemplo muestra cómo una variable puede ser declarada al inicio de cualquier bloque de un programa C.

```

#include <stdio.h>
/* Diferentes declaraciones */
int main()
{
  int x, y1; /* declarar a las variables x e y1 en la función main() */
  x = 75;
  y1 = 89;
  if (x > 10)

      int y2 = 50; /* declarar e inicializa a la variable y2 en el
                  bloque if */

      y1 = y1+y2;
}
printf("x = %d, y1 = %d\n",x,y1);
return 0;
}

```

3.10.2. Inicialización de variables

En algunos programas anteriores, se ha proporcionado un valor denominado **valor inicial**, a una variable cuando se declara. El formato general de una declaración de inicialización es:

tipo nombre-variable = expresión

expresión es cualquier expresión válida cuyo valor es del **mismo** tipo que *tipo*.

Nota. Esta sentencia declara y proporciona un valor inicial a una variable.

Las variables se pueden inicializar a la vez que se declaran, o bien, inicializarse después de la declaración. El primer método es probablemente el mejor en la mayoría de los casos, ya que combina la definición de la variable con la asignación de su valor inicial.

```

char respuesta = 'S';
int contador = 1;
float peso = 156.45;
int anyo = 1993;

```

Estas acciones crean variables *respuesta*, *contador*, *peso* y *anyo*, que almacenan en memoria los valores respectivos situados a su derecha.

El segundo método consiste en utilizar sentencias de asignación diferentes después de definir la variable, como en el siguiente caso:

```
char barra;
barra = '/';
```

3.10.3. Declaración o definición

La diferencia entre *declaración* y *definición* es sutil. Una declaración introduce un nombre de una variable y asocia un tipo con la variable. Una definición es una declaración que asigna simultáneamente memoria a la variable.

```
double x;          /* declara el nombre de la variable x de tipo double */
char c_var;       /* declara c_var de tipo char */
int i;            /* definido pero no inicializado */
int i = 0;        /* definido e inicializado a cero.*/
```

3.11. DURACIÓN DE UNA VARIABLE

Dependiendo del lugar donde se definan las variables de C, éstas se pueden utilizar en la totalidad del programa, dentro de una función o pueden existir sólo temporalmente dentro de un bloque de una función. La zona de un programa en la que una variable está activa se denomina, normalmente, *ámbito* o *alcance* («scope»).

El *ámbito* (*alcance*) de una variable se extiende hasta los límites de la definición de su bloque. Los tipos básicos de variables en C son:

- variables *locales*;
- variables *globales*;
- variables *dinámicas*.

3.11.1. Variables locales

Las variables locales son aquéllas definidas en el interior de una función y son visibles sólo en esa función específica. Las reglas por las que se rigen las variables locales son:

1. En el interior de una función, una variable local no puede ser modificada por ninguna sentencia externa a la función.
2. Los nombres de las variables locales no han de ser Únicos. Dos, tres o más funciones pueden definir variables de nombre *Interruptor*: Cada variable es distinta y pertenece a la función en que está declarada.
3. Las variables locales de las funciones no existen en memoria hasta que se ejecuta la función. Esta propiedad permite ahorrar memoria, ya que permite que varias funciones compartan la misma memoria para sus variables locales (pero no a la vez).

Por la razón dada en el punto 3, las variables locales se llaman también *automáticas* o *auto*, ya que dichas variables se crean automáticamente en la entrada a la función y se liberan también automáticamente cuando se termina la ejecución de la función.

```
#include <stdio.h>

int main()
```

```

{
    int a, b, c, suma, numero; /*variables locales */

    printf("Cuantos números a sumar:");
    scanf("%d",&numero);

    suma = a + b + c;
    ...
    return 0;
}

```

3.11.2. Variables globales

Las *variables globales* son variables que se declaran fuera de la función y por defecto (omisión) son visibles a cualquier función, incluyendo main() .

```

#include <stdio.h>

int a, b, c; /* declaración de variables globales */

int main()
{
    int valor; /* declaración de variable local */
    printf("Tres valores: ");
    scanf("%d %d %d",&a,&b,&c); /* a,b,c variables globales */
    valor = a+b+c;
}

```

Todas las variables locales desaparecen cuando termina **su** bloque. Una variable global es visible desde el punto en que se define hasta el final del programa (archivo fuente).

La memoria asignada a una variable global permanece asignada a través de la ejecución del programa, tomando espacio válido según se utilice. Por esta razón, se debe evitar utilizar muchas variables globales dentro de un programa. Otro problema que surge con variables globales es que una función puede asignar un valor específico a una variable global. Posteriormente, en otra función, y por olvido, se pueden hacer cambios en la misma variable. Estos cambios dificultarán la localización de errores.

3.11.3. Variables dinámicas

Las *variables dinámicas* tienen características que en algunos casos son similares tanto a variables locales como a globales. Al igual que una variable local, una variable dinámica se crea y libera durante la ejecución del programa. La *diferencia entre una variable local y una variable dinámica* es que la variable dinámica se crea tras su petición (en vez de automáticamente, como las variables locales), es decir, a su voluntad, y se libera cuando ya no se necesita. Al igual que una variable global, se pueden crear variables dinámicas que son accesibles desde múltiples funciones. Las variables dinámicas se examinan en detalle en el capítulo de *punteros* (Capítulo 10).

En el segmento de código C siguiente, *Q* es una variable *global* por estar definida fuera de las funciones y es accesible desde todas las sentencias. Sin embargo, las definiciones dentro de *main*, como *A*, son *locales* a *main*. Por consiguiente, sólo las sentencias interiores a *main* pueden utilizar *A*.

```
#include <stdio.h>
int Q;           Alcance o ámbito global
                 Q, variable global

int main()
{
    int A;       Local u main
                 A, variable local

    A = 124;
    Q = 1;

    {
        int B;   Primer subnivel en main
                 B, variable local

        B = 124;
        A = 457;
        Q = 2;

        {
            int C; Subnivel más interno de main
                   C, variable local

            C = 124;
            B = 457;
            A = 788;
            Q = 3;
        }
    }
}
```

3.12. ENTRADAS Y SALIDAS

Los programas interactúan con el exterior, a través de datos de entrada o datos de salida. La biblioteca C proporciona facilidades para entrada y salida, para lo que todo programa deberá tener el archivo de cabecera *stdio.h*. En C la entrada y salida se lee y escribe de los dispositivos estándar de entrada y salida, se denominan *stdin* y *stdout* respectivamente. La salida, normalmente, es a pantalla del ordenador, la entrada se capta del teclado.

En el archivo *stdio.h* están definidas macros, constantes, variables y funciones que permiten intercambiar datos con el exterior. A continuación se muestran las más habituales y fáciles de utilizar.

3.12.1. Salida

La salida de datos de un programa se puede dirigir a diversos dispositivos, pantalla, impresora, archivos. La salida que se trata a continuación va a ser a pantalla, además será formateada. La función *printf()* visualiza en la pantalla datos del programa, transforma los datos, que están en representación binaria, a ASCII según los códigos transmitidos. Así, por ejemplo,

```
suma = 0;
suma = suma+10;
printf("%s %d", "Suma = ", suma);
```

visualiza

```
Suma = 10
```

El número de argumentos de `printf()` es indefinido, por lo que se pueden transmitir cuantos datos se desee. **Así**, suponiendo que

```
i = 5      j = 12      c = 'A'      n = 40.791512
```

la sentencia

```
printf("%d %d %c %f", i, j, c, n);
```

visualizará en pantalla

```
5 12 A 40.791512
```

La forma general que tiene la función `printf()`

```
printf (cadena-de-control, dato1, dato2, ...)
```

cadena-de-control contiene los tipos de los datos y forma de mostrarlos.
dato1, dato2 ... variables, constantes, datos de salida.

`printf()` convierte, da forma de salida a los datos y los escribe en pantalla. La cadena de control contiene códigos de formato que se asocian uno a uno con los datos. Cada código comienza con el carácter %, a continuación puede especificarse el ancho mínimo del dato y termina con el carácter de conversión. **Así**, suponiendo que

```
i = 11      j = 12      c = 'A'      n = 40.791512  
printf("%x %3d %c %.3f", i, j, c, n);
```

visualizará en pantalla

```
B 12 A 40.792
```

El primer dato es 11 en hexadecimal (`%x`), el segundo es el número entero 12 en un ancho de 3, le sigue el carácter A y, por último, el número real `n` redondeado a 3 cifras decimales (`%.3f`). Un signo menos a continuación de % indica que el dato se ajuste a la izquierda en vez del ajuste a la derecha por defecto.

```
printf("%15s", "HOLA LUCAS");  
printf("%-15s", "HOLA LUCAS");
```

visualizará en pantalla

```
      HOLA LUCAS  
HOLA LUCAS
```

Los códigos de formato más utilizados y su significado:

<code>%d</code>	El dato se convierte a entero decimal.
<code>%o</code>	El dato entero se convierte a octal.
<code>%x</code>	El dato entero se convierte a hexadecimal.
<code>%u</code>	El dato entero se convierte a entero sin signo.
<code>%c</code>	El dato se considera de tipo carácter.
<code>%e</code>	El dato se considera de tipo <code>float</code> . Se convierte a notación científica, de la forma <code>{-}n.mmmmmE{+ -}dd</code> .
<code>%f</code>	El dato se considera de tipo <code>float</code> . Se convierte a notación decimal, con parte entera y los dígitos de precisión.
<code>%g</code>	El dato se considera de tipo <code>float</code> . Se convierte según el código <code>%e</code> o <code>%f</code> dependiendo de cual sea la representación más corta.
<code>%s</code>	El dato ha de ser una cadena de caracteres.
<code>%lf</code>	El dato se considera de tipo <code>double</code> .

C utiliza *secuencias de escape* para visualizar caracteres que no están representados por símbolos tradicionales, tales como `\a`, `\b`, etc. Las secuencias de escape clásicas se muestran en la Tabla 3.7.

Las secuencias de escape proporcionan flexibilidad en las aplicaciones mediante efectos especiales.

```
printf("\n Error · Pulsar una tecla para continuar \n");

printf("\n");          /* salta a una nueva línea */

printf("Yo estoy preocupado\n no por el \n sino por ti.\n");
```

la última sentencia visualiza

```
Yo estoy preocupado
no por el
sino por ti.
```

debido a que la secuencia de escape `\n` significa *nueva línea* o *salto de línea*. Otros ejemplos:

```
printf("\n Tabla de números \n");    /* uso de \n para nueva línea */

printf("\nNum1\t Num2\t Num3\n");    /* uso de \t para tabulaciones */

printf("%c", '\a');                  /* uso de \a para alarma sonora */
```

en los que se utilizan los caracteres de secuencias de escape de *nueva línea* (`\n`), *tabulación* (`\t`) y *alarma* (`\a`).

Tabla 3.7. Caracteres secuencias de escape.

Secuencia de escape	Significado
<code>\a</code>	Alarma
<code>\b</code>	Retroceso de espacio
<code>\f</code>	Avance de página
<code>\n</code>	Retorno de carro y avance de línea
<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulación
<code>\v</code>	Tabulación vertical
<code>\\</code>	Barra inclinada
<code>\?</code>	Signo de interrogación
<code>\"</code>	Dobles comillas
<code>\000</code>	Número octal
<code>\xhh</code>	Número hexadecimal
<code>\0</code>	Cero, nulo (ASCII 0)

Ejemplo 3.4

El listado *SECESC*. C utiliza secuencias de escape, tales como emitir sonidos (pitidos) en el terminal dos veces y a continuación presentar dos retrocesos de espacios en blanco.

```
/* Programa:SECESC.C
   Propósito: Mostrar funcionamiento de secuencias de escape
*/
```

```

#include <stdio.h >

int main()
{
    char sonidos='\a'; /* secuencia de escape alarma en sonidos */

    char bs='\b';      /* almacena secuencia escape retroceso en bs */

    printf("%c%c",sonidos,sonidos);/* emite el sonido dos veces */
    printf("ZZ");          /* imprime dos caracteres */

    printf("%c%c",bs,bs); /* mueve el cursor al primer carácter 'Z' */

    return 0;
}

```

3.12.2. Entrada

La entrada de datos a un programa puede tener diversas fuentes, teclado, archivos en disco. La entrada que consideramos ahora es a través del teclado, asociado al archivo estándar de entrada **stdin**. La función más utilizada, por su versatilidad, para entrada formateada es `scanf()`.

El archivo de cabecera `stdio.h` de la biblioteca C proporciona la definición (el prototipo) de `scanf()`, así como de otras funciones de entrada o de salida. La forma general que tiene la función `scanf()`

```
scanf(cadena_de_control, var1, var2, var3, ...)
```

cadena-de-control contiene los tipos de los datos y si se desea su anchura.
var1, var2 ... variables del tipo de los códigos de control.

Los códigos de formato más comunes son los ya indicados en la salida. Se pueden añadir, como sufixo del código, ciertos modificadores como **I** o **L**. El significado es «largo», aplicado a `float` (`%lf`) indica tipo `double`, aplicado a `int` (`%ld`) indica entero largo.

```
int n; double x;
scanf("%d %lf",&n,&x);
```

La entrada tiene que ser de la forma

```
134 -1.4E-4
```

En este caso la función `scanf()` devuelve `n=134` `x=-1.4E-4` (en doble precisión). Los argumentos `var1, var2 ...` de la función `scanf()` se pasan por dirección o referencia pues van a ser modificados por la función para devolver los datos. Por ello necesitan el operador de dirección, el prefijo `&`. Un error frecuente se produce al escribir, por ejemplo,

```
double x;
scanf("%lf",x);
```

en vez de

```
scanf("%lf",&x);
```

Las variables que se pasan a `scanf()` se transmiten por referencia para poder ser modificadas y transmitir los datos de entrada, para ello se hacen preceder de `&`.

Un ejemplo típico es el siguiente:

```
printf("introduzca v1 y v2:");
scanf("%d %f",&v1,&v2); /*lectura valores v1 y v2 */

printf("Precio de venta al público");
scanf("%f",&Precio_venta);

printf("Base y altura: ");
scanf("%f %f",&b,&h);
```

La función `scanf()` termina cuando ha captado tantos datos como códigos de control se han especificado, o cuando un dato no coincide con el código de control especificado.

Ejemplo 3.5

¿Cuál es la salida del siguiente programa, si se introducen por teclado las letras **LJ**?

```
#include <stdio.h>
int main()
{
    char primero, ultimo;
    printf("Introduzca su primera y Última inicial:");
    scanf("%c %c",&primero,&ultimo);
    printf("Hola, %c . %c .\n",primero,ultimo);
    return 0
}
```

3.12.3. Salida de cadenas de caracteres

Con la función `printf()` se puede dar salida a cualquier dato, asociándolo el código que le corresponde. En particular, para dar salida a una cadena de caracteres se utiliza el código `%s`. Así,

```
char arbol[] = "Acebo";
printf("%s\n", arbol);
```

Para salida de cadenas, la biblioteca C proporciona la función específica `puts()`. Tiene un solo argumento, que es una cadena de caracteres. Escribe la cadena en la salida estándar (pantalla) y añade el fin de línea. Así,

```
puts(arbol);
```

muestra en pantalla lo mismo que `printf("%s\n", arbol);`

Ejemplo 3.6

¿Cuál es la salida del siguiente programa?

```
#include <stdio.h>
#define T "Tambor de hojalata."
int main()
```

```

{
    char st[21] = "Todo puede hacerse."
    puts(T);
    puts("Permiso para salir en la toto.");
    puts(st);
    puts(&st[8]);
    return 0;
}

```

3.12.4. Entrada de cadenas de caracteres

La entrada de una cadena de caracteres se hace con la función más general `scanf()` y el código `%s`. Así, por ejemplo,

```

char nombre[51];
printf("Nombre del atleta: ");
scanf("%s", nombre);
printf("Nombre introducido: %s", nombre);

```

La entrada del nombre podría ser

```
Junipero Serra
```

La salida

```
Nombre introducido: Junipero
```

`scanf()` con el código `%s` capta palabras, el criterio de terminación es el encontrarse un blanco, o bien fin de línea.

También comentar que `nombre` no tiene que ir precedido del operador de dirección `&`. En C el identificador de un array, `nombre` lo es, tiene la dirección del array, por lo que en `scanf()` se transmite la dirección del array `nombre`.

La biblioteca de C tiene una función específica para captar una cadena de caracteres, la función `gets()`. Capta del dispositivo estándar de entrada una cadena de caracteres, termina la captación con un retorno de carro. El siguiente ejemplo muestra cómo captar una línea de como máximo 80 caracteres.

```

char linea[81];
puts("Nombre y dirección");
gets(linea);

```

La función `gets()` tiene un solo argumento, una variable tipo cadena. Capta la cadena de entrada y la devuelve en la variable pasada como argumento.

```
gets(variable_cadena);
```

Tanto con `scanf()` como con `gets()`, el programa inserta al final de la cadena el carácter que indica fin de cadena, el carácter nulo, `\0`. Siempre hay que definir las cadenas con un espacio más del previsto como máxima longitud para el carácter fin de cadena.

3.13. RESUMEN

Este capítulo le ha introducido a los componentes básicos de un programa C. En posteriores capítulos se analizarán en profundidad cada uno de los componentes. En este capítulo ha aprendido lo siguiente:

- La estructura general de un programa C.
- La mayoría de los programas C tienen una o más directivas `#include` al principio del programa fuente. Estas directivas `#include` proporcionan información adicional para crear su programa; éste, normalmente, utiliza `#include` para acceder a funciones definidas en archivos de biblioteca.
- Cada programa debe incluir una función llamada `main()`, aunque la mayoría de los programas tendrán muchas funciones además de `main()`.
- En C, se utiliza la función `scanf()` para obtener entrada del teclado. Para cadenas puede utilizarse, además, la función `gets()`.
- Para visualizar salida a la pantalla, se utiliza la función `printf()`. Para cadenas puede utilizarse la función `puts()`.
- Los nombres de los identificadores deben comenzar con un carácter alfabético, seguido por

un número de caracteres alfabéticos o numéricos, o bien, el carácter subrayado (`_`). El compilador normalmente ignora los caracteres posterior al 32.

- Los tipos de datos básicos de C son: enteros (`int`), entero largo (`long`), carácter (`char`), coma flotante (`float`, `double`). Cada uno de los tipos enteros tiene un calificador `unsigned` para almacenar valores positivos.
- Los tipos de datos carácter utilizan 1 byte de memoria; el tipo entero utiliza 2 bytes; el tipo entero largo utiliza 4 bytes y los tipos de coma flotante 4, 8 o 10 bytes de almacenamiento.
- Se utilizan conversiones forzosas de tipo o *moldes/ahormados* de tipos (`cast`) para convertir un tipo a otro. El compilador realiza automáticamente muchas conversiones de tipos. Por ejemplo, si se asigna un entero a una variable `float`, el compilador convierte automáticamente el valor entero a un tipo `float`.

Se puede seleccionar explícitamente una conversión de tipos precediendo la variable o expresión con (*tipo*), en donde *tipo* es un tipo de dato válido.

3.14. EJERCICIOS

3.1. ¿Cuál es la salida del siguiente programa'?

```
#include <stdio.h>
int main()
{
    char pax[] = "Juan Sin Miedo";
    printf("%s %s\n", pax, &pax[4]);
    puts(pax);
    puts(&pax[4]);
    return 0;
}
```

3.2. Escribir y ejecutar un programa que imprima su nombre y dirección.

3.3. Escribir y ejecutar un programa que imprima una página de texto con no más de 40 caracteres por línea.

3.4. Depurar el programa siguiente:

```
#include <stdio.h>

void main()
{
    printf("El lenguaje de programación C")
}
```

3.5. Escribir un programa que imprima la letra B con asteriscos,

```
*****
*   *
*   *
*   *
*****
*   *
*   *
*   *
*****
```

CAPÍTULO 4

OPERADORES Y EXPRESIONES

CONTENIDO

- 4.1. Operadores y expresiones.
- 4.2. Operador de asignación.
- 4.3. Operadores aritméticos.
- 4.4. Operadores de incrementación y decrementación.
- 4.5. Operadores relacionales.
- 4.6. Operadores lógicos.
- 4.7. Operadores de manipulación de bits.
- 4.8. Operador condicional.
- 4.9. Operador `comma`.
- 4.10. Operadores especiales, `:` `()` , `El`.
- 4.11. El operador `sizeof`.
- 4.12. Conversiones de tipos.
- 4.13. Prioridad y asociatividad.
- 4.14. *Resumen*.
- 4.15. *Ejercicios*.
- 4.16. *Problemas*.

INTRODUCCIÓN

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Este capítulo muestra como C hace uso de los operadores y expresiones para la resolución de operaciones. Los operadores fundamentales que se analizan en el capítulo son:

- aritméticos, lógicos y relacionales;
- de manipulación de bits;
- condicionales;
- especiales.

Además se analizarán las conversiones de tipos de datos y las reglas que seguirá el compilador cuando concurren en una misma expresión diferentes tipos de operadores. Estas reglas se conocen como *prioridad* y *asociatividad*.

CONCEPTOS CLAVE

- Asignación.
- Asociatividad.
- Conversión explícita.
- Conversiones de tipos.
- Evaluación en cortocircuito.
- Expresión.
- Incrementación/decrementación.
- Manipulación de bits.
- Operador.
- Operador `sizeof`.
- Prioridad/precedencia.

4.1. OPERADORES Y EXPRESIONES

Los programas C constan de datos, sentencias de programas y *expresiones*. Una expresión es, normalmente, una ecuación matemática, tal como 3+5. En esta expresión, el símbolo más (+) es el *operador* de suma, y los números 3 y 5 se llaman *operandos*. En síntesis, una *expresión* es una secuencia de operaciones y operandos que especifica un cálculo.

Cuando se utiliza el **+** entre números (o variables) se denomina *operador binario*, debido a que el operador **+** suma dos números. Otro tipo de operador de C es el *operador unitario* («unario»), que actúa sobre un Único valor. Si la variable x contiene el valor 5, -x es el valor -5. El signo menos (-) es el operador unitario menos.

C soporta un conjunto potente de operadores unarios, binarios y de otros tipos.

Sintaxis

Variable = *expresión*

variable **identificador** válido C declarado como variable.

expresión una **constante**, otra variable a la que **se** ha asignado **previamente un valor** o una fórmula que **se ha** evaluado y cuyo **tipo** es el **de variable**.

Una *expresión* es un elemento de un programa que toma un valor. En algunos casos puede también realizar una operación.

Las expresiones pueden ser valores constantes o variables simples, tales como 25 o 'Z'; pueden ser valores o variables combinadas con operadores (a++, m==n, etc.); o bien pueden ser valores combinados con funciones tales como toupper('b') .

4.2. OPERADOR DE ASIGNACIÓN

El operador = asigna el valor de la expresión derecha a la variable situada a su izquierda.

```
codigo = 3467;
fahrenheit = 123.456;
coordX = 525;
coordY = 725;
```

Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así,

```
a = b = c = 45;
```

equivale a

```
a = (b = (c = 45));
```

o dicho de otro modo, a las variables a, b y c se asigna el valor 45.

Esta propiedad permite inicializar varias variables con una sola sentencia

```
int a, b, c;
a = b = c = 5;     /* se asigna 5 a las variables a, b y c */
```

Además del operador de asignación =, C proporciona cinco operadores de asignación adicionales. En la Tabla 4.1 aparecen los seis operadores de asignación.

Estos operadores de asignación actúan como una notación abreviada para expresiones utilizadas con frecuencia. Así, por ejemplo, si se desea multiplicar 10 por i, se puede escribir

```
i = i * 10;
```

Tabla 4.1. Operadores de asignación de C.

Símbolo	uso	Descripción
=	$a = b$	Asigna el valor de b a a .
*=	$a *= b$	Multiplica a por b y asigna el resultado a la variable a .
/=	$a /= b$	Divide a entre b y asigna el resultado a la variable a .
%=	$a \% = b$	Fija a al resto de a/b .
+=	$a += b$	Suma b y a y lo asigna a la variable a .
-=	$a -= b$	Resta b de a y asigna el resultado a la variable a .

C proporciona un operador abreviado de asignación ($*=$), que realiza una asignación equivalente

$i *= 10;$ *equivale a* $i = i * 10;$

Tabla 4.2. Equivalencia de operadores de asignación.

Operador	Sentencia abreviada	Sentencia no abreviada
+=	$m += n$	$m = m + n;$
-=	$m -= n$	$m = m - n;$
*=	$m *= n$	$m = m * n;$
/=	$m /= n$	$m = m / n;$
%=	$m \% = n$	$m = m \% n;$

Estos operadores de asignación no siempre se utilizan, aunque algunos programadores C se acostumbran a su empleo por el ahorro de escritura que suponen.

4.3. OPERADORES ARITMÉTICOS

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Los operadores aritméticos C siguen las reglas algebraicas típicas de jerarquía o prioridad. Estas reglas especifican la precedencia de las operaciones aritméticas.

Considere la expresión

$$3 + 5 * 2$$

¿Cuál es el valor correcto, $16 (8 * 2)$ o $13 (3 + 10)$? De acuerdo a las citadas reglas, la multiplicación se realiza antes que la suma. Por consiguiente, la expresión anterior equivale a:

$$3 + (5 * 2)$$

En C las expresiones interiores a paréntesis se evalúan primero; a continuación, se realizan los operadores unitarios, seguidos por los operadores de multiplicación, división, resto, suma y resta.

Tabla 4.3. Operadores aritméticos.

Operador	Tipos enteros	Tipos reales	Ejemplo
+	Suma	Suma	$x + y$
-	Resta	Resta	$b - c$
*	Producto	Producto	$x * y$
/	División entera: cociente	División en coma flotante	$b / 5$
%	División entera: resto		$b \% 5$

Tabla 4.4. Precedencia de operadores matemáticos básicos.

Operador	Operación	Nivel de precedencia
+, -	+25, -6.745	1
*, /, %	5*5 <i>es</i> 25	2
	25/5 <i>es</i> 5	
	25%6 <i>es</i> 1	
+, -	2+3 <i>es</i> 5	3
	2-3 <i>es</i> -1	

Obsérvese que los operadores + y -, cuando se utilizan delante de un operador, actúan como operadores unitarios más y menos.

+75 /* 75 significa que *es* positivo */
 -154 /* 154 significa que *es* negativo */

Ejemplo 4.1

1. ¿Cuál es el resultado de la expresión: $6 + 2 * 3 - 4/2$?

$$\begin{array}{r}
 6 + \underline{2 * 3} - 4 / 2 \\
 6 + 6 - 4/2 \\
 \underline{6 + 6} - 2 \\
 12 - 2 \\
 \hline
 10
 \end{array}$$

2. ¿Cuál es el resultado de la expresión: $5 * 5(5 + (6-2) + 1)$?

$$\begin{array}{r}
 5 * (5 + (6-2) + 1) \\
 5 * (\underline{5 + 4 + 1}) \\
 5 * 10 \\
 50
 \end{array}$$

3. ¿Cuál es el resultado de la expresión: $7 - 6/3 + 2 * 3/2 - 4/2$?

$$\begin{array}{r}
 7 - \underline{6/3} + 2 * 3/2 - 4/2 \\
 7 - 2 + \underline{2 * 3/2} - 4/2 \\
 7 - 2 + 3 - \underline{4/2} \\
 7 - 2 + 3 - 2 \\
 \underline{5} + 3 - 2 \\
 8 - 2 \\
 \hline
 6
 \end{array}$$

4.3.1. Asociatividad

En una expresión tal como

$$3 * 4 + 5$$

el compilador realiza primero la multiplicación —por tener el operador $*$ prioridad más alta— y luego la suma, por tanto, produce 17. Para forzar un orden en las operaciones se deben utilizar paréntesis

$$3 * (4 + 5)$$

produce 27, ya que $4+5$ se realiza en primer lugar.

La *asociatividad* determina el orden en que se agrupan los operadores de igual prioridad; es decir, de izquierda a derecha o de derecha a izquierda. Por ejemplo,

$$x - y + z \quad \text{se agrupa como} \quad (x - y) + z$$

ya que $-$ y $+$, con igual prioridad, tienen asociatividad de izquierda a derecha. Sin embargo,

$$x = y = z$$

se agrupa como

$$x = (y = z)$$

dado que su asociatividad es de derecha a izquierda.

Tabla 4.5. Prioridad y asociatividad.

Prioridad (mayor a menor)	Asociatividad
$+$, $-$ (<i>unarios</i>)	izquierda-derecha (\rightarrow)
$*$, $/$, $\%$	izquierda-derecha (\rightarrow)
$+$,	izquierda-derecha (\rightarrow)

Ejemplo 4.2

¿Cuál es el resultado de la expresión: $7 * 10 - 5 \% 3 * 4 + 9$?

Existen tres operadores de prioridad más alta ($*$, $\%$ y $*$)

$$70 - 5 \% 3 * 4 + 9$$

La asociatividad es de izquierda a derecha, por consiguiente se ejecuta a continuación $\%$

$$70 - 2 * 4 + 9$$

y la segunda multiplicación se realiza a continuación, produciendo

$$70 - 8 + 9$$

Las dos operaciones restantes son de igual prioridad y como la asociatividad es a izquierda, se realizará la resta primero y se obtiene el resultado

$$62 + 9$$

y, por último, se realiza la suma y se obtiene el resultado final de

4.3.2. Uso de paréntesis

Los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su prioridad y asociatividad. Las subexpresiones entre paréntesis se evalúan en primer lugar según el modo estándar y los resultados se combinan para evaluar la expresión completa. Si los paréntesis están «anidados» —es decir, un conjunto de paréntesis contenido en otro— se ejecutan en primer lugar los paréntesis más internos. Por ejemplo, considérese la expresión

$$(7 * (10 - 5) \% 3) * 4 + 9$$

La subexpresión $(10 - 5)$ se evalúa primero, produciendo

$$(7 * 5 \% 3) * 4 + 9$$

A continuación se evalúa de izquierda a derecha la subexpresión $(7 * 5 \% 3)$

$$(35 \% 3) * 4 + 9$$

seguida de

$$2 * 4 + 9$$

Se realiza a continuación la multiplicación, obteniendo

$$8 + 9$$

y la suma produce el resultado final

$$17$$

Precaución

Se debe tener cuidado en la escritura de expresiones que contengan dos o más operaciones para asegurarse que **se evalúan** en el orden previsto. Incluso aunque no se requieran paréntesis, deben utilizarse para clarificar el orden concebido de evaluación y escribir expresiones complicadas en términos de expresiones más simples. Es importante, sin embargo, que los paréntesis estén equilibrados - cada paréntesis a la izquierda tiene un correspondiente paréntesis a la derecha que aparece posteriormente en la **expresión**— ya que existen paréntesis **desequilibrados** se producirá un error de compilación.

$$((8 - 5) + 4 - (3 + 7)) \quad \text{error de compilación, falta paréntesis final a la derecha}$$

4.4. OPERADORES DE INCREMENTACIÓN Y DECREMENTACIÓN

De las características que incorpora C, una de las más Útiles son los operadores de incremento ++ y decremento --. Los operadores ++ y --, denominados de *incrementación* y *decrementación*, suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable.

Tabla 4.6. Operadores de incrementación (++) y decrementación (--).

Incrementación	Decrementación
++n	--n
n += 1	n -= 1
n = n + 1	n = n - 1

Por consiguiente,

```
a + t
```

es igual que

```
a = a + 1
```

Estos operadores tienen la propiedad de que pueden utilizarse como sufijo o prefijo, el resultado de la expresión puede ser distinto, dependiendo del contexto.

Las sentencias

```
++n;
```

```
n++;
```

tienen el mismo efecto: así como

```
--n;
```

```
n--;
```

Sin embargo, cuando se utilizan como expresiones tales como

```
m = n++;
```

```
printf(" n = %d", n--);
```

el resultado es distinto si se utilizan como prefijo.

```
m = ++n;
```

```
printf(" n = %d", --n);
```

t+n produce un valor que es mayor en uno que el de **n++**, y **--n** produce un valor que es menor en uno que el valor de **n--**. Supongamos que

```
n = 8;
```

```
m = ++n; /* incrementa n en 1, 9, y lo asigna a m */
```

```
n = 9;
```

```
printf(" n = %d", --n); /* decrementa n en 1, 8, y lo pasa a printf() */
```

```
n = 8;
```

```
m = n++; /* asigna n(8) a m, después incrementa n en 1 (9) */
```

```
n = 9;
```

```
printf(" n = %d", n--); /* pasa n(9) a printf(), después decrementa n */
```

En este otro ejemplo,

```
int a = 1, b;
```

```
b = a++; /* b vale 1 y a vale 2 */
```

```
int a = 1, b;
```

```
b = ++a; /* b vale 2 y a vale 2 */
```

Si los operadores `++` y `--` están de prefijos, la operación de incremento o decremento se efectúa antes que la operación de asignación; si los operadores `++` y `--` están de sufijos, la asignación se efectúa en primer lugar y la incrementación o decrementación a continuación.

Ejemplo

```
int i = 10;
int j;
...
j = i++;
```

Ejemplo 4.3

Demostración del funcionamiento de los operadores de incremento/decremento.

```
#include <stdio.h>
/* Test de operadores ++ y -- */
void main()
{
    int m = 45, n = 75;
    printf( " m = %d, n = %d\n",m,n);
    ++m;
    --n;
    printf( " m = %d, n = %d\n",m,n);
    m++;
    n--;
    printf( " m = %d, n = %d\n",m,n);
}
```

Ejecución

```
m = 45, n = 75
m = 46, n = 74
m = 47, n = 73
```

En este contexto, el orden de los operadores es irrelevante.

Ejemplo 4.4

Diferencias entre operadores de preincremento y postincremento.

```
#include <stdio.h>
/* Test de operadores ++ y -- */
void main()
{
    int m = 99, n;
    n = ++m;
    printf("m = %d, n = %d\n",m,n);
    n = m++;
    printf("m = %d, n = %d\n",m,n);
    printf("m = %d \n",m++);
    printf("m = %d \n",++m);
}
```

Ejecución

```

m = 100, n = 100
m = 101, n = 100
m = 101
m = 103

```

Ejemplo 4.5

Orden de evaluación no predecible en expresiones.

```

#include <stdio.h>

void main()
{
    int n = 5, t;
    t = ++n * --n;
    printf("n = %d, t = %d\n", n, t);
    printf("%d %d %d\n", ++n, ++n, ++n);
}

```

Ejecución

```

n = 5, t = 25
8 7 6

```

Aunque parece que aparentemente el resultado de `t` será 30, en realidad es 25, debido a que en la asignación de `t`, `n` se incrementa a 6 y a continuación se decreta a 5 antes de que se evalúe el operador producto, calculando `5 * 5`. Por último, las tres subexpresiones se evalúan de derecha a izquierda sera `8 7 6` al contrario de `6 7 8` que parece que aparentemente se producirá.

4.5. OPERADORES RELACIONALES

C no tiene tipos de datos lógicos o booleanos, como Pascal, para representar los valores verdadero (*true*) y falso (*false*). En su lugar se utiliza el tipo `int` para este propósito, con el valor entero 0 que representa a falso y distinto de cero a verdadero.

<i>falso</i>	<i>cero</i>
<i>verdadero</i>	distinto de <i>cero</i>

Operadores tales como `>=` y `==` que comprueban una relación entre dos operandos se llaman **operadores relacionales** y se utilizan en expresiones de la forma

expresión, operador-relacional expresión

expresión, y expresión expresiones compatibles C
operador-relacional un operador de la tabla 4.7

Los operadores relacionales se usan normalmente en sentencias de selección (*if*) o de iteración (*while*, *for*), que sirven para comprobar una condición. Utilizando operadores relacionales se realizan operaciones de igualdad, desigualdad y diferencias relativas. La Tabla 4.7 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: *char*, *int*, *float*, *double*, etc.

Cuando se utilizan los operadores en una expresión, el operador relacional produce un 0, o un 1, dependiendo del resultado de la condición. 0 se devuelve para una condición *falsa*, y 1 se devuelve para una condición *verdadera*. Por ejemplo, si se escribe

```
c = 3 < 7;
```

la variable *c* se pone a 1, dado que como 3 es menor que 7, entonces la operación *<* devuelve un valor de 1, que se asigna a *c*.

Precaución

Un error típico, incluso entre programadores experimentales, es confundir el operador de asignación (*=*) con el operador de igualdad (*==*).

Tabla 4.7. Operadores relacionales de C.

Operador	Significado	Ejemplo
<i>==</i>	<i>Igual a</i>	<i>a == b</i>
<i>!=</i>	<i>No igual a</i>	<i>a != b</i>
<i>></i>	<i>Mayor que</i>	<i>a > b</i>
<i><</i>	<i>Menor que</i>	<i>a < b</i>
<i>>=</i>	<i>Mayor o igual que</i>	<i>a >= b</i>
<i><=</i>	<i>Menor o igual que</i>	<i>a <= b</i>

Ejemplo

- Si *x*, *a*, *b* y *c* son de tipo *double*, *numero* es de tipo *int* e *inicial* es de tipo *char*, las siguientes expresiones booleanas son válidas:

```
x < 5.75
b * b >= 5.0 * a * c
numero == 100
inicial != '5'
```

- En datos numéricos, los operadores relacionales se utilizan normalmente para comparar. Así, si

```
x = 3.1
la expresión
x < 7.5
```

produce el valor 1 (*true*). De modo similar si

```
numero = 27
```

la expresión

```
numero == 100
```

produce el valor 0 (*false*).

- Los caracteres se comparan utilizando los códigos numéricos (véase Apéndice B, código ASCII)

'A' < 'C' es 1, verdadera (*true*), ya que A es el código 65 y es menor que el código 67 de C.

'a' < 'c' es 1, verdadera (*true*): a (código 97) y b (código 99).

'b' < 'B' es 0, falsa (*false*) ya que b (código 98) no es menor que B (código 66).

Los operadores relacionales tienen menor prioridad que los operadores aritméticos, y asociatividad de izquierda a derecha. Por ejemplo,

```
m+5 <= 2 * n           equivale a           (m+5) <= (2 * n)
```

Los operadores relacionales permiten comparar dos valores. Así, por ejemplo (*if* significa si, se verá en el capítulo siguiente),

```
if (Nota-asignatura < 9)
```

comprueba si *Nota-asignatura* es menor que 9. En caso de desear comprobar si la variable y el número son iguales, entonces utilizar la expresión

```
if (Nota-asignatura == 9)
```

Si, por el contrario, se desea comprobar si la variable y el número no son iguales, entonces utilice la expresión

```
if (Nota-asignatura != 9)
```

- Las cadenas de caracteres no pueden compararse directamente. Por ejemplo,

```
char nombre[26];
```

```
gets(nombre)
```

```
if (nombre < "Marisa")
```

El resultado de la comparación es inesperado, no se están comparando alfabéticamente, lo que se compara realmente son las direcciones en memoria de ambas cadenas (punteros). Para una comparación alfabética entre cadenas se utiliza la función `strcmp()` de la biblioteca de C (`string.h`). Así,

```
if (strcmp(nombre, "Marisa") < 0) /* alfabéticamente nombre es menor */
```

4.6. OPERADORES LÓGICOS

Además de los operadores matemáticos, C tiene también operadores lógicos. Estos operadores se utilizan con expresiones para devolver un valor verdadero (cualquier entero distinto de cero) o un valor falso (0). Los operadores lógicos se denominan también operadores *booleanos*, en honor de George Boole, creador del álgebra de Boole.

Los operadores lógicos de C son: **not (!)**, **and (&&)** y **or (||)**. El operador lógico **!** (*not, no*) produce *falso* (cero) si su operando es verdadero (distinto de cero) y viceversa. El operador lógico **&&** (*and, y*) produce verdadero *sólo* si ambos operandos son verdadero (no cero); si cualquiera de los operandos es falso produce *falso*. El operador lógico **||** (*or, o*) produce verdadero si cualquiera de los operandos es verdadero (distinto de cero) y produce falso sólo si ambos operandos son falsos. La Tabla 4.8 muestra los operadores lógicos de C.

Tabla 4.8. Operadores lógicos.

Operador	Operación lógica	Ejemplo
Negación (!)	No lógica	!(x >= y)
Y lógica (&&)	operando-1 && operando-2	m < n && i > j
O lógica	operando-1 operando-2	m = 5 n != 10

Tabla 4.9. Tabla de verdad del operador lógico NOT (!).

Operando (a)	NOT a
Verdadero (1)	Falso (0)
Falso (0)	Verdadero (1)

Tabla 4.10. Tabla de verdad del operador lógico AND.

Operandos		
a	b	a && b
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Falso (0)
Falso (0)	Verdadero (1)	Falso (0)
Falso (0)	Falso (0)	Falso (0)

Tabla 4.11. Tabla de verdad del operador lógico OR (||).

Operandos		
a	b	a b
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Verdadero (1)
Falso (0)	Verdadero (1)	Verdadero (1)
Falso (0)	Falso (0)	Falso (0)

Al igual que los operadores matemáticos, el valor de una expresión formada con operadores lógicos depende de: (*u*)el operador y (*h*)sus argumentos. Con operadores lógicos existen sólo dos valores posibles para expresiones: **verdadero** y **falso**. La forma más usual de mostrar los resultados de operaciones lógicas es mediante las denominadas **tablas de verdad**, que muestran como funcionan cada uno de los operadores lógicos.

Ejemplo

```
!(x+7 == 5)
(anum > 5) && (Respuesta == 'S')
(bnum > 3) || (Respuesta == 'N')
```

Los operadores lógicos se utilizan en expresiones condicionales y mediante sentencias `if`, `while` o `for`, que se analizarán en capítulos posteriores. Así, por ejemplo, la sentencia `if` (**si la condición es verdadera/falsa**...) se utiliza para evaluar operadores lógicos.

```
1. if ((a < b) && (c > d))
   {
     puts("Los resultados no son válidos");
   }
```

Si la variable *a* es menor que *b* y, al mismo tiempo, *c* es mayor que *d*, entonces visualizar el mensaje: Los resultados no son válidos.

```
2. if ((ventas > 50000) || (horas < 100))
{
    prima = 100000;
}
```

Si la variable *ventas* es mayor 50000 **o bien** la variable *horas* es menor que 100, entonces asignar a la variable *prima* el valor 100000.

```
3. if (!(ventas < 2500))
{
    prima = 12500;
}
```

En este ejemplo, si *ventas* es mayor que o igual a 2500, se inicializará *prima* al valor 12500.

El operador `!` tiene prioridad más alta que `&&`, que a su vez tiene mayor prioridad que `||`. La asociatividad es de izquierda a derecha.

La precedencia de los operadores es: los operadores matemáticos tienen precedencia sobre los operadores relacionales, y los operadores relacionales tienen precedencia sobre los operadores lógicos. La siguiente sentencia:

```
if ((ventas < sal_min * 3 && ayos > 10 * iva)...
```

equivale a

```
if ((ventas < (sal_min * 3)) && (ayos > (10 * iva)))...
```

4.6.1. Evaluación en cortocircuito

En C los operandos de la izquierda de `&&` y `||` se evalúan siempre en primer lugar; si el valor del operando de la izquierda determina de forma inequívoca el valor de la expresión, el operando derecho no se evalúa. Esto significa que si el operando de la izquierda de `&&` es falso o el de `||` es verdadero, el operando de la derecha no se evalúa. Esta propiedad se denomina *evaluación en cortocircuito* y se debe a que si *p* es falso, la condición *p && q* es falsa con independencia del valor de *q* y de este modo C no evalúa *q*. De modo similar si *p* es verdadera la condición *p || q* es verdadera con independencia del valor de *q* y C no evalúa a *q*.

Ejemplo 4.6

Supongamos que se evalúa la expresión

```
(x > 0.0) && (log(x) >= 0.5)
```

Dado que en una operación lógica *Y* (`&&`) si el operando de la izquierda (*x > 0.0*) es falso (*x es negativo o cero*), la expresión lógica se evalúa a falso, y en consecuencia, no es necesario evaluar el segundo operando. En el ejemplo anterior la expresión evita calcular el logaritmo de números (*x*) negativos o cero.

La evaluación en **cortocircuito** tiene dos beneficios importantes:

1. Una expresión *booleana* se puede utilizar para *guardar* una operación potencialmente insegura en una segunda expresión *booleana*.
2. Se puede ahorrar una considerable cantidad de tiempo en la evaluación de condiciones complejas.

Ejemplo 4.7

Los *beneficios anteriores se aprecian en la expresión booleana*

```
(n != 0) && (x < 1.0/n)
```

ya que no se puede producir un error de división por cero al evaluar esta expresión, pues si *n* es 0, entonces la primera expresión

```
n != 0
```

es falsa y la segunda expresión

```
x < 1.0/n
```

no se evalúa.

De modo similar, tampoco se producirá un error de división por cero al evaluar la condición

```
(n == 0) || (x >= 5.0/n)
```

ya que si *n* es 0, la primera expresión

```
n == 0
```

es verdadera y entonces no se evalúa la segunda expresión

```
x >= 5.0/n
```

Aplicación

Dado el test condicional

```
if ((7 > 5) || (ventas < 30) && (30 != 30))...
```

C examina sólo la primera condición ($7 > 5$), ya que como es verdadera, la operación lógica $|$ (O) será verdadera, sea cual sea el valor de la expresión que le sigue.

Otro ejemplo es el siguiente:

```
if ((8 < 4) && (edad > 18) && (letra-inicial == 'Z'))...
```

En este caso, C examina la primera condición y su valor es falso; por consiguiente, sea cual sea el valor que sigue al operador $\&\&$, la expresión primitiva será falsa y toda la subexpresión a la derecha de $(8 < 4)$ no se evalúa por C.

Por último, en la sentencia

```
if ((10 > 4) || (num == 0)) ...
```

la operación `num == 0` nunca se evaluará.

4.6.2. Asignaciones booleanas (lógicas)

Las sentencias de asignación booleanas se pueden escribir de modo que dan como resultado un valor de tipo `int` que será cero o uno.

Ejemplo

```
int edad, MayorDeEdad, juvenil;
scanf("%d",&edad);
MayorDeEdad = (edad > 18); /* asigna el valor de edad > 18 MayorDeEdad.
    Cuando edad es mayor que 18, MayorDeEdad es 1 , sino 0 */
juvenil = (edad > 15) && (edad <= 18); /* asigna 1 a juvenil si edad está
    comprendida entre 15(mayor que 15) y 18 (inclusive 18). */
```

Ejemplo 4.8

Las sentencias de asignación siguientes asignan valores cero o uno a los dos tipos de variables *int*, *rango* y *es-letra*. La variable *rango* es 1 (*true*) si el valor de *n* está en el rango *-100 a 100*; la variable *es-letra* es 1 (*verdadera*) si *car* es una letra mayúscula o minúscula.

```
a. rango = (n > -100) && (n < 100);
b. es-letra = (( 'A' <= car) && (car <= 'Z ')) ||
    (('a' <= car) && (car <= 'z '));
```

La expresión de *a* es 1 (*true*) si *n* cumple las condiciones expresadas (*n* mayor de *-100* y menor de *100*); en caso contrario es 0 (*false*). La expresión *b* utiliza los operadores *&&* y *||*. La primera subexpresión (antes de *||*) es 1 (*true*) si *car* es una letra mayúscula; la segunda subexpresión (después de *||*) es 1 (*true*) si *car* es una letra minúscula. En resumen, *es-letra* es 1 (*true*) si *car* es una letra, y 0 (*false*) en caso contrario.

4.7. OPERADORES DE MANIPULACIÓN DE BITS

Una de las razones por las que C se ha hecho tan popular en computadoras personales es que el lenguaje ofrece muchos operadores de manipulación de bits a bajo nivel.

Los operadores de manipulación o tratamiento de bits (*bitwise*) ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Estas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador.

Cada operador de manipulación de bits realiza una operación lógica bit a bit sobre datos internos. Los operadores de manipulación de bits se aplican sólo a variables y constantes *char*, *int* y *long*, y no a datos en coma flotante. Dado que los números binarios constan de 1,s y 0,s (denominados *bits*), estos 1 y 0 se manipulan para producir el resultado deseado para cada uno de los operadores.

Las siguientes tablas de verdad describen las acciones que realizan los diversos operadores sobre los diversos patrones de bit de un dato *int* (*char* o *long*).

Tabla 4.12. Operadores lógicos bit a bit.

Operador	Operación
&	Y (AND) lógica bit a bit
	O (OR) lógica (inclusiva) bit a bit
^	O (XOR) lógica (exclusiva) bit a bit (OR exclusive, XOR)
~	Complemento a uno (inversión de todos los bits)
<<	Desplazamiento de bits a izquierda
>>	Desplazamiento de bits a derecha

$A \& B == C$ $0 \& 0 == 0$ $0 \& 1 == 0$ $1 \& 0 == 0$ $1 \& 1 == 1$	$A B == C$ $0 0 == 0$ $0 1 == 1$ $1 0 == 1$ $1 1 == 1$	$A \wedge B == C$ $0 \wedge 0 == 0$ $0 \wedge 1 == 1$ $1 \wedge 0 == 1$ $1 \wedge 1 == 0$	$A \quad \neg A$ $1 \quad 0$ $0 \quad 1$
---	--	---	--

Ejemplo

1. Si se aplica el operador & de manipulación de bits a los números 9 y 14, se obtiene un resultado de 8. La Figura 4.1 muestra cómo se realiza la operación.

$$\begin{array}{rcll}
 2. \quad (\&) \quad 0x3A6B & = & 0011 & 1010 & 0110 & 1011 \\
 & \quad 0x00F0 & = & 0000 & 0000 & 1111 & 0000 \\
 & \quad 0x3A6B \ \& \ 0x00F0 & = & 0000 & 0000 & 0110 & 0000 = 0x0060
 \end{array}$$

$$\begin{array}{rcll}
 3. \quad (|) \quad 152 \quad 0x0098 & = & 0000 & 0000 & 1001 & 1000 \\
 & \quad 5 \quad 0x0005 & = & 0000 & 0000 & 0000 & 0101 \\
 & \quad 152 \ | \ 5 & = & 0000 & 0000 & 1001 & 1101 = 0x009d
 \end{array}$$

$$\begin{array}{rcll}
 4. \quad (\wedge) \quad 83 \quad 0x53 & = & 0101 & 0011 \\
 & \quad 204 \quad 0xcc & = & 1100 & 1100 \\
 & \quad 83 \wedge 204 & = & 1001 & 1111 = 0x9f
 \end{array}$$

$$\begin{array}{rcll}
 9 \text{ decimal equivale a} & 1 & 0 & 0 & 1 & \text{binario} \\
 & \& & \& & \& & \& \\
 14 \text{ decimal equivale a} & 1 & 1 & 1 & 0 & \text{binario} \\
 = & 1 & 0 & 0 & 0 & \text{binario} \\
 - & & 8 & & & \text{decimal}
 \end{array}$$

Figura 4.1. Operador & de manipulación de bits.

4.7.1. Operadores de asignación adicionales

Al igual que los operadores aritméticos, los operadores de asignación abreviados están disponibles también para operadores de manipulación de bits. Estos operadores se muestran en la Tabla 4.13.

Tabla 4.13. Operadores de asignación adicionales.

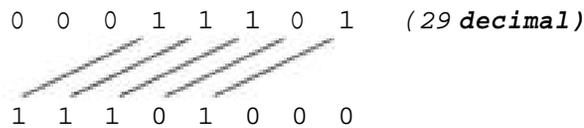
Símbolo	uso	Descripción
<<=	a <<= b	Desplaza a a la izquierda b bits y asigna el resultado a a.
>>=	a >>= b	Desplaza a a la derecha b bits y asigna el resultado a a.
&=	a &= b	Asigna a a el valor a&b.
^=	a ^= b	Establece a a a^b.
=	a = b	Establece a a a b.

4.7.2. Operadores de desplazamiento de bits (>>, <<)

Equivalen a la instrucción SHR (>>) y SHL (<<) de los microprocesadores 80x86. Efectúa un desplazamiento a la derecha (>>) o a la izquierda (<<) de n posiciones de los bits del operando, siendo n un número entero. El número de bits desplazados depende del valor a la derecha del operador. Los formatos de los operadores de desplazamiento son:

1. `valor << numero-de-bits;`
2. `valor >> numero-de-bits;`

El *valor* puede ser una variable entera o carácter, o una constante. El *numero-de-bits* determina cuántos bits se desplazarán. La Figura 4.2 muestra lo que sucede cuando el número 29 (binario 0001 1101) se desplaza a la izquierda tres bits con un desplazamiento a la izquierda bit a bit (<<).



Después de tres desplazamientos

Figura 4.2. Desplazamiento a la izquierda tres posiciones de los bits del número binario equivalente a 29.

Supongamos que la variable `num1` contiene el valor 25, si se desplaza tres posiciones (`num << 3`), se obtiene el nuevo número 200 (11001000 en binario).

```
int num1 = 25;           /* 00011001 binario */
int despl, desp2;

despl = num1 << 3;      /* 11001000 binario */
```

En los siguientes ejemplos se desplazan los bits de una variable a la derecha y a la izquierda. El resultado es una división y una multiplicación respectivamente.

```
int x ,y,d ;
x=y= 24;

d = x >> 2 ;           /* 0x18>>2 = 0001 1000 >> 2
                       = 0000 0110 = 6 ( división por 4 ) */

d = y << 2;           /* 0x18<<2 = 0001 1000 >> 2
                       = 0110 0000 = 0x60 (96) (multiplicación por 4)*/
```

*4.7.3. Operadores de direcciones

Son operadores que permiten manipular las direcciones de las variables y registros en general:

```
*expresión
&valor_i (lvalue)
registro.miembro
puntero-hacia-registro -> miembro
```

Tabla 4.14. Operadores de direcciones.

Operador	Acción
*	Lee o modifica el valor apuntado por la expresión. Se corresponde con un puntero y el resultado es del tipo apuntado.
&	Devuelve un puntero al objeto utilizado como operando, que debe ser un <i>lvalue</i> (variable dotada de una dirección de memoria). El resultado es un puntero de tipo idéntico al del operando. Permite acceder a un miembro de un dato agregado (unión, estructura).
->	Accede a un miembro de un dato agregado (unión, estructura) apuntado por el operando de la izquierda.

4.8. OPERADOR CONDICIONAL

El operador condicional, `?:`, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas (derecha a izquierda).

Al ser un operador ternario requiere tres operandos. El operador *condicional* se utiliza para reemplazar a la sentencia `if-else` lógica en algunas situaciones. El formato del operador condicional es:

```
expresion-c ? expresion-v : expresion-f;
```

Se evalúa *expresion-c* y su valor (cero = falso, distinto de cero = verdadero) determina cuál es la expresión a ejecutar; si la condición es verdadera se ejecuta *expresion_v* y si es falsa se ejecuta *expresion-f*.

La Figura 4.3 muestra el funcionamiento del operador condicional.

```
(ventas > 150000)    ?    comision = 100           comision = 0;

    si ventas es mayor
    que 150.000 se
    ejecuta:

comision = 100

    si ventas no es
    mayor que 150.000 se
    ejecuta:

comision = 0
```

Figura 4.3. Formato de un operador condicional.

Otros ejemplos del uso del operador `?:` son:

```
n >= 0 ? 1 : -1          /*I si n es positivo, -1 si es negativo */
m >= n ? m : n          /* devuelve el mayor valor de m y n */

/*escribe x, y escribe el carácter fin de línea(\n) si x%5(resto 5) es
o, en caso contrario un tabulador(\t) */
printf("%d %c", x, x%5 ? '\t' : '\n');
```

La precedencia de `?:` es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es a derechas.

4.9. OPERADOR COMA

El *operador coma* permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha. La expresión más a la derecha determina el resultado global. El uso del operador coma es como sigue:

```
expresión , expresión , expresión , ... , expresión
```

Cada expresión se evalúa comenzando desde la izquierda y continuando hacia la derecha. Por ejemplo, en

```
int i = 10, j = 25;
```

dado que el operador coma se asocia de izquierda a derecha, la primera variable está declarada e inicializada antes que la segunda variable *j*. Otros ejemplos son:

```
i++, j++;           equivale a      i++; j++;
i++, j++, k++;     equivale a      i++; j++; k++;
```

El operador coma tiene la menor prioridad de todos los operadores C, y se asocia de izquierda a derecha.

El resultado de la expresión global se determina por el valor de *expresión*... Por ejemplo,

```
int i, j, resultado;
resultado = j = 10, i = j, ++i;
```

El valor de esta expresión y valor asignado a *resultado* es 11. En primer lugar, a *j* se asigna el valor 10, a continuación a *i* se asigna el valor de *j*. Por último, *i* se incrementa a 11.

La técnica del operador coma permite operaciones interesantes

```
i = 10;
j = (i = 12, i + 8);
```

Cuando se ejecute la sección de código anterior, *j* vale 20, ya que *i* vale 10 en la primera sentencia, en la segunda toma *i* el valor 12 y al sumar *i* + 8 resulta 20.

4.10. OPERADORES ESPECIALES (), [], {}

C admite algunos operadores especiales que sirven para propósitos diferentes. Entre ellos se destacan: `()`, `[]`.

4.10.1. El operador ()

El operador `()` es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explícitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función, resolver los conflictos de prioridad entre operadores.

4.10.2. El operador []

Sirve para dimensionar los arrays y designar un elemento de un array.

Ejemplos de ello:

```
double v[20];           /* define un array de 20 elementos */
printf("v[2] = %e",v[2]); /* escribe el elemento 2 de v */
return v[i-INFERIOR];  /* devuelve el elemento i-INFERIOR */
```

4.11. EL OPERADOR SIZEOF

Con frecuencia su programa necesita conocer el tamaño en bytes de un tipo de dato o variable. C proporciona el operador `sizeof`, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, array, registro, etc.). El formato del operador es

```
sizeof (nombre-variable)
sizeof (tipo_dato)
sizeof (expresión)
```

Ejemplo 4.9

Si se supone que el tipo `int` consta de cuatro bytes y el tipo `double` consta de ocho bytes, las siguientes expresiones proporcionarán los valores 1, 4 y 8 respectivamente

```
sizeof(char)
sizeof(unsigned int)
sizeof(double).
```

El operador `sizeof` se puede aplicar también a expresiones. Se puede escribir

```
printf("La variable k es %d bytes ", sizeof(k));
printf("La expresión a + b ocupa %d bytes ", sizeof (a + b));
```

El operador `sizeof` es un operador unitario, ya que opera sobre un valor Único. Este operador produce un resultado que es el tamaño, en bytes, del dato o tipo de dato especificados. Debido a que la mayoría de los tipos de datos y variables requieren diferentes cantidades de almacenamiento interno en computadores diferentes, el operador `sizeof` permite consistencia de programas en diferentes tipos de computadores.

El operador `sizeof` se denomina también *operador en tiempo de compilación*, ya que en tiempo de compilación, el compilador sustituye cada ocurrencia de `sizeof` en su programa por un valor entero sin signo (`unsigned`). El operador `sizeof` se utiliza en programación avanzada.

Ejercicio 4.1

Suponga que se desea conocer el tamaño, en bytes, de variables de coma flotante y de doble precisión de su computadora. El siguiente programa realiza esta tarea:

```
/* Imprime el tamaño de valores de coma flotante y double */
#include <stdio.h>

int main()
{
    printf("El tamaño de variables de coma flotante es %d \n",
           sizeof(float));
    printf("El tamaño de variables de doble precisión es %d \n",
           sizeof(double));
```

```

return 0;
}

```

Este programa producirá diferentes resultados en diferentes clases de computadores. Compilando este programa bajo C, el programa produce la salida siguiente:

```

El tamaño de variables de coma flotante es: 4
El tamaño de variables de doble precisión es: 8

```

4.12. CONVERSIONES DE TIPOS

Con frecuencia, se necesita convertir un valor de un tipo a otro sin cambiar el valor que representa. Las conversiones de tipos pueden ser *implícitas* (ejecutadas automáticamente) o *explícitas* (solicitadas específicamente por el programador). C hace muchas conversiones de tipos automáticamente:

- o C convierte valores cuando se asigna un valor de un tipo a una variable de otro tipo.
- o C convierte valores cuando se combinan tipos mixtos en expresiones.
- o C convierte valores cuando se pasan argumentos a funciones.

4.12.1. Conversión implícita

Los tipos fundamentales (básicos) pueden ser mezclados libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operandos de tipo más bajo se convierten en los de tipo más alto.

```

int i = 12;
double x = 4;
x = x+i;    /*valor de i se convierte en double antes de la suma */
x = i/5;    /* primero hace una división entera i/5==2, 2 se convierte a
              tipo doble: 2.0 y se asigna a x */

x = 4.0;
x = x/5     /* convierte 5 a tipo double, hace una división real: 0.8 y se
              asigna a x */

```

4.12.2. Reglas

Si cualquier operando es de tipo `char`, `short` o enumerado se convierte en tipo `int` y si los operandos tienen diferentes tipos, la siguiente lista determina a qué operación convertirá. Esta operación se llama *promoción integral*.

```

int
unsigned int
long
unsigned long
float
double

```

El tipo que viene primero en esta lista se convierte en el que viene segundo. Por ejemplo, si los tipos operandos son `int` y `long`, el operando `int` se convierte en `long`.

```

char c = 65;        /* 65 se convierte en tipo char permitido */
char c = 10000;    /* permitido, pero resultados impredecibles */

```

4.12.3. Conversión explícita

C fuerza la conversión explícita de tipos mediante el operador de molde (*cast*). El operador molde tiene el formato:

```
(tiponombre)valor      /* convierte valor a tiponombre */
(float)i;              /* convierte i a float */
(int)3.4;              /* convierte 3.4 a entero, 3*/
(int*) malloc(2*16);  /* convierte el valor devuelto por malloc: void*
                       a int*. Es una conversión de punteros. */
```

El operador *molde* (*tipo, cast*) tiene la misma prioridad que otros operadores unitarios tales como +, - y !

```
precios = (int)19.99 + (int)11.99;
```

4.13. PRIORIDAD Y ASOCIATIVIDAD

La prioridad o precedencia de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores C vienen en una tabla con dieciséis grupos. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

- Si dos operadores se aplican al mismo operando, el operador con mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- La asociatividad izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

Prioridad	Operadores	Asociatividad
1	x -> [] O	I - D
2	++ -- ~ ! - + & * sizeof	D - I
3	.* -->*	I - D
4	* / %	I - D
5	+ -	I - D
6	<< >>	I - D
7	< <= > >=	I - D
8	== !=	I - D
9	&	I - D
10	^	I - D
11		I - D
12	&&	I - D
13		I - D
14	?: (<i>expresión condicional</i>)	D - I
15	= * - /= %= += -=	D - I
16	, (<i>operador coma</i>)	I - D

I - D : Izquierda - Derecha.
 D - I : Derecha - Izquierda.

4.14. RESUMEN

Este capítulo examina los siguientes temas:

- Concepto de operadores y expresiones.
- Operadores de asignación: básicos y aritméticos.
- Operadores aritméticos, incluyendo +, -, *, / y % (módulos).
- Operadores de incremento y decremento. Estos operadores se aplican en formatos *pre* (anterior) y *post* (posterior). C permite aplicar estos operadores a variables que almacenan caracteres, enteros e incluso números en coma flotante.
- Operadores relacionales y lógicos que permiten construir expresiones lógicas. C no soporta un tipo lógico (*boolean*) predefinido y en su lugar considera 0 (cero) como *falso* y cualquier valor distinto de cero como *verdadero*,
- Operadores de manipulación de bits que realizan operaciones bit a bit (*bitwise*), AND, OR, XOR y NOT. C soporta los operadores de desplazamiento de bits << y >>.
- Operadores de asignación de manipulación de bits que ofrecen formatos abreviados para sentencias simples de manipulación de bits.
- El operador coma, que es un operador muy especial, separa expresiones múltiples en las mismas sentencias y requiere que el programa evalúe totalmente una expresión antes de evaluar la siguiente.
- La expresión condicional, que ofrece una forma abreviada para la sentencia alternativa simple-doble *if-else*, que se estudiará en el capítulo siguiente.
- Operadores especiales: (), [].
- Conversión de tipos (*typecasting*) o moldeado, que permite forzar la conversión de tipos de una expresión.
- Reglas de prioridad y asociatividad de los diferentes operadores cuando se combinan en expresiones.
- El operador `sizeof`, que devuelve el tamaño en bytes de cualquier tipo de dato o una variable.

4.15. EJERCICIOS

4.1. Determinar el valor de las siguientes expresiones aritméticas:

15 / 12	15 % 12
24 / 12	24 % 12
123 / 100	123 % 100
200 / 100	200 % 100

4.2. ¿Cuál es el valor de cada una de las siguientes expresiones?

a) $15 * 14 - 3 * 7$
 b) $-4 * 5 * 2$
 c) $(24 + 2 * 6) / 4$

d) $a / a / a * b$
 e) $3 + 4 * (8 * (4 - (9 + 3) / 6))$
 f) $4 * 3 * 5 + 8 * 4 * 2 - 5$
 g) $4 - 40 / 5$
 h) $(-5) \% (-2)$

4.3. Escribir las siguientes expresiones aritméticas como expresiones de computadora: La potencia puede hacerse con la función `pow()`, por ejemplo $(x + y)^2 == \text{pow}(x+y, 2)$

a) $\frac{x}{y} + 1$ e) $(a+b) \frac{c}{d}$

$$b) \frac{x+y}{x-y}$$

$$f) [(a+b)^2]^2$$

$$c) x + \frac{Y}{z}$$

$$g) \frac{xy}{1-4x}$$

$$\frac{x + \frac{Y}{z}}{z}$$

$$h) \frac{xy}{mn}$$

$$d) \frac{b}{c+d}$$

$$i) (x+y)^i \cdot (a-b)$$

4.4. ¿Cuál de los siguientes identificadores son válidos?

n	85	Nombre
MiProblema	AAAAAAAAAA	
MiJuego	Nombre- Apellidos	
Mi Juego	Saldo-Actual	
write	92	
m&m	Universidad	
_m_m	Pontificia	
registro	Set 15	
A B	* 143	Edad

4.5. X es una variable entera e Y una variable carácter. Qué resultados producirá la sentencia `scanf("%d %d",&x,&y)` si la entrada es

- a) 5 c
- b) 5C

4.6. Escribir un programa que lea un entero, lo multiplique por 2 y a continuación lo escriba de nuevo en la pantalla.

4.7. Escribir las sentencias de asignación que permitan intercambiar los contenidos (valores) de dos variables.

4.8. Escribir un programa que lea dos enteros en las variables x e y, y a continuación obtenga los valores de : 1. `x / y`, 2. `x % y`. Ejecute el programa varias veces con diferentes pares de enteros como entrada.

4.9. Escribir un programa que solicite al usuario la longitud y anchura de una habitación y a continuación visualice su superficie con cuatro decimales.

4.10. Escribir un programa que convierta un número dado de segundos en el equivalente de minutos y segundos.

4.11. Escribir un programa que solicite dos números decimales y calcule su suma, visualizando la suma ajustada a la derecha. Por ejemplo, si los números son 57.45 y 425.55, el programa visualizará:

```
57.45
425.55
483.00
```

4.12. ¿Cuáles son los resultados visualizados por el siguiente programa, si los datos proporcionados son 5 y 8?

```
#include <stdio.h>
const int M = 6;
int main()
{
    int a, b, c;
    gets("Introduce el valor de a
        y de b");
    scanf("%d %d",&a,&b);
    c = 2 * a - b;
    c -= M;
    b = a + c - M;
    a = b * M;
    printf("\n a = %d\n",a);
    b = - 1;
    printf(" %6d %6d",b,c);
    return 0;
}
```

4.13. Escriba un programa para calcular la longitud de la circunferencia y el área del círculo para un radio introducido por el teclado.

4.14. Escribir un programa que visualice valores tales como:

```
7.1
7.12
7.123
7.1234
7.12345
7.123456
```

4.15. Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.

- 4.16. Escribir una sentencia lógica (boolean) que clasifique un entero x en una de las siguientes categorías.

$$x < 0 \quad \text{o bien} \quad 0 \leq x \leq 100 \\ \text{o bien} \quad x > 100$$

- 4.17. Escribir un programa que introduzca el número de un mes (1 a 12) y visualice el número de días de ese mes.

- 4.18. Escribir un programa que lea dos números y visualice el mayor, utilizar el operador ternario ? :.

- 4.19. El domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera, y se determina mediante el siguiente cálculo sencillo.

$$A = \text{año} \% 19$$

$$B = \text{año} \% 4 \\ C = \text{año} \% 7 \\ D = (19 * A + 24) \% 30 \\ E = (2 * B + 4 * C + 6 * D + 5) \% 7 \\ N = (22 + D + E)$$

donde N indica el número de día del mes de marzo (si N es igual o menor que 31) o abril (si es mayor que 31). Construir un programa que tenga como entrada un año y determine la fecha del domingo de Pascua.

Nota: utilizar el operador ternario ? : para seleccionar.

- 4.20. Determinar si el carácter asociado a un código introducido por teclado corresponde a un carácter alfabético, dígito, de puntuación, especial o no imprimible.

4.16. PROBLEMAS

- 4.1. Escribir un programa que lea dos enteros de tres dígitos y calcule e imprima su producto, cociente y el resto cuando el primero se divide por el segundo. La salida será justificada a derecha.

- 4.2. Una temperatura Celsius (centígrados) puede ser convertida a una temperatura equivalente F de acuerdo a la siguiente fórmula:

$$f = \left(\frac{9}{5}\right) c + 32$$

Escribir un programa que lea la temperatura en grados Celsius y la escriba en F.

- 4.3. Un sistema de ecuaciones lineales

$$ax + by = c \\ dx + ey = f$$

se puede resolver con las siguientes fórmulas:

$$x = \frac{ce - bf}{ae - bd} \quad y = \frac{af - cd}{ae - bd}$$

Diseñar un programa que lea dos conjuntos de coeficientes (a, b y c ; d, e y f) y visualice los valores de x e y .

- 4.4. Escribir un programa que dibuje el rectángulo siguiente:

```
* * * * *
*           *
*           *
*           *
* * * * *
```

- 4.5. Modificar el programa anterior, de modo que se lea una palabra de cinco letras y se imprima en el centro del rectángulo.

- 4.6. Escribir un programa C que lea dos números y visualice el mayor.

- 4.7. Escribir un programa para convertir una medida dada en pies a sus equivalentes en : a) yardas; b) pulgadas; c) centímetros, y d) metros (1 pie = 12 pulgadas, 1 yarda = 3 pies, 1 pulgada = 2,54 cm, 1 m = 100 cm). Leer el número de pies e imprimir el número de yardas, pies, pulgadas, centímetros y metros.

- 4.8. Teniendo como datos de entrada el radio y la altura de un cilindro queremos calcular: el área lateral y el volumen del cilindro.

- 4.9. Calcular el área de un triángulo mediante la fórmula:

$$\text{Área} = (p(p - a)(p - b)(p - c))^{1/2}$$

donde p es el semiperímetro, $p = (a + b + c)/2$, siendo a, b, c los tres lados del triángulo.

- 4.10. Escribimos un programa en el que se introducen como datos de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representan hectómetros, decámetros y metros respectivamente. Se ha de escribir, con un rótulo representativo, la longitud en decímetros.

- 4.11. Construir un programa que calcule y escriba el producto, cociente entero y resto de dos números de tres cifras.

- 4.12. Construir un programa para obtener la hipotenusa y los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

- 4.13. Escribir un programa que desglose cierta cantidad de segundos introducida por teclado en su equivalente en semanas, días, horas, minutos y segundos.

- 4.14. Escribir un programa que exprese cierta cantidad de pesetas en billetes y monedas de curso legal.

- 4.15. La fuerza de atracción entre dos masas, m_1 y m_2 separadas por una distancia d , está dada por la fórmula:

$$F = \frac{G * m_1 * m_2}{d^2}$$

donde G es la constante de gravitación universal

$$G = 6.673 \times 10^{-8} \text{ cm}^3/\text{g. seg}^2$$

Escribir un programa que lea la masa de dos cuerpos y la distancia entre ellos y a continuación obtenga la fuerza gravitacional entre ella. La salida debe ser en dinas; un dina es igual a $g\text{r. cm}/\text{seg}^2$.

- 4.16. La famosa ecuación de Einstein para conversión de una masa m en energía viene dada por la fórmula

$$E = cm^3 \quad c \text{ es la velocidad de la luz}$$

$$c = 2.997925 \times 10^{10} \text{ cm}/\text{sg}$$

Escribir un programa que lea una masa en gramos y obtenga la cantidad de energía producida cuando la masa se convierte en energía.

Nota: Si la masa se da en gramos, la fórmula produce le energía en ergios.

- 4.17. La relación entre los lados (a, b) de un triángulo y la hipotenusa (h) viene dada por la fórmula

$$a^2 + b^2 = h^2$$

Escribir un programa que lea la longitud de los lados y calcule la hipotenusa.

- 4.18.** Escribir un programa que lea la hora de un día de notación de 24 horas y la respuesta en notación de 12 horas. Por ejemplo, si la entrada es 13:45, la salida será

1 : 45 PM

El programa pedirá al usuario que introduzca exactamente cinco caracteres. Así, por ejemplo, las nueve en punto se introduce como

o9 :00

- 4.19.** Escribir un programa que lea el radio de un círculo y a continuación visualice: área del

círculo, diámetro del círculo y longitud de la circunferencia del círculo.

- 4.20.** Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo, 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiplos de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo será).

- 4.21.** Construir un programa que indique si un número introducido por teclado es positivo, igual a cero, o negativo, utilizar para hacer la selección el operador ? : .

CAPÍTULO 5

ESTRUCTURAS DE SELECCIÓN: SENTENCIAS `IF` Y `SWITCH`

TENII

- 5.1. Estructuras de control.
- 5.2. La sentencia `if`.
- 5.3. Sentencia `if` de dos alternativas:
`if-else`.
- 5.4. Sentencias `if-else` anidadas.
- 5.5. Sentencia de control `switch`.
- 5.6. Expresiones condicionales:
el operador `?:`.
- 5.7. Evaluación en cortocircuito
de expresiones lógicas.
- 5.8. Puesta a punto de programas.
- 5.9. Errores frecuentes
de programación.
- 5.10. *Resumen*.
- 5.10. *Ejercicios*.
- 5.12. *Problemas*.

INTRODUCCIÓN

Los programas definidos hasta este punto se ejecutan de **modo** secuencial, es decir, una sentencia después de otra. La ejecución comienza con la primera sentencia de la función y prosigue hasta la Última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se ejecutan, en qué momentos. Las *estructuras o construcciones de control* controlan la secuencia o flujo de ejecución de las sentencias. **Las** estructuras de control se dividen en tres grandes categorías en función del flujo de ejecución: *secuencia, selección y repetición*.

Este capítulo considera las *estructuras selectivas o condicionales* —sentencias **if** y **switch**— que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición.

CONCEPTOS CLAVE

- Estructura de control.
- Estructura de control selectiva.
- Sentencia **break**.
- Sentencia compuesta.
- Sentencia **enum**.
- Sentencia **if**.
- Sentencia **switch**.
- Tipo lógico en C.

5.1. ESTRUCTURAS DE CONTROL

Las **estructuras de control** controlan el flujo de ejecución de un programa o función. Las estructuras de control permiten combinar instrucciones o sentencias individuales en una simple unidad lógica con un punto de entrada y un punto de salida.

Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: **secuencia**, **selección (decisión)** y **repetición**. Hasta este momento sólo se ha utilizado el flujo secuencial. Una **sentencia compuesta** es un conjunto de sentencias encerradas entre llaves ({ y }) que se utiliza para especificar un flujo secuencial.

```
{
    sentencia ;
    sentencia ;

    sentencia ;
}
```

El control fluye de la *sentencia*₁ a la *sentencia*₂ y así sucesivamente. Sin embargo, existen problemas que requieren etapas con dos o más opciones o alternativas a elegir en función del valor de una condición o expresión.

5.2. LA SENTENCIA `if`

En C, la estructura de control de selección principal es una sentencia `if`. La sentencia `if` tiene dos alternativas o formatos posibles. El formato más sencillo tiene la sintaxis siguiente:

```
if (Expresión) Acción
```

Expresión lógica que determina si la acción se ha de ejecutar

Acción se ejecuta si la expresión lógica es verdadera

La sentencia `if` funciona de la siguiente manera. Cuando se alcanza la sentencia `if` dentro de un programa, se evalúa la **expresión** entre paréntesis que viene a continuación de `if`. Si **Expresión** es verdadera, se ejecuta **Acción**; en caso contrario no se ejecuta **Acción** (en su formato más simple, **Acción** es una sentencia simple y en los restantes formatos es una sentencia compuesta). En cualquier caso la ejecución del programa continúa con la siguiente sentencia del programa. La Figura 5.1 muestra un **diagrama de flujo** que indica el flujo de ejecución del programa.

Otro sistema de representar la sentencia `if` es:

```
if (condición) sentencia;
```

condición
sentencia

es una expresión entera(lógica).

es cualquier sentencia ejecutable, que se ejecutará sólo si la **condición** toma un valor distinto de cero.

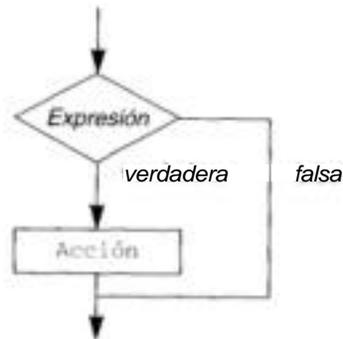


Figura 5.1. Diagrama de flujo de una sentencia básica `if`

Ejemplo 5.1

Prueba de divisibilidad

```

#include <stdio.h>
int main()
{
    int n, d;
    printf( "Introduzca dos enteros: " );
    scanf( "%d %d", &n,&d );
    if (n%d == 0) printf( " %d es divisible por %d\n",n,d );
    return 0;
}
  
```

Ejecución

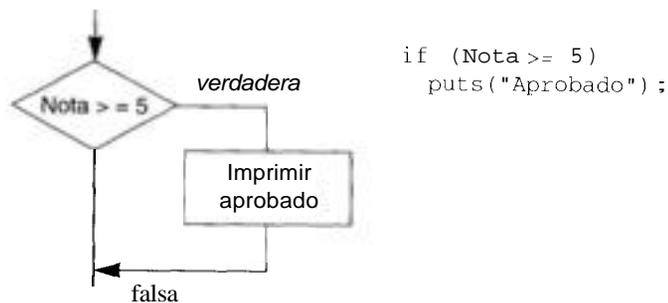
```

Introduzca dos enteros: 36 4
36 es divisible por 4
  
```

Este programa lee dos números enteros y comprueba cuál es el valor del resto de la división n entre d ($n\%d$). Si el resto es cero, n es divisible por d (en nuestro caso 36 es divisible por 4, ya que $36 : 4 = 9$ y el resto es 0).

Ejemplo 5.2

Representar la superación de un examen ($\text{Nota} \geq 5$, Aprobado).



```
#include <stdio.h>
void main()
{
    float numero;
    /* comparar número introducido por usuario */
    printf("Introduzca un número positivo o negativo: ");
    scanf("%f",&numero);

    /* comparar número con cero */
    if (numero > 0)
        printf("%f es mayor que cero\n" ,numero);
}
```

La ejecución de este programa produce

```
Introduzca un número positivo o negativo: 10.15
10.15 es mayor que cero
```

Si en lugar de introducir un número positivo se introduce un número negativo ¿Qué sucede?: nada. El programa es tan simple que sólo puede comprobar si el número es mayor que cero.

Ejemplo 5.3

```
#include <stdio.h>
void main()
{
    float numero;

    /* comparar número introducido por usuario */
    printf("Introduzca un número positivo o negativo: ");
    scanf("%f",&numero);
    /* comparar número */
    if (numero > 0)
        printf("%f es mayor que cero\n",numero);
    if (numero < 0)
        printf("%f es menor que cero\n",numero);
    if (numero == 0)
        printf("%f es igual a cero\n",numero);
}
```

Este programa simplemente añade otra sentencia `if` que comprueba si el número introducido es menor que cero. Realmente, una tercera sentencia `if` se añade también y comprueba si el número es igual a cero.

Ejercicio 5.1

Visualizar la tarifa de la luz según el gasto de corriente eléctrica. Para un gasto menor de 1.000Kwxh la tarifa es 1.2, entre 1.000 y 1.850Kwxh es 1.0 y mayor de 1.850Kwxh 0.9.

```
#include <stdio.h>
#define TARIFA1 1.2
#define TARIFA2 1.0
#define TARIFA3 0.9
```

```

int main()
{
    float gasto, tasa;
    printf("\n Gasto de corriente: ");
    scanf("%f",&gasto);
    if (gasto < 1000.0)
        tasa = TARIFA1;
    if (gasto >=1000.0 && gasto <=1850.0)
        tasa = TARIFA2;
    if (gasto >1850.0)
        tasa = TARIFA3;

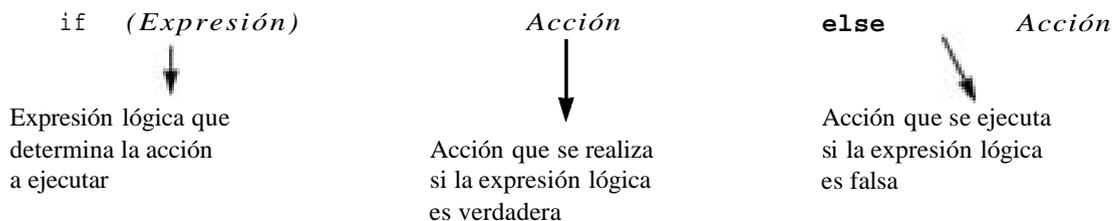
    printf("\nTasa que le corresponde a %.1f KwXh es de %f\n",
           gasto,tasa);
    return 0;
}

```

En el ejercicio se decide entre tres rangos la tasa que le corresponde. Se ha resuelto con tres selecciones simples.

5.3. SENTENCIA `if` DE DOS ALTERNATIVAS: `if-else`

Un segundo formato de la sentencia `if` es la sentencia `if-else`. Este formato de la sentencia `if` tiene la siguiente sintaxis:



En este formato *Acción* y *Acción* son individualmente, o bien una Única sentencia que termina en un punto y coma (;) o un grupo de sentencias encerrado entre llaves. Cuando se ejecuta la sentencia `if-else`, se evalúa *Expresión*. Si *Expresión* es verdadera, se ejecuta *Acción* y en caso contrario se ejecuta *Acción*. La Figura 5.2 muestra la semántica de la sentencia `if-else`.

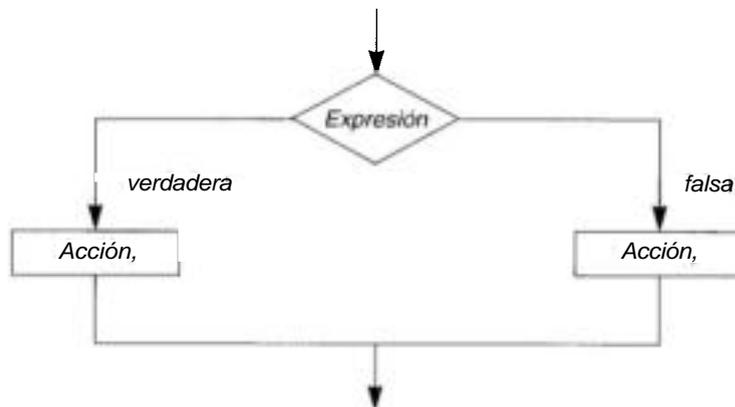


Figura 5.2. Diagrama de flujo de la representación de una sentencia `if-else`.

Ejemplos

```
1. if (salario > 100000)
    salario-neto = salario - impuestos;
    else
    salario-neto = salario;
```

Si *salario* es mayor que 100.000 se calcula el salario neto, restándole los impuestos; en caso contrario (**else**) el salario neto es igual al salario (**bruto**).

```
2. if (Nota >= 5)
    puts ("Aprobado");
    else
    puts ("suspense");
```

Si *Nota* es mayor o igual que 5 se escribe *Aprobado*; en caso contrario, *Nota* menor que 5, se escribe *Suspense*.

Formatos

```
1. if (expresión_lógica)
    sentencia
```

```
2. if (expresión_lógica)
    sentencia,
    else
    sentencia,
```

```
3. if (expresión_lógica) sentencia
```

```
4. if (expresión_lógica) sentencia else sentencia
```

Si *expresión lógica* es verdadera se ejecuta *sentencia* o bien *sentencia*,, si es falsa (*sino*, en caso contrario) se ejecuta *sentencia*.

Ejemplos

```
1. if (x > 0.0)
    producto = producto * x;
2. if (x != 0.0)
    producto = producto * x;
```

```
/* Se ejecuta la sentencia de asignación cuando x no es igual a 0.
   en este caso producto se multiplica por x y el nuevo valor se
   guarda en producto reemplazando el valor antiguo.
   Si x es igual a 0, la multiplicación no se ejecuta.
*/
```

Ejemplo 5.4

Prueba de divisibilidad (igual que el Ejemplo 5.1, al que se ha añadido la cláusula else)

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

```

{
    int n, d;
    printf( "Introduzca dos enteros: " );
    scanf("%d %d",&n,&d);
    if (n%d ==0)
        printf("%d es divisible por %d\n",n,d);
    else
        printf("%d no es divisible por %d\n",n,d);
    return 0;
}

```

Ejecución

```

Introduzca dos enteros 36 5
36 no es divisible por 5

```

Comentario

36 no es divisible por 5 ya que 36 dividido entre 5 produce un resto de 1 ($n\%d \neq 0$, es falsa, y se ejecuta la cláusula `else`).

Ejemplo 5.5

Calcular el mayor de dos números leídos del teclado y visualizarlo en pantalla.

```

#include <stdio.h>
int main()
{
    int x, y;
    printf( "Introduzca dos enteros: " );
    scanf("%d %d",&x,&y);
    if (x > y)
        printf("%6d\n",x);
    else
        printf("%6d\n",y);
    return 0;
}

```

Ejecución

```

Introduzca dos enteros: 17 54

```

Comentario

La condición es $(x > y)$. Si x es mayor que y , la condición es «verdadera» (*true*) y se evalúa a 1; en caso contrario la condición es «falsa» (*false*) y se evalúa a 0. De este modo se imprime x (en un campo de ancho 6, `%6d`) cuando es mayor que y , como en el ejemplo de la ejecución.

Ejemplo 5.6

Dada la función $f(x)$, calcular la función para un valor dado de x y visualizarlo en pantalla

$$f(x) = \begin{cases} x^2 - x & \text{para } x \leq 0.0 \\ -x^2 + 3x & \text{para } x > 0 \end{cases}$$

```
#include <stdio.h>
#include <math.h>
int main()
{
    float f,x;
    printf("\n Elige un valor de x: ");
    scanf("%f",&x);
    /* selección del rango en que se encuentra x */
    if (x <=0.0)
        f = pow(x,2) - x;
    else
        f = -pow(x,2) + 3*x;
    printf("f(%.1f) = %.3f",x,f);
    return 0;
}
```

Ejecución

Elige un valor de x: **-1.5**
 $f(-1.5) = 3.750$

Comentario

Una vez introducido x , se evalúa la condición $x \leq 0.0$, si es cierta asigna a f , $x^2 - x$; en caso contrario asigna a f , $-x^2 + 3x$.

5.4. SENTENCIAS if-else ANIDADAS

Hasta este punto, las sentencias `if` implementan decisiones que implican una o dos alternativas. En esta sección, se mostrará como se puede utilizar la sentencia `if` para implementar decisiones que impliquen diferentes alternativas.

Una sentencia `if` es anidada cuando la sentencia de la rama verdadera o la rama falsa, es a su vez una sentencia `if`. Una sentencia `if` anidada se puede utilizar para implementar decisiones con varias alternativas o multi-alternativas.

Sintaxis :

```
if (condición)
    sentencia
else if (condición)
    sentencia

else if (condición)
    sentencia
else
    sentencia,
```

Ejemplo 5.7

```
/* incrementar contadores de números positivos, números negativos o
ceros */
```

```
if (x > 0)
    num-pos = num-pos + 1;
else
    if (x < 0)
        num_neg = num_neg + 1;
    num-ceros = num-ceros + 1;
```

La sentencia `if` anidada tiene tres alternativas. Se incrementa una de las tres variables (`num-pos`, `num-neg` y `num-ceros`) en 1, dependiendo de que `x` sea mayor que cero, menor que cero o igual a cero, respectivamente. Las cajas muestran la estructura lógica de la sentencia `if` anidada; la segunda sentencia `if` es la acción o tarea falsa (a continuación de `else`) de la primera sentencia `if`.

La ejecución de la sentencia `if` anidada se realiza como sigue: se comprueba la primera condición (`x > 0`); si es verdadera, `num_pos` se incrementa en 1 y se salta el resto de la sentencia `if`. Si la primera condición es falsa, se comprueba la segunda condición (`x < 0`); si es verdadera `num-neg` se incrementa en uno; en caso contrario se incrementa `num-ceros` en uno. Es importante considerar que la segunda condición se comprueba *sólo si* la primera condición es *falsa*.

5.4.1. Sangría en las sentencias `if` anidadas

El formato multibifurcación se compone de una serie de sentencias `if` anidadas, que se pueden escribir en cada línea una sentencia `if`. La sintaxis multibifurcación anidada es:

Formato 1:

```
if (expresión-lógica)
    sentencia
else
    if (expresión-lógica)
        else
            if (expresión-lógica)
                sentencia
            else
                if (expresión-lógica)
                    sentencia
                else
                    sentencia
```

Formato 2:

```
if (expresión-lógica)
    sentencia
else if (expresión-lógica)
    sentencia
else if (expresión-lógica)
    sentencia
else if (expresión-lógica)
    sentencia
else
    sentencia
```

Ejemplos

```
1. if (x > 0)
    z = 2*log(x);
    else
        if (y > 0)
```

```

        z = sqrt(x) + sqrt(y);
2. if (x > 0)
    z = 2*log(x);
    else if (y > 0)
        z = sqrt(x) + sqrt(y);
    else
        puts( "\n *** Imposible calcular z");

```

Ejemplo 5.8

El siguiente programa realiza selecciones múltiples con la sentencia compuestas *if-else*.

```

#include <stdio.h>
void main()

    float numero;
    printf( " introduzca un número positivo o negativo: ");
    scanf("%f",&numero);
    /* comparar número con cero */
    if (numero > 0)
    {
        printf("%.2f %s", numero, "es mayor que cero\n");
        puts( "pruebe de nuevo introduciendo un número negativo");
    }
    else if (numero < 0)
    {
        printf("%.2f %s", numero, "es menor que cero\n");
        puts( "pruebe de nuevo introduciendo un número negativo");
    }
    else
    { printf("%.2f %s", numero, "es igual a cero\n");
      puts( "           ¿por qué no introduce otro número? ");
    }
}

```

5.4.2. Comparación de sentencias *if* anidadas y secuencias de sentencias *if*

Los programadores tienen dos alternativas: 1) usar una secuencia de sentencias *if*; 2) una Única sentencia *if* anidada. Por ejemplo, la sentencia *if* del Ejemplo 5.7 se puede reescribir como la siguiente secuencia de sentencias *if*.

```

if (x > 0)
    num_pos = num_pos + 1;
if (x < 0)
    num_neg = num_neg + 1;
if ( x == 0)
    num_ceros = num_ceros + 1;

```

Aunque la secuencia anterior es lógicamente equivalente a la original, no es tan legible ni eficiente. Al contrario que la sentencia `if` anidada, la secuencia no muestra claramente cual es la sentencia a ejecutar para un valor determinado de `x`. Con respecto a la eficiencia, la sentencia `if` anidada se ejecuta más rápidamente cuando `x` es positivo ya que la primera condición (`x > 0`) es verdadera, lo que significa que la parte de la sentencia `if` a continuación del primer `else` se salta. En contraste, se comprueban siempre las tres condiciones en la secuencia de sentencias `if`. Si `x` es negativa, se comprueban dos condiciones en las sentencias `if` anidadas frente a las tres condiciones de las secuencias de sentencias `if`. Una estructura típica **`if-else`** anidada permitida es:

```

if (numero > 0)
{
    ...
}
else
{
    if ( ...)
    {
        ...
    }
    else
    {
        if ( ...)
        {
            ...
        }
    }
}
    ...
}

```

Ejercicio 5.9

Existen diferentes formas de escribir sentencias `if` anidadas.

- ```

if (a > 0) if (b > 0) ++a; else if (c > 0)
if (a < 5) ++b; else if (b < 5) ++c; else --a;
else if (c < 5) --b; else --c; else a = 0;

```
- ```

if (a > 0)                /* forma más legible */
if (b > 0) ++a;
else
    if (c > 0)
        if (a < 5) ++b;
        else
            if (b < 5) ++c;
            else --a;
    else
        if (c < 5) --b;
        else --c;
    else
        a = 0;

```
- ```

if (a > 0) /* forma más legible */

```

```

 if (b > 0) ++a;
 else if (c > 0)
 if (a < 5) ++b;
 else if (b < 5) ++c;
 else --a;
 else if (c < 5) --b;
 else --c;
else
 a = 0;

```

---

### Ejercicio 5.10

Calcular el mayor de tres números enteros.

```

#include <stdio.h>
int main()
{
 int a, b, c, mayor;
 printf("\nIntroduzca tres enteros:");
 scanf("%d %d %d",&a,&b,&c);
 if (a > b)
 if (a > c) mayor = a;
 else mayor = c;
 else
 if (b > c) mayor = b;
 else mayor = c;
 printf("El mayor es %d \n",mayor);
 return 0;
}

```

---

### Ejecución

```

Introduzca tres enteros: 77 54 85
El mayor es 85

```

### Análisis

Al ejecutar el primer `if`, la condición `(a > b)` es verdadera, entonces se ejecuta la segunda `if`. En el segundo `if` la condición `(a > c)` es falsa, en consecuencia el primer `else mayor = 85` y se termina la sentencia `if`, a continuación se ejecuta la última línea y se visualiza `El mayor es 85`.

## 5.5. SENTENCIA DE CONTROL `switch`

La sentencia `switch` es una sentencia C que se utiliza para seleccionar una de entre múltiples alternativas. La sentencia `switch` es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada *expresión de control o selector*. El valor de esta expresión puede ser de tipo `int` o `char`, pero no de tipo `float` ni `double`.

**Sintaxis**

```

switch (selector)
{
 case etiqueta, : sentencias,;
 case etiqueta, : sentencias,;

 .

 case etiqueta, : sentencias,;
 default: sentencias,; /* opcional. */
}

```

La expresión de control o *selector* se evalúa y se compara con cada una de las etiquetas de `case`. La expresión *selector* debe ser un tipo ordinal (por ejemplo, `int`, `char`, pero no `float` o `string`). Cada *etiqueta* es un valor Único, constante y cada etiqueta debe tener un valor diferente de los otros. Si el valor de la expresión *selector* es igual a una de las etiquetas `case` —por ejemplo, *etiqueta*— entonces la ejecución comenzará con la primera sentencia de la secuencia *sentencia* y continuará hasta que se encuentra el final de la sentencia de control `switch`, o hasta encontrar la sentencia `break`. Es habitual que después de cada bloque de sentencias correspondiente a una secuencia se desee terminar la ejecución del `switch`; para ello se sitúa la sentencia `break` como Última sentencia del bloque. `break` hace que siga la ejecución en la siguiente sentencia `switch`.

**Sintaxis con `break`**

```

switch (selector)
{
 case etiqueta, : sentencias,;
 break;
 case etiqueta, : sentencias,;
 break;

 .

 case etiqueta, : sentencias,;
 break;
 default: sentencias,; /* opcional */
}

```

El tipo de cada etiqueta debe ser el mismo que la expresión de *selector*. Las expresiones están permitidas como etiquetas pero sólo si cada operando de la expresión es por sí misma una constante —por ejemplo, `4 + 8` o bien `m * 15`—, siempre que `m` hubiera sido definido anteriormente como constante con nombre.

Si el valor del selector no está listado en ninguna etiqueta `case`, no se ejecutará ninguna de las opciones a menos que se especifique una acción por defecto (omisión). La omisión de una etiqueta `default` puede crear un error lógico difícil de prever. Aunque la etiqueta `default` es opcional, se recomienda su uso a menos que se esté absolutamente seguro de que todos los valores de *selector* estén incluidos en las etiquetas `case`.

Una sentencia `break` consta de la palabra reservada `break` seguida por un punto y coma. Cuando la computadora ejecuta las sentencias siguientes a una etiqueta `case`, continúa hasta que se alcanza una sentencia `break`. Si la computadora encuentra una sentencia `break`, termina la sentencia `switch`. Si se omiten las sentencias `break`, después de ejecutar el código de `case`, la computadora ejecutará el código que sigue a la siguiente `case`.

---

### Ejemplo 5.11

```
switch (opcion)
{
 case 0:
 puts ("Cero!");
 break;
 case 1:
 puts("Uno!");
 break;
 case 2:
 puts("Dos!");
 break;
 default:
 puts ("Fuera de rango");
}
```

---

### Ejemplo 5.12

```
switch (opcion)
{
 case 0:
 case 1:
 case 2:
 puts ("Menor de 3");
 break;
 case 3:
 puts("Igual a 3");
 break;
 default:
 puts ("Mayor que 3");
}
```

---

### Ejemplo 5.13

*Comparación de las sentencias if-else-if y switch.* Se necesita saber si un determinado carácter `car` es una vocal. *Solución con if-else-if.*

```
if ((car == 'a') || (car == 'A'))
 printf("%c es una vocal\n", car);
else if ((car == 'e') || (car == 'E'))
 printf("%c es una vocal\n", car);
else if ((car == 'i') || (car == 'I'))
 printf("%c es una vocal\n", car);
```

```

else if ((car == 'o') || (car == 'O'))
 printf("%c es una vocal\n", car);
else if ((car == 'u') || (car == 'U'))
 printf("%c es una vocal\n", car);
else
 printf("%c no es una vocal\n", car);

```

*Solución con `switch`.*

```

switch (car) {
 case 'a' case 'A':
 case 'e' case 'E':
 case 'i' case 'I':
 case 'ó' case 'O':
 case 'u' case 'U':
 printf("%c es una vocal\n", car);
 break;
 default
 printf("%c no es una vocal\n", car);
}

```

---

I

### Ejemplo 5.15

*Dada una nota de un examen mediante un código escribir el literal que le corresponde a la nota.*

```

/* Programa resuelto con la sentencia switch */
#include <stdio.h>

int main()
{
 char nota;
 printf("Introduzca calificación (A-F) y pulse Intro:");
 scanf("%c", ¬a);

 switch (nota)
 {
 case 'A': puts("Excelente. Examen superado");
 break;
 case 'B': puts("Notable. Suficiencia");
 break;
 case 'C': puts("Aprobado");
 break;
 case 'D':
 case 'F': puts("Suspendido");
 break;
 default:
 puts ("No es posible esta nota");
 }
 puts("Final de programa");
 return 0;
}

```

---

I

Cuando se ejecuta la sentencia `switch`, se evalúa `nota`; si el valor de la expresión es igual al valor de una etiqueta, entonces se transfiere el flujo de control a las sentencias asociadas con la etiqueta

correspondiente. Si ninguna etiqueta coincide con el valor de nota se ejecuta la sentencia `default` y las sentencias que vienen detrás de ella. Normalmente la última sentencia de las sentencias que vienen después de una `case` es una sentencia `break`. Esta sentencia hace que el flujo de control del programa salte a la siguiente sentencia de `switch`. Si no existiera `break`, se ejecutarían también las sentencias restantes de la sentencia `switch`.

#### Ejecución de prueba 1

Introduzca calificación (A-F) y pulse Intro: **A**  
Excelente. Examen superado  
Final de programa

#### Ejecución de prueba 2

introduzca calificación (A-F) y pulse Intro: **B**  
Notable. Suficiencia  
Final de programa

#### Ejecución de prueba 3

Introduzca calificación (A-F) y pulse Intro: **E**  
No es posible esta nota  
Final de programa

### Precaución

Si se olvida `break` en una sentencia `switch`, el compilador no **emitirá** un mensaje de error ya que se habrá escrito una sentencia `switch` correcta sintácticamente pero no realizará las tareas previstas.

---

### Ejemplo 5.15

*Seleccionar un tipo de vehículo según un valor numérico.*

```
int tipo-vehiculo;
printf("Introduzca tipo de vehiculo:");
scanf("%d",&tipo_vehículo);
switch(tipo_vehículo)
{
 case 1:
 printf("turismo\n");
 peaje = 500;
 break;
 case 2:
 printf("autobus\n");
 peaje = 3000;
 break;
 case 3:
 printf("motocicleta\n");
 peaje = 300;
 break;
 default:
 printf("vehículo no autorizado\n");
}
}
```

Si se omite esta `break` el vehículo primero será turismo y luego autobús

---

Cuando la computadora comienza a ejecutar un `case` no termina la ejecución del `switch` hasta que se encuentra, o bien una sentencia `break`, o bien la Última sentencia del `switch`.

### 5.5.1. Caso particular de `case`

Está permitido tener varias expresiones `case` en una alternativa dada dentro de la sentencia `switch`. Por ejemplo, se puede escribir:

```
switch(c) {
 case '0':case '1': case '2': case '3': case '4':
 case '5':case '6': case '7': case '8': case '9':
 num_digitos++; /*se incrementa en 1 el valor de num_digitos */
 break;
 case ' ': case '\t': case '\n':
 num_blanco++; /*se incrementa en 1 el valor de num_blanco*/
 break;
 default:
 num_distintos++;
}
```

---

### 5.5.2. Uso de sentencias `switch` en menús

La sentencia `if-else` es más versátil que la sentencia `switch` y se puede utilizar unas sentencias `if-else` anidadas o multidecisión, en cualquier parte que se utiliza una sentencia `case`. Sin embargo, normalmente, la sentencia `switch` es más clara. Por ejemplo, la sentencia `switch` es idónea para implementar menús.

Un *menú* de un restaurante presenta una lista de alternativas para que un cliente elija entre sus diferentes opciones. Un menú en un programa de computadora hace la misma función: presentar una lista de alternativas en la pantalla para que el usuario elija una de ellas.

En los capítulos siguientes se volverá a tratar el tema de los menús en programación con ejemplos prácticos.

## 5.6. EXPRESIONES CONDICIONALES: EL OPERADOR `?` :

Las sentencias de selección (`if` y `switch`) consideradas hasta ahora, son similares a las sentencias previstas en otros lenguajes, tales como Pascal y Fortran 90. C tiene un tercer mecanismo de selección, una expresión que produce uno de dos valores, resultado de una expresión lógica o booleana (también denominada condición). Este mecanismo se denomina *expresión condicional*. Una expresión condicional tiene el formato `C ? A : B` y es realmente una operación ternaria (tres operandos) en el que `C`, `A` y `B` son los tres operandos y `?` : es el operador.

### *Sintaxis*

*condición* ? *expresión*, : *expresión*,

*condición*

*expresión /expresión*

es una expresión lógica

son expresiones compatibles de tipos

Se evalúa *condición*, si el valor de *condición* es verdadera (distinto de cero) entonces se devuelve como resultado el valor de *expresión*; si el valor de *condición* es falsa (cero) se devuelve como resultado el valor de *expresión*.

Uno de los medios más sencillos del operador condicional (*? :*) es utilizar el operador condicional y llamar a una de dos funciones.

## Ejemplos

1. Selecciona con el operador *? :* la ejecución de una función u otra.

```
a == b ? funcion1() : funcion2();
```

es equivalente a la siguiente sentencia:

```
if (a == b)
 funcion1();
else
 funcion2();
```

2. El operador *? :* se utiliza en el siguiente segmento de código para asignar el menor de dos valores de entrada a *menor*.

```
int entrada1, entrada2;
int menor;
scanf("%d %d",&entrada1,&entrada2);
menor = entrada1 <= entrada2 ? entrada1 : entrada2;
```

## Ejemplo 5.16

*Seleccionar el mayor de dos números enteros con la sentencia if-else y con el operador ? :*

```
#include <stdio.h>

void main()
{
 float n1, n2;

 printf("Introduzca dos números positivos o negativos:");
 scanf("%d %d",&n1,&n2);
 /* selección con if-else */
 if (n1 > n2)
 printf("%d > %d",n1,n2);
 else
 printf("%d <= %d",n1,n2);

 /* operador condicional */
 n1 > n2 ? printf("%d > %d",n1,n2) : printf("%d <= %d",n1,n2);
}
```

## 5.7. EVALUACIÓN EN CORTOCIRCUITO DE EXPRESIONES LÓGICAS

Cuando se evalúan expresiones lógicas en C se puede emplear una técnica denominada *evaluación en cortocircuito*. Este tipo de evaluación significa que se puede detener la evaluación de una expresión lógica tan pronto como su valor pueda ser determinado con absoluta certeza. Por ejemplo, si el valor de `(soltero == 's')` es falso, la expresión lógica `(soltero == 's') && (sexo == 'h') && (edad > 18) && (edad <= 45)` será falsa con independencia de cual sea el valor de las otras condiciones. La razón es que una expresión lógica del tipo

```
falso && (...)
```

debe ser siempre falsa, cuando uno de los operandos de la operación AND es falso. En consecuencia no hay necesidad de continuar la evaluación de las otras condiciones cuando `(soltero == 's')` se evalúa a falso.

El compilador C utiliza este tipo de evaluación. Es decir, la evaluación de una expresión lógica de la forma, `a && b` se detiene si la subexpresión `a` de la izquierda se evalúa a falsa.

C realiza evaluación en cortocircuito con los operadores `&&` y `||`, de modo que evalúa primero la expresión más a la izquierda de las dos expresiones unidas por `&&` o bien por `||`. Si de esta evaluación se deduce la información suficiente para determinar el valor final de la expresión (independiente del valor de la segunda expresión), el compilador de C no evalúa la segunda expresión.

### Ejemplo 5.17

Si `x` es negativo, la expresión

```
(x >= 0) && (y > 1)
```

se evalúa en cortocircuito ya que `x >= 0` será falso y, por tanto, el valor final de la expresión será falso.

En el caso del operador `||` se produce una situación similar. Si la primera de las dos expresiones unidas por el operador `||` es *verdadera*, entonces la expresión completa es *verdadera*, con independencia de que el valor de la segunda expresión sea *verdadero* o *falso*. La razón es que el operador OR produce resultado verdadero si el primer operando es verdadero.

Otros lenguajes, distintos de C, utilizan **evaluación completa**. En evaluación completa, cuando dos expresiones se unen por un símbolo `&&` o `||`, se evalúan siempre ambas expresiones y a continuación se utilizan las tablas de verdad de `&&` o bien `||` para obtener el valor de la expresión final.

### Ejemplo 5.18

Si `x` es cero, la condición

```
if ((x != 0.0) && (y/x > 7.5))
```

es falsa ya que `(x != 0.0)` es falsa. Por consiguiente, no hay necesidad de evaluar la expresión `(y/x > 7.5)` cuando `x` sea cero, de utilizar evaluación completa se produciría un error en tiempo de ejecución. Sin embargo, si altera el orden de las expresiones, al evaluar el compilador la sentencia `if`

```
if ((y/x > 7.5) && (x != 0.0))
```

se produciría un error en tiempo de ejecución de división por cero (`"division by zero"`).

**El orden de las expresiones con operadores `&&` y `||` puede ser crítico en determinadas situaciones.**

## 5.8. PUESTA A PUNTO DE PROGRAMAS

### Estilo y diseño

1. El estilo de escritura de una sentencia `if` e `if-else` es el sangrado de las diferentes líneas en el formato siguiente:

```

if (expresión-lógica)
 sentencia
else
 sentencia

if (expresión-lógica)
{
 sentencia

 sentencia
}
else
{
 sentencia

 sentencia
}

```

En el caso de sentencias `if-else-if` utilizadas para implementar una estructura de selección multialternativa se suele escribir de la siguiente forma:

```

if (expresión-lógica)
 sentencia
else if (expresión_lógica)
 sentencia

```

```

else if (expresión-lógica)
 sentencia
else
 sentencia

```

2. Una construcción de selección múltiple se puede implementar más eficientemente con una estructura `if-else-if` que con una secuencia de sentencias independientes `if`. Por ejemplo:

```

printf('introduzca nota");
scanf("%d",¬a);
if (nota < 0 | nota > 100)
{
 printf(" %d no es una nota válida.\n",nota);
 return'?';
}
if ((nota >= 90) && (nota <= 100))
 return 'A';
if ((nota >= 80) && (nota < 90))
 return 'B';
if ((nota >= 70) && (nota < 80))

```

```

 return 'C';
if ((nota >= 60) && (nota < 70))
 return 'D';
if (nota < 60)
 return 'F';

```

Con independencia del valor de `nota` se ejecutan todas las sentencias `if`; 5 de las expresiones lógicas son expresiones compuestas, de modo que se ejecutan 16 operaciones con independencia de la `nota` introducida. En contraste, las sentencias `if` anidadas reducen considerablemente el número de operaciones a realizar (3 a 7), todas las expresiones son simples y no se evalúan todas ellas siempre.

```

printf ("Introduzca nota");
scanf ("%d", ¬a);
if (nota < 0 || nota > 100)
{
 printf ("%d no es una nota válida.\n", nota);
 return '?';
}
else if (nota >= 90)
 return 'A';
else if (nota >= 80)
 return 'B';
else if (nota >= 70)
 return 'C';
else if (nota >= 60)
 return 'D';
else
 return 'F';

```

## 5.9. ERRORES FRECUENTES DE PROGRAMACIÓN

1. Uno de los errores más comunes en una sentencia `if` es utilizar un operador de asignación (`=`) en lugar de un operador de igualdad (`==`).
2. En una sentencia `if` anidada, cada cláusula `else` se corresponde con la `if` precedente más cercana. Por ejemplo, en el segmento de programa siguiente

```

if (a > 0)
if (b > 0)
c = a + b;
else
c = a + abs(b);
d = a * b * c;

```

¿Cuál es la sentencia `if` asociada a `else`?

El sistema más fácil para evitar errores es el sangrado o indentación, con lo que ya se aprecia que la cláusula `else` se corresponde a la sentencia que contiene la condición `b > 0`

```

if (a > 0)
 if (b > 0)
 c = a + b;
 else
 c = a + abs(b);
d = a * b * c;

```

3. Las comparaciones con operadores `==` de cantidades algebraicamente iguales pueden producir una expresión lógica falsa, debido a que la mayoría de los números reales no se almacenan exactamente. Por ejemplo, aunque las expresiones reales siguientes son equivalentes:

```
a * (1/a)
1.0
```

son algebraicamente iguales, la expresión

```
a * (1/a) == 1.0
```

puede ser falsa debido a que `a` es real.

4. Cuando en una sentencia `switch` o en un bloque de sentencias falta una de las llaves `{, }` aparece un mensaje de error tal como:

```
Error ...: Compound statement missing } in function
```

Si no se tiene cuidado con la presentación de la escritura del código, puede ser muy difícil localizar la llave que falta.

5. El selector de una sentencia `switch` debe ser de tipo entero o compatible entero. Así las constantes reales

```
2.4, -4.5, 3.1416
```

no pueden ser utilizadas en el selector.

6. Cuando se utiliza una sentencia `switch`, asegúrese que el selector de `switch` y las etiquetas `case` son del mismo tipo (`int`, `char` pero **no** `float`). Si el selector se evalúa a un valor no listado en ninguna de las etiquetas `case`, la sentencia `switch` no gestionará ninguna acción; por esta causa se suele poner una etiqueta `default` para resolver este problema.

## 5.10. RESUMEN

### Sentencia `if`

#### Una alternativa

```
if (a != 0)
 resultado = a/b;
```

#### Dos alternativas

```
if (a >= 0)
 f = 5*cos(a*pi/180.);
else
 f = -2*sin(a*pi/180.) + 0.5;
```

### Múltiples alternativas

```
if (x < 0)
{
 puts("Negativo");
 abs_x = -x;
}
else if (x == 0)
{
 puts("Cero");
 abs_x = 0;
}
else
{
 puts("positivo");
 abs_x = x;
}
```

**Sentencia** `switch`

```

case 'A' : case 'a' :
 puts("Sobresaliente");
 break;
case 'B' : case 'b' :
 puts("Notable");
 break;
case 'C' : case 'c' :
 puts("Aprobado");
 break;
case 'D' : case 'd' :
 puts("Suspenso");
 break;
default:
 puts("nota no válida");
}

```

**5.11. EJERCICIOS**

5.1. ¿Qué errores de sintaxis tiene la siguiente sentencia?

```

if x > 25.0
 y = x
else
 y = z;

```

5.2. ¿Qué valor se asigna a `consumo` en la sentencia `if` siguiente si velocidad es 120?

```

if (velocidad > 80)
 consumo = 10.00;
else if (velocidad > 100)
 consumo = 12.00;
else if (velocidad > 120)
 consumo = 15.00;

```

5.3. Explique las diferencias entre las sentencias de la columna de la izquierda y de la columna de la derecha. Para cada una de ellas deducir el valor final de `x` si el valor inicial de `x` es 0.

|                                   |                             |
|-----------------------------------|-----------------------------|
| <code>if (x &gt;= 0)</code>       | <code>if (x &gt;= 0)</code> |
| <code>  x++;</code>               | <code>  x++;</code>         |
| <code>else if (x &gt;= 1);</code> | <code>if (x &gt;= 1)</code> |
| <code>  x+= 2;</code>             | <code>  x+= 2;</code>       |

5.4. ¿Qué salida producirá el código siguiente, cuando se empotra en un programa completo y primera-opcion vale 1? ¿Y si primera-opcion vale 2?

```

int primera-opcion;

switch (primera-opcion + 1)
{
 case 1:
 puts("Cordero asado");
 break;
 case 2:
 puts("Chuleta lechal");
 break;
 case 3:
 puts("Chuletón");
 case 4:
 puts("Postre de Pastel");
 break;
 default:
 puts("Buen apetito");
}

```

5.5. ¿Qué salida producirá el siguiente código, cuando se empotra en un programa completo?

```
int x = 2;
puts ("Arranque");
if (x <= 3)
 if (x != 0)
 puts ("Hola desde el segundo
 if");
 else
 puts ("Hola desde el else.");
puts ("Fin\nArranque de nuevo");
if (x > 3)
 if (x != 0)
 puts ("Hola desde el segundo
 if.");
 else
 puts ("Hola desde el else.");
puts ("De nuevo fin");
```

5.6. Escribir una sentencia `if-else` que visualice la palabra `Alta` si el valor de la variable `nota` es mayor que `100` y `Baja` si el valor de esa `nota` es menor que `100`.

5.7. ¿Qué hay de incorrecto en el siguiente código?

```
if (x = 0) printf ("%d = 0\n", x);
else printf ("%d != 0\n", x);
```

5.8. ¿Cuál es el error del siguiente código?

```
if (x < y < z) printf ("%d < %d <
 %d\n", x, y, z);
```

5.9. ¿Cuál es el error de este código?

```
printf ("Introduzca n:");
scanf ("%d", &n);
if (n < 0)
 puts ("Este número es negati-
 vo. Pruebe de nuevo.");
 scanf ("%d", &n);
else
 printf ("conforme. n=%d\n", n);
```

5.10. Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.

5.11. Escribir una sentencia `if-else` que clasifique un entero `x` en una de las siguientes categorías y escriba un mensaje adecuado:

|         |               |                     |
|---------|---------------|---------------------|
| $x < 0$ | <i>o bien</i> | $0 \leq x \leq 100$ |
|         | <i>o bien</i> | $x > 100$           |

5.12. Escribir un programa que introduzca el número de un mes (1 a 12) y visualice el número de días de ese mes.

5.13. Se trata de escribir un programa que clasifique enteros leídos del teclado de acuerdo a los siguientes puntos:

- si el entero es 30 o mayor, o negativo, visualizar un mensaje en ese sentido;
- en caso contrario, si es un nuevo primo, potencia de 2, o un número compuesto, visualizar el mensaje correspondiente;
- si son cero o 1, visualizar 'cero' o 'unidad'.

5.14. Escribir un programa que determine el mayor de tres números.

5.15. El domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera, y se determina mediante el siguiente cálculo sencillo:

$$\begin{aligned}
 A &= \text{año mod } 19 \\
 B &= \text{año mod } 4 \\
 C &= \text{año mod } 7 \\
 D &= (19 * A + 24) \text{ mod } 30 \\
 E &= (2 * B + 4 * C + 6 * D + 5) \\
 &\quad \text{mod } 7 \\
 N &= (22 + D + E)
 \end{aligned}$$

Donde `N` indica el número de día del mes de marzo (si `N` es igual o menor que 3) o abril (si es mayor que 31). Construir un programa que determine fechas de domingos de Pascua.

5.16. Codificar un programa que escriba la calificación correspondiente a una nota, de acuerdo con el siguiente criterio:

|     |           |                     |
|-----|-----------|---------------------|
| 0   | $a < 5.0$ | Suspense            |
| 5   | $a < 6.5$ | Aprobado            |
| 6.5 | $a < 8.5$ | Notable             |
| 8.5 | $a < 10$  | Sobresaliente       |
| 10  |           | Matrícula de honor. |

5.17. Determinar si el carácter asociado a un código introducido por teclado corresponde a un carácter alfabético, dígito, de puntuación, especial o no imprimible.

## 5.12. PROBLEMAS

- 5.1. Cuatro enteros entre 0 y 100 representan las puntuaciones de un estudiante de un curso de informática. Escribir un programa para encontrar la media de estas puntuaciones y visualizar una tabla de notas de acuerdo al siguiente cuadro:

| Media  | Puntuación |
|--------|------------|
| 90-100 | A          |
| 80-89  | B          |
| 70-79  | C          |
| 60-69  | D          |
| 0-59   | E          |

- 5.2. Escribir un programa que lea la hora de un día de notación de 24 horas y la respuesta en notación de 12 horas. Por ejemplo, si la entrada es 13:45, la salida será

1:45 PM

El programa pedirá al usuario que introduzca exactamente cinco caracteres. Así, por ejemplo, las nueve en punto se introduce como

09:00

- 5.3. Escribir un programa que acepte fechas escritas de modo usual y las visualice como tres números. Por ejemplo, la entrada

15, Febrero 1989

producirá la salida

15 2 1989

- 5.4. Escribir un programa que acepte un número de tres dígitos escrito en palabra y a continuación los visualice como un valor de tipo entero. La entrada se termina con un punto. por ejemplo, la entrada

doscientos veinticinco

producirá la salida

225

- 5.5. Escribir un programa que acepte un año escrito en cifras arábigas y visualice el año escrito en números romanos, dentro del rango 1000 a 2000.

**Nota:** Recuerde que V = 5 X = 10 L = 50  
C = 100 D = 500 M = 1000

|                  |              |
|------------------|--------------|
| IV = 4           | XL = 40      |
| MCM = 1900       | CM = 900     |
| MCMLX = 1960     | MCML = 1950  |
| MCMLXXXIX = 1989 | MCMXL = 1940 |

- 5.6. Se desea redondear un entero positivo N a la centena más próxima y visualizar la salida. Para ello la entrada de datos debe ser los cuatro dígitos A,B,C,D, del entero N. Por ejemplo, si A es 2, B es 3, C es 6 y D es 2, entonces N será 2362 y el resultado redondeado será 2400. Si N es 2342, el resultado será 2300, y si N = 2962, entonces el número será 3000. Diseñar el programa correspondiente.

- 5.7. Se quiere calcular la edad de un individuo, para ello se va a tener como entrada dos fechas en el formato día (1 a 31), mes (1 a 12) y año (entero de cuatro dígitos), correspondientes a la fecha de nacimiento y la fecha actual, respectivamente. Escribir un programa que calcule y visualice la edad del individuo. Si es la fecha de un bebé (menos de un año de edad), la edad se debe dar en meses y días; en caso contrario, la edad se calculará en años.

- 5.8. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo, 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiplos de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo será).

- 5.9. Escribir un programa que calcule el número de días de un mes, dados los valores numéricos del mes y el año.

- 5.10. Se desea calcular el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:

- Horas semanales trabajadas < 38 a una tasa dada.
- Horas extras (38 o más) a una tasa 50 por 100 superior a la ordinaria.
- Impuestos 0 por 100, si el salario bruto es menor o igual a 50.000 pesetas.
- Impuestos 10 por 100, si el salario bruto es mayor de 50.000 pesetas.

- 5.11. Determinar el menor número de billetes y monedas de curso legal equivalentes a cierta cantidad de pesetas (cambio óptimo).

- 5.12. Escribir y ejecutar un programa que simule un calculador simple. Lee dos enteros y un carácter. Si el carácter es un +, se imprime la suma; si es un -, se imprime la diferencia; si es un \*, se imprime el producto; si es un /, se imprime el cociente; y si es un % se imprime el resto. **Nota:** utilizar la sentencia `switch`.

## CAPÍTULO 6

# ESTRUCTURAS DE CONTROL: BUCLES

---

### CONTENIDO

- 6.1.** La sentencia *while*.
- 6.2.** Repetición: el bucle *for*.
- 6.3.** Precauciones en el uso de *for*.
- 6.4.** Repetición: el bucle *do-while*.
- 6.5.** Comparación de bucles *while*, *for* y *do-while*.
- 6.6.** Diseño de bucles.
- 6.7.** Bucles anidados.
- 6.8.** *Resumen*.
- 6.9.** *Ejercidos*.
- 6.10.** *Problemas*.
- 6.11.** *Proyectos de programación*.

---

## INTRODUCCIÓN

Una de las características de las computadoras que aumentan considerablemente su potencia es su capacidad para ejecutar **una** tarea muchas (*repetidas*) veces con gran velocidad, precisión y fiabilidad. Las tareas repetitivas es **algo** que los humanos encontramos difíciles y tediosas de realizar. En este capítulo se estudian las *estructuras de control iterativas o repetitivas* que realizan la repetición o iteración de acciones. C soporta tres tipos de estructuras de control: los bucles **while**, **for** y **do-while**. Estas estructuras de control o sentencias repetitivas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

---

## CONCEPTOS CLAVE

- Bucle.
- Comparación de **while**, **for** y **do**.
- Control de bucles.
- Iteración/repetición.
- Optimización de bucles.
- Sentencia **break**.
- Sentencia **do-while**.
- Sentencia **for**.
- Sentencia **while**.
- Terminación de un bucle.

## 6.1. LA SENTENCIA `while`

Un **bucle** (*ciclo*) es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un número de veces. La sentencia (o grupo de sentencias) que se repiten en un bloque se denomina **cuerpo** del bucle y cada repetición del cuerpo del bucle se llama **iteración** del bucle. Las dos principales cuestiones de diseño en la construcción del bucle son: ¿Cuál es el cuerpo del bucle? ¿Cuántas veces se iterará el cuerpo del bucle?

Un bucle `while` tiene una **condición** del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle `while` es un bucle *pretest* de modo que cuando se ejecuta el mismo, se evalúa la condición *antes* de que se ejecute el cuerpo del bucle. La Figura 6.1 representa el diagrama del bucle `while`.

El diagrama indica que la ejecución de la sentencia o sentencias expresadas se repite *mientras* la condición del bucle permanece verdadera y termina cuando se hace falsa. También indica el diagrama anterior que la condición del bucle se evalúa antes de que se ejecute el cuerpo del bucle y, por consiguiente, si esta condición es inicialmente falsa, el cuerpo del bucle no se ejecutará. En otras palabras, el cuerpo de un bucle `while` se ejecutará *cero o más veces*.

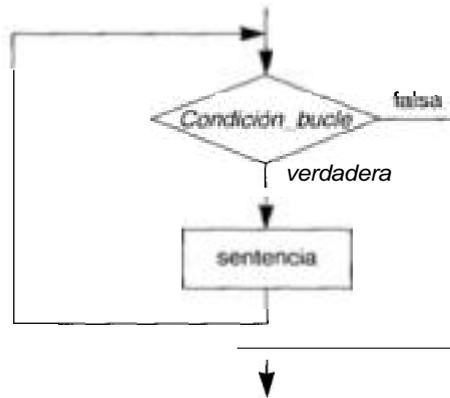


Figura 6.1

### Sintaxis

1 `while` (*condición-bucle*)  
    *Sentencia*;  $\longrightarrow$  *cuerpo*

2 `while` (*condición-bucle*)  
 {  
   *sentencia-1*;  
   *sentencia-2*;  
   :  
   :  
   *sentencia-n*;  
 }

} *cuerpo*

**while** es una palabra reservada C  
*condición\_bucle* es una expresión lógica o booleana  
*sentencia* es una sentencia simple o compuesta

El **comportamiento o funcionamiento** de una sentencia (bucle) `while` es:

1. Se evalúa la *condición-bucle*
2. **Si** *condición-bucle* es verdadera (distinto de cero):
  - a. La *sentencia* especificada, denominada el **cuerpo** del bucle, se ejecuta.
  - b. Vuelve el control al paso 1.
3. **En caso contrario:**  
 El control se transfiere a la sentencia siguiente al bucle o sentencia `while`.

Las sentencias del cuerpo del bucle se repiten **mientras** que la expresión lógica (condición del bucle) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa, se termina y se **sale** del bucle y se ejecuta la siguiente sentencia de programa después de la sentencia **while**.

```
/* cuenta hasta 10 */
int x = 0;
while (x < 10)
 printf("X: %d", x++);
```

---

## Ejemplo

```
/* visualizar n asteriscos */
contador = 0;
while (contador < n)
{
 printf(" * ");
 contador++;
} /* fin de while */
```

→ inicialización  
 → prueba/condición  
 → actualización (incrementa en 1 contador)

---

La variable que representa la condición del bucle se denomina también **variable de control del bucle** debido a que su valor determina si el cuerpo del bucle se repite. La variable de control del bucle debe ser: 1) inicializada, 2) comprobada, y 3) actualizada para que el cuerpo del bucle se ejecute adecuadamente. Cada etapa se resume así:

1. **Inicialización.** Contador se establece a un valor inicial (se inicializa a cero, aunque podría ser otro su valor) antes de **que** se alcance la sentencia `while`.
2. **Prueba/condición.** Se comprueba el valor de `contador` antes de que comience la repetición de cada bucle (denominada **iteración opasada**).
3. **Actualización.** Contador se actualiza (su valor se incrementa en 1, mediante el operador `++`) durante cada iteración.

Si la variable de control no se actualiza el bucle se ejecutará «siempre». Tal bucle se denomina **bucle infinito**. En otras palabras un bucle infinito (sin terminación) se producirá cuando la condición del bucle permanece y no se hace falsa en ninguna iteración.

```
/* bucle infinito */
contador = 1;
while (contador < 100)
```

```

{
 printf("%d \n", contador);
 contador--; \longrightarrow decrementa en 1 contador
}

```

contador se inicializa a 1 (menor de 100) y como contador-- decrementa en 1 el valor de contador en cada iteración, el valor del contador nunca llegará a valer 100, valor necesario de contador para que la condición del bucle sea falsa. Por consiguiente, la condición contador < 100 siempre será verdadera, resultando un bucle infinito, cuya salida será:

```

1
0
-1
-2
-3
-4

```

### Ejemplo

```

/* Bucle de muestra con while */
#include <stdio.h>
int main()
{
 int contador = 0; /* inicializa la condición */
 while(contador < 5) /* condición de prueba */
 {
 contador ++; /* cuerpo del bucle */
 printf ("contador: %d \n", contador);
 }
 printf ("Terminado.Contador: %d \n", contador);
 return 0;
}

```

### Ejecución

```

contador: 1
contador: 2
contador: 3
contador: 4
contador: 5
Terminado.Contador: 5

```

#### 6.1.1. Operadores de incremento y decremento (++ , --)

C ofrece los operadores de incremento (++) y decremento (--) que soporta una sintaxis abreviada para añadir (incrementar) o restar (decrementar) 1 al valor de una variable. Recordemos del Capítulo 4 la sintaxis de ambos operadores:

```

++nombreVariable /* preincremento */
nombreVariable++ /* postincremento */

--nombreVariable /* predecremento */
nombreVariable-- /* postdecremento */

```

---

### Ejemplo 6.1

Si *i* es una variable entera cuyo valor es 3, las variables *k* e *i* toman los valores sucesivos que se indican en las sentencias siguientes:

```

k = i++; /* asigna el valor 3 a k y 4 a i */
k = ++i; /* asigna el valor 5 a k y 5 a i */
k = i--; /* asigna el valor 5 a k y 4 a i */
k = --i; /* asigna el valor 3 a k y 3 a i */

```

---

### Ejemplo 6.2

Uso del operador de incremento ++ para controlar la iteración de un bucle (una de las aplicaciones más usuales de ++).

```

/* programa cálculo de calorías */
#include <stdio.h>

int main0
{
 int num_de_elementos, cuenta,
 calorías_por_alimento, calorías-total;

 printf("¿Cuántos alimentos ha comido hoy? ");
 scanf("%d ", &num_de_elementos);

 calorías-total = 0;
 cuenta = 1;
 printf("Introducir el número de calorías de cada uno de los ");
 printf("%d %s", num_de_elementos, "alimentos tomados:\n");

 while (cuenta++ <= num_de_elementos)

 scanf("%d", &calorías_por_alimento);
 calorías-total += calorías_por_alimento;
 }

 printf("Las calorías totales consumidas hoy son = ");
 printf("%d \n", calorías-total);
 return 0;
}

```

---

### Ejecución de muestra

```

¿Cuántos alimentos ha comido hoy? 8
Introducir el número de calorías de cada uno de los 8 alimentos tomados:
500 50 1400 700 10 5 250 100
Las calorías totales consumidas hoy son = 3015

```

### 6.1.2. Terminaciones anormales de un bucle (ciclo)

Un error típico en el diseño de una sentencia `while` se produce cuando el bucle sólo tiene una sentencia en lugar de varias sentencias como se planeó. El código siguiente

```
contador = 1;
while (contador < 25)
 printf("%d\n", contador);
 contador++;
```

visualizará infinitas veces el valor 1. Es decir, entra en un bucle infinito del que nunca sale porque no se actualiza (modifica) la variable de control `contador`.

La razón es que el punto y coma al final de la línea `printf("%d\n", contador);` hace que el bucle termine en ese punto y coma, aunque aparentemente el sangrado pueda dar la sensación de que el cuerpo de `while` contiene dos sentencias, `printf()` y `contador++`; El error se hubiera detectado rápidamente si el bucle se hubiera escrito correctamente

```
contador = 1;
while (contador < 25)
 printf("%d\n", contador);
 contador++;
```

La solución es muy sencilla, utilizar las llaves de la sentencia compuesta:

```
contador = 1;
while (contador < 25)
{
 printf("%d\n", contador);
 contador++;
}
```

### 6.1.3. Diseño eficiente de bucles

Una cosa es analizar la operación de un bucle y otra diseñar eficientemente sus propios bucles. Los principios a considerar son: primero, analizar los requisitos de un nuevo bucle con el objetivo de determinar su inicialización, prueba (condición) y actualización de la variable de control del bucle. El segundo es desarrollar *patrones estructurales* de los bucles que se utilizan frecuentemente.

### 6.1.4. Bucles `while` con cero iteraciones

El cuerpo de un bucle no se ejecuta nunca si la prueba o condición de repetición del bucle no se cumple, es falsa (es cero en C), cuando se alcanza `while` la primera vez.

```
contador = 10;
while (contador > 100)
{
 ...
}
```

El bucle anterior nunca se ejecutará ya que la condición del bucle (`contador > 100`) es falsa la primera vez que se ejecuta. El cuerpo del bucle nunca se ejecutará.

### 6.1.5. Bucles controlados por centinelas

Normalmente, no se conoce con exactitud cuantos elementos de datos se procesarán antes de comenzar su ejecución. Esto se produce bien porque hay muchos datos a contar normalmente o porque el número de datos a procesar depende de cómo prosigue el proceso de cálculo.

Un medio para manejar esta situación es instruir al usuario a introducir un único dato definido y especificado denominado *valor centinela* como Último dato. La condición del bucle comprueba cada dato y termina cuando se lee el valor centinela. El valor centinela se debe seleccionar con mucho cuidado y debe ser un valor que no pueda producirse como dato. En realidad el centinela es un valor que sirve para terminar el proceso del bucle.

En el siguiente fragmento de código hay un bucle con centinela; se introducen notas mientras que ésta sea distinta de centinela.

```
/*
 entrada de datos numéricos,
 centinela -1
*/
const int centinela = -1;
printf ("Introduzca primera nota:");
scanf ("%d", ¬a);
while (nota != centinela)
{
 cuenta++;
 suma += nota;
 printf ("Introduzca la siguiente nota: ");
 scanf ("%d", ¬a);
} /* fin de while */
puts ("Final");
```

#### Ejecución

Si se lee el primer valor de nota, por ejemplo 25 y luego se ejecuta, la salida **podría** ser ésta:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1 /* valor del centinela */
Final
```

### 6.1.6. Bucles controlados por indicadores (banderas)

En lenguajes, como Pascal, que tienen el tipo `bool`, se utiliza una variable booleana con frecuencia como indicadores o *banderas de estado* para controlar la ejecución de un bucle. El valor del indicador se inicializa (normalmente a falso "false") antes de la entrada al bucle y se redefine (normalmente a verdadero "true") cuando un suceso específico ocurre dentro del bucle. En C no existe el tipo booleano, por lo que se utiliza como bandera una variable entera que puede tomar dos valores, 1 o 0. Un *bucle controlado por bandera-indicador* se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

**Ejemplo 6.3**

Se desea leer diversos datos tipo carácter introducidos por teclado mediante un bucle *while* y se debe terminar el bucle cuando se lea un dato tipo dígito (rango '0' á '9').

La variable bandera, `digito-leido` se utiliza como un indicador que representa cuando un dígito se ha introducido por teclado.

**Variable bandera**

`digito-leido`

**Significado**

su valor es falso (cero) antes de entrar en el bucle y mientras el dato leído sea un carácter y es verdadero cuando el dato leído es un dígito.

El problema que se desea resolver es la lectura de datos carácter y la lectura debe detenerse cuando el dato leído sea numérico (un dígito de '0' a '9'). Por consiguiente, antes de que el bucle se ejecute y se lean los datos de entrada, la variable `digito_leido` se inicializa a falso (cero). Cuando se ejecuta el bucle, éste debe continuar ejecutándose mientras el dato leído sea un carácter y en consecuencia la variable `digito-leido` tiene de valor falso y se debe detener el bucle cuando el dato leído sea un dígito y en este caso el valor de la variable `digito-leido` se debe cambiar a verdadero (uno). En consecuencia la condición del bucle debe ser `!digito-leido` ya que esta condición es verdadera cuando `digito-leido` es falso. El bucle *while* será similar a:

```
digito-leido = 0; /* no se ha leído ningún dato */
while (!digito_leido)
{
 printf ("Introduzca un carácter: ");
 scanf ("%c", &car);
 digito-leido = (('0' <= car) && (car <= '9'));
 ...
} /* fin de while */
```

El bucle funciona de la siguiente forma:

1. Entrada del bucle: la variable `digito-leido` tiene por valor «falso».
2. La condición del bucle `!digito_leido` es verdadera, por consiguiente se ejecutan las sentencias del interior del bucle.
3. Se introduce por teclado un dato que se almacena en la variable `car`. Si el dato leído es un carácter, la variable `digito-leido` se mantiene con valor falso (0) ya que ése es el resultado de la sentencia de asignación.

```
digito-leido = (('0' <= car) && (car <= '9'));
```

Si el dato leído es un dígito, entonces la variable `digito-leido` toma el valor verdadero (1), resultante de la sentencia de asignación anterior.

4. El bucle se termina cuando se lee un dato tipo dígito ('0' a '9') ya que la condición del bucle es falsa.

**Modelo de bucle controlado por un indicador**

El formato general de un bucle controlado por indicador es el siguiente:

1. Establecer el indicador de control del bucle a «falso» o «verdadero» (a cero o a uno) con el objeto de que se ejecute el bucle *while* correctamente la primera vez (normalmente se suele inicializar a «falso»).

## 2. Mientras la condición de control del bucle sea verdadera:

### 2.1. Realizar las sentencias del cuerpo del bucle.

2.2. Cuando se produzca la condición de salida (en el ejemplo anterior que el dato carácter leído fuese un dígito) se cambia el valor de la variable indicador o bandera, con el objeto de que entonces la condición de control se haga falsa y, por tanto, el bucle se termina.

## 3. Ejecución de las sentencias siguientes al bucle.

### Ejemplo 6.4

Se desea leer un dato numérico  $x$  cuyo valor ha de ser mayor que cero para calcular la función  $f(x) = x \cdot \log(x)$ .

La variable bandera, `x_positivo` se utiliza como un indicador que representa que el dato leído es mayor que cero. Por consiguiente, antes de que el bucle se ejecute y se lea el dato de entrada, la variable `x_positivo` se inicializa a falso (0). Cuando se ejecuta el bucle, éste debe continuar ejecutándose mientras el número leído sea negativo o cero y en consecuencia la variable `x_positivo` tenga el valor falso y se debe detener el bucle cuando el número leído sea mayor que cero y en este caso el valor de la variable `x_positivo` se debe cambiar a verdadero (uno). En consecuencia la condición del bucle debe ser `!x_positivo` ya que esta condición es verdadera cuando `x_positivo` es falso. A la salida del bucle se calcula el valor de la función y se escribe:

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

int main()
{
 float f,x;
 int xpositivo;
 x_positivo = 0; /* inicializado a falso */
 while (!x_positivo)
 {
 printf("\n Valor de x: ");
 scanf("%f",&x);
 xpositivo = (x > 0.0); /* asigna verdadero(1) si cumple la
 condición*/
 }
 f = x*log(x);
 printf(" f(%.1f) = %.3e",x,f);
 return 0;
}
```

### 6.1.7. La sentencia `break` en los bucles

La sentencia `break` se utiliza, a veces, para realizar una terminación anormal del bucle. Dicho de otro modo, una terminación antes de lo previsto. Su sintaxis es:

**break;**

La sentencia `break` se utiliza para la salida de un bucle `while` o `do-while`, aunque también se puede utilizar dentro de una sentencia `switch`, siendo éste su uso más frecuente.

```

while (condición)
{
 if (condición2)
 break;
 /* sentencias */
}

```

---

### Ejemplo 6.5

El siguiente código extrae y visualiza valores de entrada desde el dispositivo estándar de entrada (*stdin*) hasta que se encuentra un valor especificado

```

int clave = -9;
int entrada;
while (scanf("%d",&entrada))
{
 if (entrada != clave)
 printf("%d\n",entrada);
 else
 break;
}

```

---

¿Cómo funciona este bucle `while`? La función `scanf()` devuelve el número de datos captados de dispositivo de entrada o bien cero si se ha introducido fin-de-fichero. Al devolver un valor distinto de cero el bucle se ejecutaría indefinidamente, sin embargo, cuando `entrada==clave` la ejecución sigue por `else` y la sentencia `break` que hace que la ejecución siga en la sentencia siguiente al bucle `while`.

### Ejecución

El uso de **break** en un bucle no es muy recomendable ya que puede hacer difícil la comprensión del comportamiento del programa. En particular, suele hacer muy difícil verificar los invariantes de los bucles. Por otra parte suele ser fácil la reescritura de los bucles sin la sentencia **break**. El bucle del Ejemplo 6.5 escrito sin la escritura de `break`:

```

int clave;
int entrada;
while ((scanf("%d",&entrada)) && (entrada != clave))
{
 printf("%d\n",entrada);
}

```

#### 6.1.8. Bucles `while (true)`

La condición que se comprueba en un bucle `while` puede ser cualquier expresión válida C. Mientras que la condición permanezca verdadera (distinto de 0), el bucle `while` continuará ejecutándose. Se puede crear un bucle que nunca termine utilizando el valor 1 (verdadero) para la condición que se comprueba.

```

1: /*Listado while (true) */
2: #include <stdio.h>
3: int main()

```

```

4: {
5: int flag = 1, contador = 0;
6: while (flag)
7: {
8: contador++;
9: if (contador > 10)
10: break;
11: }
12: printf("Contador: %d\n", contador);
13: return 0;
14: }

```

### Salida

Contador: 11

### Análisis

En la línea 6, un bucle `while` se establece con una condición que nunca puede ser falsa. El bucle incrementa la variable `contador` en la línea 8, y a continuación la línea 9 comprueba si el contador es mayor que 10. Si no es así el bucle se itera de nuevo. Si `contador` es mayor que 10, la sentencia `break` de la línea 10 termina el bucle `while`, y la ejecución del programa pasa a la línea 12.

## Ejercicio 6.1

*Calcular la media de seis números.*

El cálculo típico de una media de valores numéricos es: leer sucesivamente los valores, sumarlos y dividir la suma total por el número de valores leídos. El código más simple podría ser:

```

float num1;
float num2;
float num3;
float num4;
float num5;
float num6;
float media;
scanf("%f %f %f %f %f", &num1, &num2, &num3, &num4, &num5, &num6);
media = (num1+num2+num3+num4+num5+num6) / 6;

```

Evidentemente, si en lugar de 6 valores fueran 1.000, la modificación del código no sólo sería de longitud enorme sino que la labor repetitiva de escritura sería tediosa. Por ello, la necesidad de utilizar un bucle. El algoritmo más simple sería:

```

definir número de elementos como constante de valor 6
Inicializar contador de números
Inicializar acumulador (sumador) de números
Mensaje de petición de datos
mientras no estén leídos todos los datos hacer
 Leer número
 Acumular valor del número a variable acumulador
 Incrementar contador de números
fin-mientras
Calcular media (Acumulador/Total número)
Visualizar valor de la media
Fin

```

El código en C es:

```

/* Calculo de la media de seis números */
#include <stdio.h>
#include <string.h>

int main()
{
 const int TotalNum = 6;
 int ContadorNum = 0;
 float SumaNum = 0;
 float media;
 printf("Introduzca %d números\n", TotalNum);
 while (ContadorNum < TotalNum)
 {
 /* valores a procesar */
 float numero;
 scanf("%f", &numero); /* leer siguiente número */
 SumaNum += numero; /* añadir valor a Acumulador */
 ++ContadorNum; /* incrementar números leídos */
 }
 media = SumaNum/ContadorNum;
 printf("Media: %.2f \n", media);
 return 0;
}

```

## 6.2. REPETICIÓN: EL BUCLE `for`

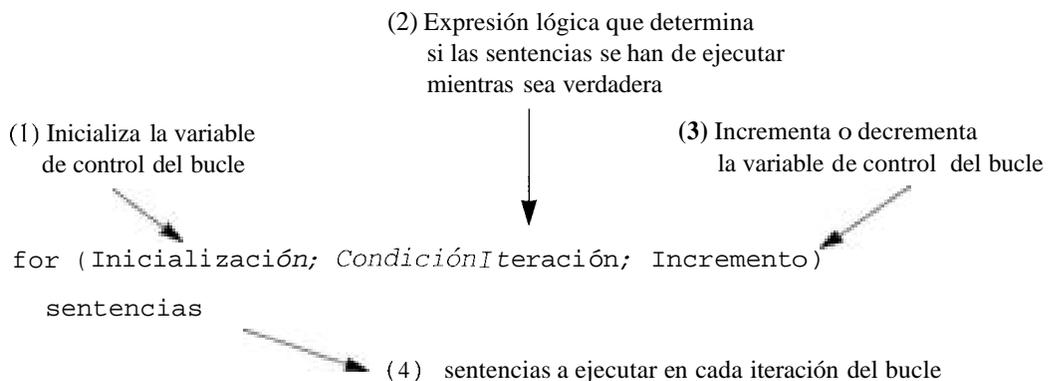
El bucle `for` de C es superior a los bucles `for` de otros lenguajes de programación tales como BASIC, Pascal y Fortran ya que ofrece más control sobre la inicialización y el incremento de las variables de control del bucle.

Además del bucle `while`, C proporciona otros dos tipos de bucles `for` y `do`. El bucle `for` que se estudia en esta sección es el más adecuado para implementar *bucles controlados por contador* que son bucles en los que un conjunto de sentencias se ejecutan una vez por cada valor de un rango especificado, de acuerdo al algoritmo:

*por cada valor de una variable-contador de un rango específico: ejecutar sentencias*

La sentencia `for` (bucle `for`) es un método para ejecutar un bloque de sentencias un número fijo de veces. El bucle `for` se diferencia del bucle `while` en que las operaciones de control del bucle se sitúan en un solo sitio: la cabecera de la sentencia.

### Sintaxis



El bucle `for` contiene las cuatro partes siguientes:

- **Parte de inicialización**, que inicializa las variables de control del bucle. Se pueden utilizar variables de control del bucle simples o múltiples.
- **Parte de condición**, que contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera.
- **Parte de incremento**, que incrementa o decrementa la variable o variables de control del bucle.
- **Sentencias**, acciones o sentencias que se ejecutarán por cada iteración del bucle.

La sentencia `for` es equivalente al siguiente código `while`

```

inicialización;
while (condiciónIteración)
{
 sentencias del bucle for;
 incremento;
}

```

---

### Ejemplo 1

```

int i;
/* imprimir Hola 10 veces */
for (i = 0; i < 10; i++)
 printf("Hola!");

```

---

### Ejemplo 2

```

int i;
for (i = 0; i < 10; i++)
{
 printf("Hola!\n");
 printf("El valor de i es: %d", i);
}

```

---

### Ejemplo 3

```

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

#define M 15
#define f(x) exp(2*x) - x

int main()
{
 int i;
 double x;
 for (i = 1; i <= M; i++)
 {
 printf("Valor de x: ");
 scanf("%lf",&x);
 printf("f(%.1lf) = %.4g\n",x,f(x));
 }
 return 0;
}

```

---

En este ejemplo se define la constante simbólica M y una «función en línea» (también llamada una macro con argumentos). El bucle se realiza 15 veces; cada iteración pide un valor de x, calcula la función y escribe los resultados. El diagrama de sintaxis de la sentencia `for` es:

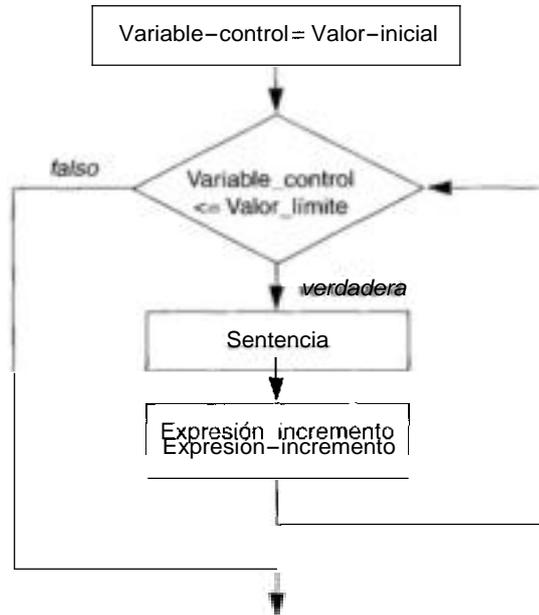


Figura 6.2. Diagrama de sintaxis de un bucle `for`

Existen dos formas de implementar la sentencia `for` que se utilizan normalmente para implementar bucles de conteo: *formato ascendente*, en el que la variable de control se incrementa y *formato descendente*, en el que la variable de control se decrementa.

```

for (var_control=valor_inicial; var_control<=valor_límite; exp_incremento)
 sentencia ;
 formato ascendente

for (var_control=valor_inicial; var_control>=valor_límite; exp_decremento)
 sentencia ;
 formato descendente

```

*Ejemplo de formato ascendente*

```

int n;

for (n = 1; n <= 10; n++)
 printf("%d \t %d \n", n, n * n);

```

La variable de control es `n` y su valor inicial es 1 de tipo entero, el valor límite es 10 y la expresión de incremento es `n++`. Esto significa que el bucle ejecuta la sentencia del cuerpo del bucle una vez por cada valor de `n` en orden ascendente 1 a 10. En la primera iteración (pasada) `n` tomará el valor 1; en la segunda iteración el valor 2 y así sucesivamente hasta que `n` toma el valor 10. La salida que se producirá al ejecutarse el bucle será:

```

1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100

```

### Ejemplo de formato descendente

```

int n;
for (n = 10; n > 5; n--)
 printf("%d \t %d \n", n, n * n);

```

La salida de este bucle es:

```

10 100
9 81
8 64
7 49
6 36

```

debido a que el valor inicial de la variable de control es 10, y el límite que se ha puesto es  $n > 5$  (es decir, verdadera cuando  $n = 10, 9, 8, 7, 6$ ); la expresión de decremento es el operador de decremento  $n--$  que decrementa en 1 el valor de  $n$  tras la ejecución de cada iteración.

### Otros intervalos de incremento/decremento

Los rangos de incremento/decremento de la variable o expresión de control del bucle pueden ser cualquier valor y no siempre 1, es decir 5, 10, 20, 4, ..., dependiendo de los intervalos que se necesiten. Así el bucle

```

int n;
for (n = 0; n < 100; n += 20)
 printf("%d \t %d \n", n, n * n);

```

utiliza la expresión de incremento

```
n += 20
```

que incrementa el valor de  $n$  en 20, dado que equivale a  $n = n + 20$ . Así la salida que producirá la ejecución del bucle es:

```

0 0
20 400
40 1600
60 3600
80 6400

```

---

## Ejemplos

```

/* ejemplo 1 */
int c;
for (c = 'A'; c <= 'Z'; c++)
 printf("%c ", c);
/* ejemplo 2 */

```

```

for (i = 9; i >= 0; i -= 3)
 printf("%d ", (i * i));

/* ejemplo 3 */
for (i = 1; i < 100; i*=2)
 printf("%d ", i);

/* ejemplo 4 */
#define MAX 25
int i, j;
for (i = 0, j = MAX; i < j; i++, j--)
 printf("%d ", (i + 2 * j));

```

El primer ejemplo inicializa la variable de control del bucle `c` al carácter 'A', equivale a inicializar al entero 65 (ASCII de A), e itera mientras que el valor de la variable `c` es menor o igual que el ordinal del carácter 'Z'. La parte de incremento del bucle incrementa el valor de la variable en 1. Por consiguiente, el bucle se realiza tantas veces como letras mayúsculas.

El segundo ejemplo muestra un bucle descendente que inicializa la variable de control a 9. El bucle se realiza mientras que `i` no sea negativo, como la variable se decreuenta en 3, el bucle se ejecuta cuatro veces con el valor de la variable de control `i`, 9, 6, 3 y 0.

El ejemplo 3, la variable de control `i` se inicializa a 1 y se incrementa en múltiplos de 2, por consiguiente, `i` toma valores de 1, 2, 4, 8, 16, 32, 64 y el siguiente 128 no cumple la condición, termina el bucle.

El ejemplo 4, declara dos variables de control `i` y `j` y las inicializa a 0 y a la constante MAX. El bucle se ejecutará mientras `i` sea menor que `j`. Las variable de control `i` se incrementa en 1, y a la vez `j` se decreuenta en 1.

---

## Ejemplo 6.6

### Suma de los 10 primeros números pares

```

#include <stdio.h>

int main()

 int n, suma = 0;
 for (n = 1; n <= 10; n++)
 suma += 2*n;

 printf("La suma de los 10 primeros números pares: %d", suma);
 return 0;
}

```

El bucle lo podríamos haber diseñado con un incremento de 2:

```

for (n = 2; n <= 20; n+=2)
 suma += n;

```

### 6.2.1. Diferentes usos de bucles `for`

El lenguaje C permite:

- El valor de la variable de control se puede modificar en valores diferentes de 1.
- Se puedan utilizar más de una variable de control.

#### *Ejemplos de incrementos/decrementos con variables de control diferentes*

La/s variable/s de control se pueden incrementar o decreuntar en valores de tipo `int`, pero también es posible en valores de tipo `float` o `double` y en consecuencia se incrementaría o decreuntaría en una cantidad decimal

```

int n;
for (n = 1; n <= 10; n = n + 2)
 printf("n es ahora igual a %d ",n);

int n,v=9;
for (n = v; n >= 100; n = n - 5)
 printf("n es ahora igual a %d ",n);

double p;
for (p= 0.75; p<= 5; p+= 0.25)
 printf("Perímetro es ahora igual a %.2lf ",p);

```

La expresión de incremento en **ANSI C** no necesita incluso ser una suma o una resta. Tampoco se requiere que la inicialización de una variable de control sea igual a una constante. Se puede inicializar y cambiar una variable de control del bucle en cualquier cantidad que se desee. Naturalmente cuando la variable de control no sea de tipo `int`, se tendrán menos garantías de precisión. Por ejemplo, el siguiente código muestra un medio más para arrancar un bucle `for`.

```

double x;
for (x = pow(y,3.0); x > 2.0; x = sqrt(x))
 printf("x es ahora igual a %.5e",x);

```

### 6.3. PRECAUCIONES EN EL USO DE `for`

Un bucle `for` se debe construir con gran precaución, asegurándose que la expresión de inicialización, la condición del bucle y la expresión de incremento harán que la condición del bucle se convierta en false en algún momento. En particular: *«si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición del bucle, entonces el número de repeticiones se puede modificar»*.

Esta regla anterior es importante, ya que **su** aplicación se considera una mala práctica de programación. Es decir, no es recomendable modificar el valor de cualquier variable de la condición del bucle dentro del cuerpo de un bucle `for`, ya que se pueden producir resultados imprevistos. Por ejemplo, la ejecución de

```

int i, limite = 11;
for (i = 0; i <= limite; i++)
{
 printf("%d\n",i);
 Ilimite++;
}

```

produce una secuencia infinita de enteros (puede terminar si el compilador tiene constantes `MAXINT`, con máximos valores enteros, entonces la ejecución terminará cuando `i` sea `MAXINT` y `limite` sea `MAXINT+1 = MININT`).

```

0
1
2
3

```

ya que a cada iteración, la expresión `limite++` incrementa `limite` en 1, antes de que `i++` incremente `i`. A consecuencia de ello, la condición del bucle `i <= limite` siempre es cierta.

Otro ejemplo de bucle mal programado:

```
int i, limite = 1;

for (i = 0; i <= limite; i++)
{
 printf("%d\n", i);
 i--;
}
```

que producirá infinitos ceros

```
0
0
0
```

ya que en este caso la expresión `i--` del cuerpo del bucle decrementa `i` en 1 antes de que se incremente la expresión `i++` de la cabecera del bucle en 1. Como resultado `i` es siempre 0 cuando el bucle se comprueba. En este ejemplo la condición para terminar el bucle depende de la entrada, el bucle está mal programado:

```
#define LIM 50
int iter, tope;
for (iter = 0; tope <= LIM; iter++)
{
 printf("%d\n", iter);
 scanf("%d", &tope);
}
```

### 6.3.1. Bucles infinitos

El uso principal de un bucle `for` es implementar bucles de conteo en el que el número de repeticiones se conoce por anticipado. Por ejemplo, la suma de enteros de 1 a  $n$ . Sin embargo, existen muchos problemas en los que el número de repeticiones no se pueden determinar por anticipado. Para estas situaciones algunos lenguajes modernos tienen sentencias específicas tales como las sentencias `LOOP` de Modula-2 y Modula-3, el bucle `DO` de FORTRAN 90 o el bucle `loop` de Ada. C no soporta una sentencia que realice esa tarea, pero existe una variante de la sintaxis de `for` que permite implementar **bucles infinitos** que son aquellos bucles que, en principio, no tienen fin.

#### Sintaxis

```
for (;;)
 sentencia ;
```

La *sentencia* se ejecuta indefinidamente a menos que se utilice una sentencia `return` o `break` (normalmente una combinación `if-break` o `if-return`).

La razón de que el bucle se ejecute indefinidamente es que se ha eliminado la expresión de inicialización, la condición del bucle y la expresión de incremento; al no existir una condición de bucle que especifique cual es la condición para terminar la repetición de sentencias, asume que la condición es verdadera (1) y éstas se ejecutarán indefinidamente. Así, el bucle

```
for (;;)
 printf("Siempre así, te llamamos siempre así...\n");
```

producirá la salida

```
Siempre así, te llamamos siempre así...
Siempre así, te llamamos siempre así...
...
```

un número ilimitado de veces, a menos que el usuario interrumpa la ejecución (normalmente pulsando las teclas `Ctrl` y `C` en ambientes PC).

Para evitar esta situación, se requiere el diseño del bucle `for` de la forma siguiente:

1. El cuerpo del bucle ha de contener todas las sentencias que se desean ejecutar repetidamente.
2. Una sentencia terminará la ejecución del bucle cuando se cumpla una determinada condición.

La sentencia de terminación suele ser `if-break` con la sintaxis

```
if (condición) break;
```

|                    |                                                                                          |
|--------------------|------------------------------------------------------------------------------------------|
| <i>condición</i>   | es una expresión lógica                                                                  |
| <code>break</code> | termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle |

y la sintaxis completa

```
for (;;) /* bucle */
{
 lista-sentencias,
 if (condición-terminación) break;
 lista-sentencias,
} /* fin del bucle */
```

*lista-sentencias* puede ser vacía, simple o compuesta.

### Ejemplo 6.7

```
#define CLAVE -999
for (;;)
{
 printf("Introduzca un número, (%d) para terminar",CLAVE);
 scanf("%d",&num);
 if (num == CLAVE) break;
 ...
}
```

### 6.3.2. Los bucles `for` vacíos

Tenga cuidado de situar un punto y coma después del paréntesis inicial del bucle `for`. Es decir, el bucle

```
for (i = 1; i <= 10; i++); problema
 puts("Sierra Magina");
```

no se ejecuta correctamente, ni se visualiza la frase "Sierra Magina" 10 veces como era de esperar, ni se produce un mensaje de error por parte del compilador.

En realidad lo que sucede es que se visualiza una vez la frase "Sierra Magina" ya que la sentencia `for` es una sentencia vacía al terminar con un punto y coma (;). Sucede que la sentencia `for` no hace absolutamente nada durante 10 iteraciones y, por tanto, después de que el bucle `for` haya terminado, se ejecuta la siguiente sentencia `puts` y se escribe "Sierra Magina".

El bucle `for` con cuerpos vacíos puede tener algunas aplicaciones, especialmente cuando se requieren ralentizaciones o temporizaciones de tiempo.

### 6.3.3. Sentencias nulas en bucles `for`

Cualquiera o todas las sentencias de un bucle `for` pueden ser nulas. Para ejecutar esta acción, se utiliza el punto y coma (;) para marcar la sentencia vacía. Si se desea crear un bucle `for` que actúe exactamente como un bucle `while`, se deben incluir las primeras y terceras sentencias vacías.

```

1: /* Listado
2: bucles for con sentencias nulas
3: */
4: #include <stdio.h>
5:
6: int main()
7: {
8: int contador = 0;
9:
10: for (;contador < 5;)
11: {
12: contador++;
13: printf(";Bucle!");
14: }
15:
16: printf("\n Contador: %d \n", Contador);
17: return 0;
18: }
```

*Salida*

```

;Bucle! ;Bucle! ;Bucle! ;Bucle! ;Bucle!
Contador: 5
```

*Análisis*

En la línea 8 se inicializa la variable del contador. La sentencia `for` en la línea 10 no inicializa ningún valor, pero incluye una prueba de `contador < 5`. No existe ninguna sentencia de incrementación, de modo que el bucle se comporta exactamente como la sentencia siguiente.

```

while(contador < 5)
{
 contador++;
 printf(";Bucle!");
}
```

### 6.3.4. Sentencias `break` y `continue`

La sentencia `break` termina la ejecución de un bucle, de una sentencia `switch`, en general de cualquier sentencia.

```
/*
Ejemplo de utilización de break
*/
#include <stdio.h>

int main()

 int contador = 0; /* inicialización */
 int max;
 printf ("Cuantos holas? ");
 scanf ("%d",&max);
 for (;;) /* bucle for que no termina nunca */
 {
 if(contador < max) /* test */
 {
 puts("Hola!");
 contador++; /* incremento */
 }
 else
 break;
 }
 return 0;
}
```

### Salida

```
Cuantos holas? 3
Hola!
Hola!
Hola!
```

La sentencia continue hace que la ejecución de un bucle vuelva a la cabecera del bucle.

```
#include <stdio.h>
int main()
{
 int clave,i;
 puts ("Introduce -9 para acabar.");
 clave = 1;
 for (i = 0; i < 8; i++) {
 if (clave == -9) continue;
 scanf ("%d",&clave);
 printf ("clave %d\n",clave);
 }
 printf("VALORES FINALES i = %d clave = %d",i,clave);
 return 0;
}
```

### Ejecución

```
Introduce -9 para acabar
4
clave 4
```

```

7
clave 7
-9
VALORES FINALES i = 8 clave = -9

```

La sentencia `continue` ha hecho que la ejecución vuelva a la cabecera del bucle `for`, como no se vuelve a cambiar el valor de `clave`, realiza el resto de las iteraciones hasta que `i` vale 8.

### 6.4. REPETICIÓN: EL BUCLE `do...while`

La sentencia `do-while` se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener la seguridad de que una determinada acción se ejecutará una o varias veces, pero al menos una vez.

**Sintaxis**

Acción (sentencia) a ejecutar al menos una vez

Expresión lógica que determina si la acción se repite

1. `do sentencia while (expresión)`
2. `do  
sentencia  
while (expresión)`

La construcción `do` comienza ejecutando `sentencia`. Se evalúa a continuación `expresión`. Si `expresión` es verdadera, entonces se repite la ejecución de `sentencia`. Este proceso continúa hasta que `expresión` es falsa. La semántica del bucle `do` se representa gráficamente en la Figura 6.3.

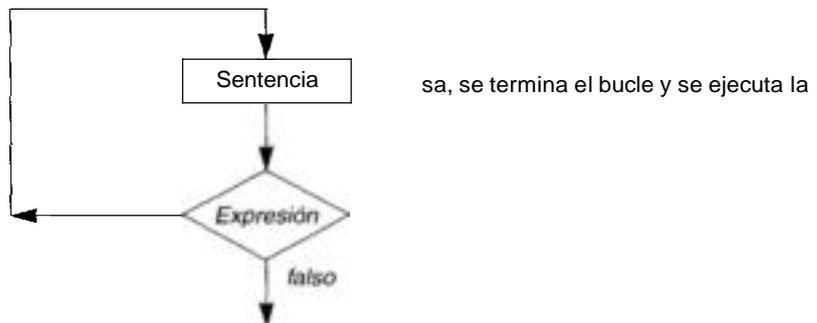


Figura 6.3. Diagrama de flujo de la sentencia `do`.

**Ejemplo 6.8**

*Bucle para introducir un dígito.*

```
do
{
 printf ("Introduzca un dígito (0-9) : ");
 scanf ("%c",&digito);
} while ((digito < '0') || ('9' < digito));
```

Este bucle se realiza mientras se introduzcan dígitos y se termina cuando se introduzca un carácter que no sea un dígito de '0' a '9'.

**Ejercicio 6.2**

*Aplicación simple de un bucle while: seleccionar una opción de saludo al usuario dentro de un programa.*

```
#include <stdio.h>
int main()
{
 char opcion;
 do
 {
 puts ("Hola");
 puts ("¿Desea otro tipo de saludo?");
 puts ("Pulse s para si y n para no,");
 printf ("y a continuación pulse intro: ");
 scanf ("%c",&opcion);
 } while (opcion == 's' || opcion == 'S');
 puts ("Adiós");
 return 0;
}
```

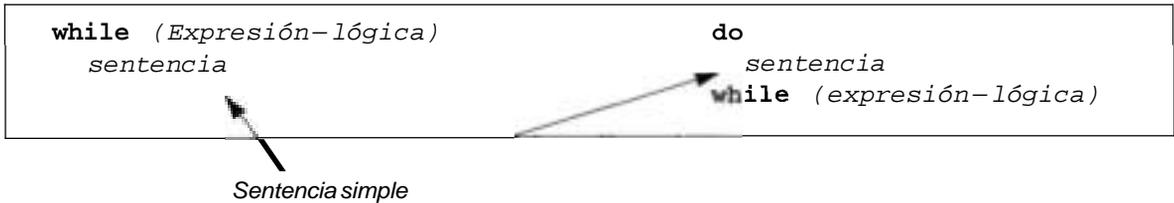
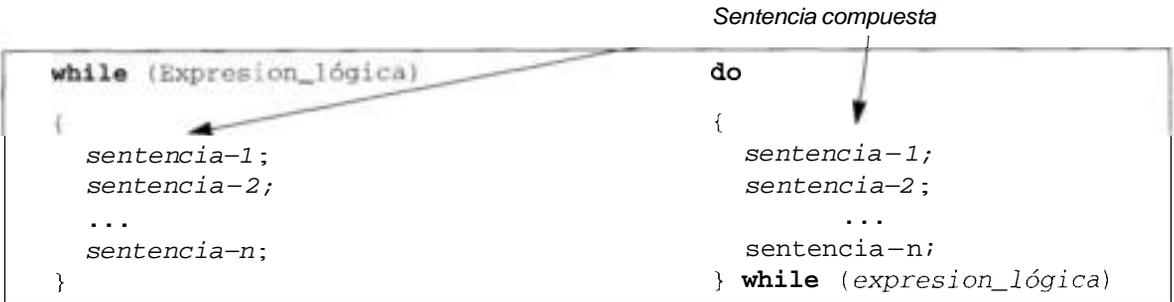
*Salida de muestra*

```
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: S
Hola
¿Desea otro tipo de saludo?
Pulse s para si y n para no,
y a continuación pulse intro: N
Adiós
```

**6.4.1. Diferencias entre while y do-while**

Una sentencia **do-while** es similar a una sentencia **while** excepto que el cuerpo del bucle se ejecuta siempre al menos una vez

**Sintaxis**



**Ejemplo 1**

```

/*
 cuenta de 0 a 10 (sin incluir el 10)
*/
int x = 0;
do
 print("X: %d",x++);
while (x < 10);

```

**Ejemplo 2**

```

/*
 Bucle para imprimir las letras minúsculas del alfabeto
*/
char car = 'a';
do
{
 printf("%d ",car);
 car++;
}while (car <= 'z');

```

**Ejemplo 6.9**

Visualizar las potencias de 2 cuyos valores estén en el rango 1 a 1.000.

```

/* Realizado con while */
potencia = 1;
while (potencia < 1000)

 printf("%d \n",potencia);
 potencia * = 2;
} /* fin de while */

```

```

/* Realizado con do-while */
potencia = 1;
do
{
 printf("%d \n",potencia);
 potencia * = 2;
} while (potencia < 1000);

```

## 6.5. COMPARACIÓN DE BUCLES `while`, `for` Y `do-while`

C proporciona tres sentencias para el control de bucles: `while`, `for` y `do-while`. El bucle `while` se repite *mientras* su condición de repetición del bucle es verdadero; el bucle `for` se utiliza normalmente cuando el conteo esté implicado, o bien el número de iteraciones requeridas se pueda determinar al principio de la ejecución del bucle, o simplemente cuando exista una necesidad de seguir el número de veces que un suceso particular tiene lugar. El bucle `do-while` se ejecuta de un modo similar a `while` excepto que las sentencias del cuerpo del bucle se ejecutan siempre al menos una vez.

La Tabla 6.1 describe cuando se usa cada uno de los tres bucles. En C, el bucle `for` es el más frecuentemente utilizado de los tres. Es relativamente fácil reescribir un bucle `do-while` como un bucle `while`, insertando una asignación inicial de la variable condicional. Sin embargo, no todos los bucles `while` se pueden expresar de modo adecuado como bucles `do-while`, ya que un bucle `do-while` se ejecutará siempre al menos una vez y el bucle `while` puede no ejecutarse. Por esta razón un bucle `while` suele preferirse a un bucle `do-while`, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 6.1. Formatos de los bucles.

|                       |                                                                                                                                                                                                                                                                                                                             |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>while</code>    | El uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.                                                              |
| <code>for</code>      | Bucle de conteo, cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y de actualización; el test de la condición precede a la ejecución del cuerpo del bucle. |
| <code>do-while</code> | Es adecuada para asegurar que al menos se ejecuta el bucle una vez.                                                                                                                                                                                                                                                         |

### Comparación de tres bucles

```

cuenta = valor-inicial;
while (cuenta < valor-parada)
{
 ...
 cuenta++;
} /* fin de while */

for (cuenta = valor-inicial; cuenta < valor-parada; cuenta++)
{
 ...
} /* fin de for */

cuenta = valor-inicial;
if (valor-inicial < valor-parada)
do
{
 ...
 cuenta++;
}while (cuenta < valor-parada);

```

## 6.6. DISEÑO DE BUCLES

El diseño de un bucle necesita tres puntos a considerar:

1. El *cuero* del bucle.
2. Las sentencias de *inicialización*.
3. Las condiciones para la *terminación* del bucle.

### 6.6.1. Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican la lectura de una lista de números y calculan su suma. Si se conoce cuántos números habrá, tal tarea se puede ejecutar fácilmente por el siguiente pseudocódigo. El valor de la variable `total` es el número de valores que se suman. La suma se acumula en la variable `suma`.

```
suma = 0;
repetir lo siguiente total veces:
 leer(siguiente);
 suma = suma + siguiente;
fin-bucle
```

Este código se implementa fácilmente con un bucle `for`

```
int cuenta, suma = 0;
for (cuenta = 1; cuenta <= total; cuenta++)
{
 scanf("%d ", &siguiente);
 suma = suma + siguiente;
}
```

Obsérvese que la variable `suma` se espera tome un valor cuando se ejecuta la siguiente sentencia

```
suma = suma + siguiente;
```

Dado que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, `suma` debe estar inicializada a algún valor antes de que se ejecute el bucle. Con el objeto de determinar el valor correcto de inicialización de `suma` se debe pensar sobre qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser ese número. Esto es, la primera vez que se ejecute el bucle el valor de `suma + siguiente` será igual a `siguiente`. Para hacer esta operación, el valor de `suma` debe ser inicializado a 0. Si en lugar de `suma`, se desea realizar productos de una lista de números, la técnica a utilizar es:

```
int cuenta, producto;
for (cuenta = producto = 1; cuenta <= total; cuenta++)
{
 scanf("%d", &siguiente);
 producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial, se inicializa junto a `cuenta` en la expresión de inicialización a 1. No se debe suponer que todas las variables se deben inicializar a cero. Si `producto` se inicializa a cero, seguiría siendo cero después de que el bucle anterior terminara.

### 6.6.2. Final de un bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada. Estos cuatro métodos son:

1. Alcanzar el tamaño de la secuencia de entrada.
2. Preguntar antes de la iteración.
3. Secuencia de entrada terminada con un valor centinela.
4. Agotamiento de la entrada.

### Tamaño de la secuencia de entrada

Si su programa puede determinar el tamaño de la secuencia de entrada por anticipado, bien preguntando al usuario o por algún otro método, se puede utilizar un bucle «repetir  $n$  veces» para leer la entrada exactamente  $n$  veces, en donde  $n$  es el tamaño de la secuencia.

### Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar, simplemente al usuario, después de cada iteración del bucle, si el bucle debe ser o no iterado de nuevo. Por ejemplo:

```
int numero, suma = 0;
char resp = 'S';
while ((resp == 'S' || (resp == 's'))
{
 printf("Introduzca un número:");
 scanf("%d",&numero);
 suma += numero;
 printf("¿Existen más números?(S para Si, N para No): ");
 scanf("%d",&resp);
}
```

Este método es muy tedioso para listas grandes de números. Cuando se lea una lista larga es preferible incluir una Única señal de parada, como se incluye en el método siguiente.

### Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es con un valor centinela. Un **valor centinela** es aquel que es totalmente distinto de todos los valores posibles de la lista que se está leyendo y de este modo indica el final de la lista. Un ejemplo típico se presenta cuando se lee una lista de números positivos; un número negativo se puede utilizar como un valor centinela para indicar el final de la lista.

```
/* ejemplo de valor centinela (número negativo) */

puts("Introduzca una lista de enteros positivos");
puts("Termine la lista con un número negativo");
suma = 0;
scanf("%d",&numero);
while (numero >= 0)
{
 suma += numero;
 scanf("%d",&numero);
}
printf("La suma es: %d\n", suma);
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4 8 15 -99
```

el valor de la suma será 27. Es decir, -99, Último número de la entrada de datos no se añade a suma. -99 es el último dato de la lista que actúa como centinela y no forma parte de la lista de entrada de números.

### Agotamiento de la entrada

Cuando se leen entradas de un archivo, se puede utilizar un valor centinela, aunque el método más frecuente es comprobar simplemente si todas las entradas del archivo han sido procesadas y se alcanza el final del bucle cuando no hay más entradas a leer. Éste es el método usual en la lectura de archivos,

que se suele utilizar una marca al final de archivo, `eof`. En el capítulo de archivos se dedicará una atención especial a la lectura de archivos con una marca de final de archivo.

### 6.6.3. Otras técnicas de terminación de bucle

Las técnicas más usuales para la terminación de bucles de cualquier tipo son:

1. Bucles controlados por contador.
2. Preguntar antes de iterar.
3. Salir con una condición bandera.

Un bucle **controlado por contador** es cualquier bucle que determina el número de iteraciones antes de que el bucle comience y a continuación repite (itera) el cuerpo del bucle esas iteraciones. La técnica de la secuencia de entrada precedida por su tamaño es un ejemplo de un bucle controlado por contador.

La técnica de **preguntar antes de iterar** se puede utilizar para bucles distintos de los bucles de entrada, pero el uso más común de esta técnica es para procesar la entrada. La técnica del valor centinela es una técnica conocida también como **salida con una condición bandera o señalizadora**. Una variable que cambia su valor para indicar que algún suceso o evento ha tenido lugar, se denomina normalmente **bandera o indicador**. En el ejemplo anterior de suma de números, la variable bandera es `numero` de modo que cuando toma un valor negativo significa que indica que la lista de entrada ha terminado.

### 6.6.4. Bucles `for` vacíos

La sentencia nula (`;`) es una sentencia que está en el cuerpo del bucle y no hace nada. Un bucle `for` se considera vacío si consta de la cabecera y de la sentencia nula (`;`).

---

#### Ejemplo

Muestra los valores del contador, de 0 a 4.

```

1: /*
2: Ejemplo de la sentencia nula en for.
3: */
4: #include <stdio.h>
5: int main0
6: {
7: int i;
8: for (i = 0; i < 5; printf("i: %d\n", i++)) ;
9: return 0;
10: }
```

#### Salida

```

i: 0
i: 1
i: 2
i: 3
i: 4
```

#### Análisis

El bucle `for` de la línea 8 incluye tres sentencias: la sentencia de **inicialización** establece el valor inicial del contador `i` a 0. La sentencias de **condición** comprueba `i < 5`, y la sentencia acción imprime el valor de `i` y lo incrementa.

---

### Ejercicio 6.3

*Escribir un programa que visualice el factorial de un entero comprendido entre 2 y 20.*

El factorial de un entero  $n$  se calcula con un bucle *for* desde 2 hasta  $n$ , teniendo en cuenta que factorial de 1 es 1 ( $1! = 1$ ) y que  $n! = n*(n-1)!$ . Así, por ejemplo,

$$4! = 4*3! = 4*3*2! = 4*3*2*1! = 4*3*2*1 = 24$$

En el programa se escribe un bucle *do-while* para validar la entrada de  $n$ , entre 2 y 20. Otro bucle *for* para calcular el factorial. El bucle *for* va a ser vacío, en la expresión de incremento se va a calcular los  $n$  productos, para ello se utiliza el operador \*= junto al de decremento (--).

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

 long int n,m, fact;
 do

 printf ("\nFactorial de número n, entre 2 y 20: ");
 scanf ("%ld",&n);
 }while ((n <2) || (n > 20));

 for (m=n, fact=1; n>1; fact *= n--) ;

 printf ("%ld! = %ld",m, fact);
 return 0;
```

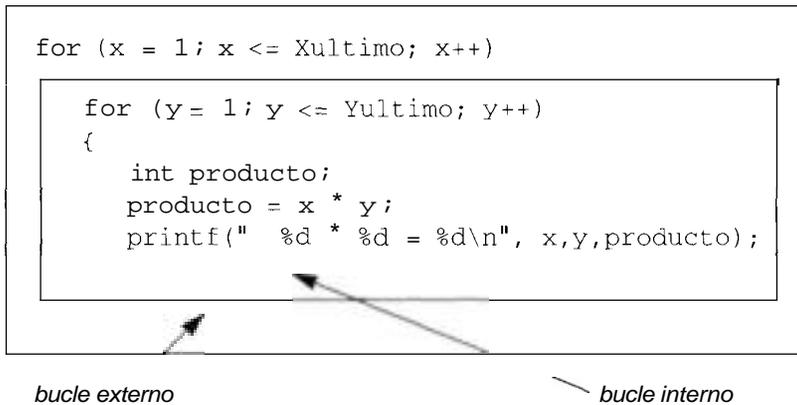
## 6.7. BUCLES ANIDADOS

Es posible *anidar* bucles. Los bucles anidados constan de un bucle externo con uno o más bucles internos. Cada vez que se repite el bucle externo, los bucles internos se repiten, se vuelven a evaluar los componentes de control y se ejecutan todas las iteraciones requeridas.

### Ejemplo 6.10

*El segmento de programa siguiente visualiza una tabla de multiplicación por cálculo y visualización de productos de la forma  $x * y$  para cada  $x$  en el rango de 1 a  $X_{ultimo}$  y desde cada  $y$  en el rango 1 a  $Y_{ultimo}$  (donde  $X_{ultimo}$ ,  $y$   $Y_{ultimo}$  son enteros prefijados). La Tabla que se desea obtener es*

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
```



El bucle que tiene *x* como variable de control se denomina **bucle externo** y el bucle que tiene *y* como variable de control se denomina **bucle interno**.

### Ejemplo 6.11

```

/*
 Escribe las variables de control de dos bucles anidados
*/
#include <stdio.h>

void main0
{
 int i,j;
 /* cabecera de la salida */
 printf("\n\t\t i \t j\n");
 for (i= 0; i < 4; i++)

 printf("Externo\t %d\n",i);
 for (j = 0; j < i; j++)
 printf("Interno\t\t %d \n",j);
 } /* fin del bucle externo */
}

```

La salida del programa es

|         | i | j |
|---------|---|---|
| Externo | 0 |   |
| Externo | 1 |   |
| Interno |   | 0 |
| Externo | 2 |   |
| Interno |   | 0 |
| Interno |   | 1 |
| Externo | 3 |   |
| Interno |   | 0 |
| Interno |   | 1 |
| Interno |   | 2 |

**Ejercicio 6.4**

Escribir un programa que visualice un triángulo isósceles.

```

 *
 * * *
 * * * * *
 * * * * * * *
 * * * * * * * *
* * * * * * * * *
```

El triángulo isósceles se realiza mediante un bucle externo y dos bucles internos. Cada vez que se repite el bucle externo se ejecutan los dos bucles internos. El bucle externo se repite cinco veces (cinco filas); el número de repeticiones realizadas por los bucles internos se basan en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo bucle interno visualiza uno o más asteriscos.

```

#include <stdio.h>
/* constantes globales */
const int num_lineas = 5;
const char blanco = " ";
const char asterisco = '*';

void main()
{
 int fila, blancos, cuenta-as;

 puts(" "); /* Deja una línea de separación */

 /* bucle externo: dibuja cada línea */
 for (fila = 1; fila <= num_lineas; fila++)
 {
 putchar('\t');
 /*primer bucle interno: escribe espacios */
 for (blancos = num_lineas-fila; blancos > 0; blancos--)
 putchar(blanco);

 for (cuenta-as = 1; cuenta-as < 2 * fila; cuenta-as++)
 putchar(asterisco);

 /* terminar línea */
 puts(" ");
 } /* fin del bucle externo */
}

```

El bucle externo se repite cinco veces, uno por línea o fila; el número de repeticiones ejecutadas por los bucles internos se basa en el valor de `fila`. La primera fila consta de un asterisco y cuatro blancos, la fila 2 consta de tres blancos y tres asteriscos, y así sucesivamente; la fila 5 tendrá 9 asteriscos ( $2 \times 5 - 1$ ). En este ejercicio se ha utilizado para salida de un carácter la función `putchar()`. Esta función escribe un argumento de tipo carácter en la pantalla.

**Ejercicio 6.5**

Ejecutar el programa siguiente que imprime una tabla de  $m$  filas por  $n$  columnas y un carácter de entrada.

```

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

```

5: int tilas, columnas;
6: int i, j;
7: char elCar;
8: printf("¿Cuántas filas?: ");
9: scanf("%d",&filas);
10: printf("¿Cuántas columnas?: ");
11: scanf("%d",&columnas);
12: printf("¿Qué carácter?: ");
13: scanf("\n%c",&elCar);
14: for (i = 0; i < filas; i++)
15: {
16: for (j = 0; j < columnas; j++)
17: putchar(elCar);
18: putchar('\n');
19: }
20: return 0;
21: }

```

### Análisis

El usuario solicita el número de filas y de columnas y un carácter a imprimir. Merece la pena comentar, que para leer el carácter a escribir es necesario saltarse el carácter fin de línea (`scanf("\n%c",&elCar)`) que se encuentra en el buffer de entrada, debido a la petición anterior del número de columnas. El primer bucle `for` de la línea 14 inicializa un contador (`i`) a 0 y a continuación se ejecuta el cuerpo del bucle `for` externo.

En la línea 16 se inicializa otro bucle `for` y un segundo contador `j` se inicializa a 0 y se ejecuta el cuerpo del bucle interno. En la línea 17 se imprime el carácter `elCar` (\*). Se evalúa la condición (`j < columnas`) y si se evalúa a *true (verdadero)*, `j` se incrementa y se imprime el siguiente carácter. Esta acción se repite hasta que `j` sea igual al número de columnas.

El bucle interno imprime doce caracteres asterisco en una misma fila y el bucle externo repite cuatro veces (número de filas) la fila de caracteres.

### Ejercicio 6.6

*Escribir en pantalla el factorial de n, entre los valores 1 a 10.*

Con dos bucles *for* se solución el problema. El bucle externo determina el número *n* cuyo factorial se calcula en el bucle interno.

```

#include <stdio.h>
#define S 10
int main()
{
 long int n,m, fact;
 for (n = 1, n <= S; n++)
 {
 fact = 1;
 for (m=n ; m>1; m--)
 fact *= m;
 printf("\t %ld! = %ld \n",n, fact);
 }
 return 0;
}

```

## 6.8. RESUMEN

En programación es habitual tener que repetir la ejecución de una sentencia, esto es lo que se conoce como bucle. Un bucle es un grupo de instrucciones que se ejecutan repetidamente hasta que se cumple una condición de terminación. Los bucles representan estructuras de control repetitivas, el número de repeticiones puede ser establecido inicialmente, o bien hacerlo depender de una condición (verdadera o falsa).

Un bucle puede diseñarse de diversas formas, las más importantes son: repetición controlada por contador y repetición controlada por condición.

- Una variable de control del bucle se utiliza para contar las repeticiones de un grupo de sentencias. Se incrementa o decrementa normalmente en 1 cada vez que se ejecuta el grupo de sentencias.
- La condición de finalización de bucle se utiliza para controlar la repetición cuando el número de ellas (iteraciones) no se conoce por adelantado. Un valor centinela se introduce para determinar el hecho de que se cumpla o no la condición.
- Los bucles `for` inicializan una variable de control a un valor, a continuación comprueban una expresión; si la expresión es verdadera se ejecutan las sentencias del cuerpo del bucle. La

expresión se comprueba cada vez que termina la iteración, cuando es falsa se termina del bucle y sigue la ejecución en la siguiente sentencia. Es importante que en el cuerpo del bucle haya una sentencia que haga que alguna vez sea falsa la expresión que controla la iteración del bucle.

- Los bucles `while` comprueban una condición, si es verdadera, se ejecuta las sentencias del bucle. A continuación, se vuelve a comprobar la condición si sigue siendo verdadera, se ejecutan las sentencias del bucle; termina cuando la condición es falsa.

La condición está en la cabecera del bucle `while`, por eso el número de veces que se repite puede ser de 0 a n.

- Los bucles `do - while` también comprueban una condición; se diferencian de los bucles `while` en que comprueban la condición al final del bucle, en vez de en la cabecera.
- La sentencia `break` produce la salida inmediata del bucle.
- La sentencia `continue` hace que cuando se ejecuta se salten todas las sentencias que vienen a continuación, y comienza una nueva iteración si se cumple la condición del bucle.



## 6.9. EJERCICIOS

6.1. ¿Cuál es la salida del siguiente segmento de programa?

```
for (cuenta = 1; cuenta < 5; cuenta++)
 printf("%d ", (2 * cuenta));
```

6.2. ¿Cuál es la salida de los siguientes bucles?

```
A for (n = 10; n > 0; n = n-2)
```

```
{
 printf("Hola");
 printf(" %d \n",n);
}
B double n = 2;
for (; n > 0; n = n-0.5)
 printf("%g ",n);
```

63. Seleccione y escriba el bucle adecuado que mejor resuelva las siguientes tareas:

- a) Suma de la serie  $1/2+1/3+1/4+1/5+\dots+1/50$ .
- b) Lectura de la lista de calificaciones de un examen de Historia.
- c) Visualizar la suma de enteros en el intervalo  $11\dots50$ .

64. Considerar el siguiente código de programa

```
int i = 1;
while (i <= n) {
 if ((i % n) == 0) {
 ++i;
 }
}
printf("%d \n", i);
```

- a) ¿Cuál es la salida si n es 0?
- b) ¿Cuál es la salida si n es 1?
- c) ¿Cuál es la salida si n es 3?

65. Considérese el siguiente código de programa

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

- a) ¿Cuál es la salida si n es 0?
- b) ¿Cuál es la salida si n es 1?
- c) ¿Cuál es la salida si n es 3?

66. ¿Cuál es la salida de los siguientes bucles?

```
int n, m;
for (n = 1; n <= 10; n++)
for (m = 10; m >= 1; m--)
 printf(" %d veces %d=%d \n", n,
 m, n * m);
```

67. Escriba un programa que calcule y visualice

$1! + 2! + 3! + \dots + (n-1)! + n!$   
 donde n es un valor de un dato.

68. ¿Cuál es la salida del siguiente bucle?

```
suma = 0;
while (suma < 100)
 suma += 5;
printf(" %d \n", suma);
```

6.9. Escribir un bucle while que visualice todas las potencias de un entero n, menores que un valor especificado max\_limite.

6.10. ¿Qué hace el siguiente bucle while? Reescribirlo con sentencias for y do-while.

```
num = 10;
while (num <= 100)
{
 printf("%d \n", num);
 num += 10;
}
```

6.11. Suponiendo que m = 3 y n = 5. ¿Cuál es la salida de los siguientes segmentos de programa?

```
A for (i = 0; i < n; i++)
{
 for (j = 0; j < i; j++)
 putchar('*');
 putchar('\n');
}
```

```
B for (i = n; i > 0; i--)
{
 for (j = m; j > 0; j--)
 putchar('*');
 putchar('\n');
}
```

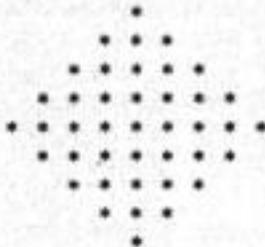
6.12. ¿Cuál es la salida de los siguientes bucles?

```
A for (i = 0; i < 10; i++)
 printf(" 2* %d = %d \n", i,
 2 * i);
```

```
B for (i = 0; i <= 5; i++)
 printf(" %d ", 2 * i + 1);
 putchar('\n');
```

```
C for (i = 1; i < 4; i++)
{
 printf(" %d ", i);
 for (j = i; j >= 1; j--)
 printf(" %d \n", j);
}
```

6.13. Escribir un programa que visualice el siguiente dibujo.



6.14. Describir la salida de los siguientes bucles.

```

A for (i = 1; i <= 5; i++)
 {
 printf(" %d \n",i);
 for (j = i; j >= 1; j-=2)
 printf(" %d \n",j);
 }

B for (i = 3; i > 0; i--)
 for (j = 1; j <= i; j++)
 for (k = i; k >= j; k--)
 printf("%d %d %d \n",
 i, j, k);

```

```

C for (i = 1; i <= 3; i++)
 for (j = 1; j <= 3; j++)
 {
 for (k = i; k <= j; k++)
 printf("%d %d %d \n", i,
 j, k);
 putchar('\n');
 }

```

6.15. ¿Cuál es la salida de este bucle?

```

i = 0;
while (i*i < 10)
{
 j = i
 while (j*j < 100)
 {
 printf("%d \n",i + j);
 j *= 2;
 }
 i++;
}
printf("\n*****\n");

```

## 6.10. PROBLEMAS

6.1. En una empresa de computadoras, los salarios de los empleados se van a aumentar según su contrato actual:

| Contrato                | Aumento % |
|-------------------------|-----------|
| 0 a 9.000 dólares       | 20        |
| 9.001 a 15.000 dólares  | 10        |
| 15.001 a 20.000 dólares | 5         |
| más de 20.000 dólares   | 0         |

Escribir un programa que solicite el salario actual del empleado y calcule y visualice el nuevo salario.

6.2. La constante  $\pi$  (3.141592...) es muy utilizada en matemáticas. Un método sencillo de calcular su valor es:

$$\pi = 4 * \left(\frac{2}{3}\right) * \left(\frac{4}{5}\right) * \left(\frac{6}{5}\right) * \left(\frac{6}{7}\right) \dots$$

Escribir un programa que efectúe este cálculo con un número de términos especificados por el usuario.

- 6.3. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de N números. El valor de N se solicitará al principio del programa y los números serán introducidos por el usuario.
- 6.4. Escribir un programa que determina y escriba la descomposición factorial de los números enteros comprendidos entre 1900 y 2000.
- 6.5. Escribir un programa que determine todos los años que son bisiestos en el siglo XXII. Un año es bisiesto si es múltiplo de 4 (1988), excepto los múltiplos de 100 que no son bisiestos salvo que a su vez también sean múltiplos de 400 (1800 no es bisiesto, 2000 sí).
- 6.6. Escribir un programa que visualice un cuadrado mágico de orden impar  $n$ , comprendido entre 3 y 11; el usuario elige el valor de  $n$ . Un cuadrado mágico se compone de números enteros comprendidos entre 1 y  $n^2$ . La suma de los números que figuran en cada línea, cada columna y cada diagonal son iguales. Un ejemplo es:

|   |   |   |
|---|---|---|
| 8 | 1 | 6 |
| 3 | 5 | 7 |
| 4 | 9 | 2 |

Un método de construcción del cuadrado consiste en situar el número 1 en el centro de la primera línea, el número siguiente en la casilla situada encima y a la derecha, y así sucesivamente. Es preciso considerar que el cuadrado se cierra sobre sí mismo: la línea encima de la primera es de hecho la última y la columna a la derecha de la última es la primera. Sin embargo, cuando la posición del número caiga en una casilla ocupada, se elige la casilla situada debajo del número que acaba de ser situado.

- 6.7. Escribir un programa que encuentre los tres primeros números perfectos pares y los tres primeros números perfectos impares.

Un número *perfecto* es un entero positivo, que es igual a la suma de todos los enteros

positivos (excluido el mismo) que son divisores del número. El primer número perfecto es 6, ya que los divisores de 6 son 1, 2, 3 y  $1 + 2 + 3 = 6$ .

- 6.8. El valor de  $e^x$  se puede aproximar por la suma

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Escribir un programa que tome un valor de  $x$  como entrada y visualice la suma para cada uno de los valores de 1 a 100.

- 6.9. El matemático italiano Leonardo Fibonacci propuso el siguiente problema. Suponiendo que un par de conejos tiene un par de crías cada mes y cada nueva pareja se hace fértil a la edad de un mes. Si se dispone de una pareja fértil y ninguno de los conejos muertos, ¿cuántas parejas habrá después de un año? Mejorar el problema calculando el número de meses necesarios para producir un número dado de parejas de conejos.
- 6.10. Para encontrar el máximo común divisor (*mcd*) de dos números se emplea el algoritmo de Euclides, que se puede describir así: Dados los enteros  $a$  y  $b$  ( $a > b$ ), se divide  $a$  por  $b$ , obteniendo el cociente  $q_1$  y el resto  $r_1$ . Si  $r_1 < > 0$ , se divide  $r$  por  $b_1$ , Obteniendo el cociente  $q_2$  y el resto  $r_2$ . Si  $r_2 < > 0$ , se divide  $r_1$  por  $r_2$ , para obtener  $q_3$  y  $r_3$ , y así sucesivamente. Se continúa el proceso hasta que se obtiene un resto 0. El resto anterior es entonces el mcd de los números  $a$  y  $b$ . Escribir un programa que calcule el mcd de dos números.
- 6.11. Escribir un programa que encuentre el primer número primo introducido por teclado.
- 6.12. Calcular la suma de la serie  $1/1 + 1/2 + \dots + 1/N$  donde  $N$  es un número que se introduce por teclado.
- 6.13. Calcular la suma de los términos de la serie:  $1/2 + 2/2^2 + 3/2^3 + \dots + n/2^n$
- 6.14. Un número perfecto es aquel número que es igual a la suma de todas sus divisiones excepto el mismo. El primer número perfecto es 6, ya

que  $1 + 2 + 3 = 6$ . Escribir un programa que muestre todos los números perfectos hasta un número dado leído del teclado.

- 6.15. Encontrar un número natural  $N$  más pequeño tal que la suma de los  $N$  primeros números exceda de una cantidad introducida por el teclado.
- 6.16. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de  $N$  números. El valor de  $N$  se solicitará al principio del programa y los números serán introducidos por el usuario.
- 6.17. Calcular el factorial de un número entero leído desde el teclado utilizando las sentencias `while`, `repeat` y `for`.
- 6.18. Encontrar el número mayor de una serie de números.
- 6.19. Calcular la media de las notas introducidas por teclado con un diálogo interactivo semejante al siguiente:

```

¿Cuántas notas? 20
Nota 1 : 7.50
Nota 2: 6.40
Nota 3: 4.20
Nota 4: 8.50
...
Nota 20: 9.50
Media de estas 20: 7.475

```

- 6.20. Determinar si un número dado leído del teclado es primo o no.
- 6.21. Calcular la suma de la serie  $1/1 + 1/2 + 1/N$  donde  $N$  es un número entero que se determina con la condición que  $1/N$  sea menor que un  $\epsilon$  prefijado (por ejemplo  $1.10^{-6}$ ).
- 6.22. Escribir un programa que calcule la suma de los 50 primeros números enteros.
- 6.23. Calcular la suma de una serie de números leídos del teclado.

6.24. Calcular la suma de los términos de la serie:  $1/2 - 2/2^2 + 3/2^3 - \dots + n/2^n$  para un valor dado de  $n$ .

- 6.25. Contar el número de enteros negativos introducidos en una línea.
- 6.26. Visualizar en pantalla una figura similar a la siguiente

```

*


```

siendo variable el número de líneas que se pueden introducir.

- 6.27. Escribir un programa para mostrar, mediante bucles, los código ascii de la letras mayúsculas y minúscula.
- 6.28. Encontrar el número natural  $N$  más pequeño tal que la suma de los  $N$  primeros números exceda de una cantidad introducida por el teclado.
- 6.29. Diseñar un programa que produzca la siguiente salida:

```

ZYXWVTSRQPONMLHJIHGFEDCBA
YXWVTSRQPONMLKJIHGFEDCBA
XWVTSRQPONMLKJIHGFEDCBA
WVTSRQPONMLKJIHGFEDCBA
VTSRQPONMLKJIHGFEDCBA
TSRQPONMLKJIHGFEDCBA
SRQPONMLKJIHGFEDCBA
RQPONMLKJIHGFEDCBA
QPONMLKJIHGFEDCBA
PONMLKJIHGFEDCBA
ONMLKJIHGFEDCBA
NMLKJIHGFEDCBA
MLKJIHGFEDCBA
LKJIHGFEDCBA
KJIHGFEDCBA
JIHGFEDCBA
IHGFEDCBA
HGFEDCBA
GFEDCBA

```

FEDCBA  
EDCBA  
DCBA  
CBA  
BA  
A

- 6.30. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de N números. El valor de N se solicitará al principio del programa y los números serán introducidos por el usuario.

- 6.31. Encontrar y mostrar todos los números de 4 cifras que cumplen la condición de que la suma de las cifras de orden impar es igual a la suma de las cifras de orden par.

- 6.32. Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras es igual al valor del número.

## 6.11. PROYECTOS DE PROGRAMACIÓN

- 6.1. Escribir un programa que visualice la siguiente salida.

```

1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1

```

- 6.2. Diseñar e implementar un programa que cuente el número de sus entradas que son positivos, negativos y cero.

- 6.3. Diseñar e implementar un programa que extraiga valores del flujo de entrada estándar y a continuación visualice el mayor y el menor de esos valores en el flujo de salida estándar. El programa debe visualizar mensajes de advertencias cuando no haya entradas.

- 6.4. Diseñar e implementar un programa que solicite al usuario una entrada como un dato tipo fecha y a continuación visualice el número del día correspondiente del año. Ejemplo, si la fecha es 30 12 1999, el número visualizado es 364.

- 6.5. Diseñar e implementar un programa que solicite a su usuario un valor no negativo n y visualice la siguiente salida:

```

1 2 3 n-1 n
1 2 3 n-1
...

```

1

- 6.6. Un carácter es un espacio en blanco si es un espacio (' '), una tabulación ('\t'), un carácter de nueva línea ('\n') o un avance de página ('\f'). Diseñar y construir un programa que cuente el número de espacios en blanco de la entrada de datos.

- 6.7. Escribir un programa que lea la altura desde la que cae un objeto, se imprima la velocidad y la altura a la que se encuentra cada segundo suponiendo caída libre.

- 6.8. Escribir un programa que convierta: a) centímetros a pulgadas; b) libras a kilogramos. El programa debe tener como entrada longitud y masa, terminará cuando se introduzcan ciertos valores clave.

- 6.9. Escribir un programa que lea el radio de una esfera y visualice su área y su volumen ( $A = 4\pi r^2$ ,  $V = 4/3\pi r^3$ ).
- 6.10. Escribir un programa que lea 3 enteros positivos día, mes y año y a continuación visualice la fecha que represente, el número de días, del mes y una frase que diga si el año es o no bisiesto. Ejemplo, 4/11/1999 debe visualizar 4 de noviembre de 1999. Ampliar el programa de modo que calcule la fecha correspondiente a 100 días más tarde.
- 6.11. Escribir y ejecutar un programa que invierta los dígitos de un entero positivo dado.
- 6.12. Implementar el algoritmo de *Euclides* que encuentra el máximo común divisor de dos números enteros y positivos.

*Algoritmo de Euclides de m y n*

El algoritmo transforma un par de enteros positivos ( $m, n$ ) en una par ( $d, 0$ ), dividiendo repetidamente el entero mayor por el menor y reemplazando el mayor con el resto. Cuando el resto es 0, el otro entero de la pareja será el máximo común divisor de la pareja original.

*Ejemplo mcd (532, 112)*

|           |     |    |    |             |
|-----------|-----|----|----|-------------|
|           | 4   | 1  | 3  | → cocientes |
| 532       | 112 | 84 | 28 |             |
| Restos 84 | 28  | 00 |    | mcd = 28    |

# CAPÍTULO 7

## FUNCIONES

---

### CONTENIDO

- 7.1. Concepto de función.
- 7.2. Estructura de **una** función.
- 7.3. Prototipos de las funciones.
- 7.4. Parámetros **de** una función.
- 7.5. Funciones en línea: macros.
- 7.6. Ámbito (alcance).
- 7.7. Clases de almacenamiento.
- 7.8. Concepto y **uso** de funciones de biblioteca.
- 7.9. Funciones **de** carácter.
- 7.10. Funciones numéricas.
- 7.11. Funciones de fecha y hora.
- 7.12. Funciones **de utilidad**.
- 7.13. Visibilidad de **una** función.
- 7.14. **Compilación** separada.
- 7.16. Variables registro (**register**).
- 7.16. Recursividad.
- 7.17. *Resumen*.
- 7.18. *Ejercicios*.
- 7.19. *Problemas*.

## INTRODUCCIÓN

Una función es un miniprograma dentro de un programa. Las funciones contienen varias sentencias bajo un solo nombre, que un programa puede utilizar una o más veces para ejecutar dichas sentencias. Las funciones ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables. En otros lenguajes como BASIC o ensamblador se denominan *subrutinas*; en Pascal, las funciones son equivalentes a *funciones* y *procedimientos*.

Este capítulo examina el papel (rol) de las funciones en un programa C. Las funciones existen de modo autónomo, cada una tiene su ámbito. Como ya conoce, cada programa C tiene al menos una función `main()`; sin embargo, cada programa C consta de muchas funciones en lugar de una función `main()` grande. La división del código en funciones hace que las mismas se puedan reutilizar en su programa y en otros programas. Después de que escriba, pruebe y depure su función, se puede utilizar nuevamente una y otra vez. Para reutilizar una función dentro de su programa, sólo se necesita llamar a la función.

Si se agrupan funciones en bibliotecas otros programas pueden reutilizar las funciones, por esa razón se puede ahorrar tiempo de desarrollo. Y dado que las bibliotecas contienen rutinas presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

La mayoría de los programadores no *construyen* bibliotecas, sino que, simplemente, las utilizan. Por ejemplo, cualquier compilador incluye más de quinientas funciones de biblioteca, que esencialmente pertenecen a la *biblioteca estándar ANSI* (American National Standards Institute). Dado que existen tantas funciones de bibliotecas, no siempre será fácil encontrar la función necesaria, más por la cantidad de funciones a consultar que por su contenido en sí. Por ello, es frecuente disponer del *manual de biblioteca de funciones* del compilador o algún libro que lo incluya.

La potencia real del lenguaje es proporcionada por la biblioteca de funciones. Por esta razón, será preciso conocer las pautas para localizar funciones de la biblioteca estándar y utilizarlas adecuadamente. En este capítulo aprenderá:

- Utilizar las funciones proporcionadas por la biblioteca estándar ANSI C, que incorporan todos los compiladores de C.
- Los grupos de funciones relacionadas entre sí y los archivos de cabecera en que están declarados.

Las funciones son una de las piedras angulares de la programación en C y un buen uso de todas las propiedades básicas ya expuestas, así como de las propiedades avanzadas de las funciones, le proporcionarán una potencia, a veces impensable, a sus programaciones. La compilación separada y la recursividad son propiedades cuyo conocimiento es esencial para un diseño eficiente de programas en numerosas aplicaciones.

---

## CONCEPTOS CLAVE

- Biblioteca de funciones,
- Compilación independiente.
- Función.
- Modularización.
- Parámetros de una función.
- Pasar parámetros por valor.
- Paso por referencia.
- Recursividad.
- Sentencia return.
- Subprograma.

## 7.1. CONCEPTO DE FUNCIÓN

C fue diseñado como un *lenguaje de programación estructurado*, también llamado *programación modular*. Por esta razón, para escribir un programa se divide éste en varios módulos, en lugar de uno solo largo. El programa se divide en muchos módulos (rutinas pequeñas denominadas *funciones*), que producen muchos beneficios: aislar mejor los problemas, escribir programas correctos más rápido y producir programas que son mucho más fáciles de mantener.

Así pues, un programa C se compone de varias funciones, cada una de las cuales realiza una tarea principal. Por ejemplo, si está escribiendo un programa que obtenga una lista de caracteres del teclado, los ordene alfabéticamente y los visualice a continuación en la pantalla, se pueden escribir todas estas tareas en un único gran programa (función `main()`).

```
int main()
{
 /* Código C para obtener una lista de caracteres */
 ...
 /* Código C para alfabeticar los caracteres */
 ...
 /* Código C para visualizar la lista por orden alfabético */
 ...
 return 0
}
```

Sin embargo, este método no es correcto. El mejor medio para escribir un programa es escribir funciones independientes para cada tarea que haga el programa. El mejor medio para escribir el citado programa sería el siguiente:

```
int main()
{
 obtenercaracteres(); /* Llamada a una función que obtiene los
 números */
 alfabeticar(); /* Llamada a la función que ordena
 alfabéticamente las letras */
 verletras(); /* Llamada a la función que visualiza
 letras en la pantalla */
 return 0; /* retorno al sistema */
}

int obtenercaracteres()
{
 /*
 * Código de C para obtener una lista de caracteres
 */
 return(0); /* Retorno a main() */
}

int alfabeticar()
{
 /*...
 * Código de C para alfabeticar los caracteres
 */
 return(0); /* Retorno a main() */
}

void verletras()
 /*...

```

```

 Código de C para visualizar lista alfabetizada
 */
 return /* Retorno a main() */
}

```

Cada función realiza una determinada tarea y cuando se ejecuta `return` se retorna al punto en que fue llamada por el programa o función principal.

### Consejo

Una buena regla para determinar la longitud de una función (número de líneas que contiene) es que no ocupe más longitud que el equivalente a una pantalla.

## 7.2. ESTRUCTURA DE UNA FUNCIÓN

Una función es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Las funciones permiten al programador un grado de abstracción en la resolución de un problema.

Las funciones en C no se pueden anidar. Esto significa que una función no se puede declarar dentro de otra función. La razón para esto es permitir un acceso muy eficiente a los datos. En C todas las funciones **son** externas o globales, es decir, pueden ser llamadas desde cualquier punto del programa.

La estructura de una función en C se muestra en la Figura 7.1.

```

tipo-de-retorno nombreFunción (listaDeParámetros)
{
 cuerpo de la función
 return expresión
}

```

*tipo-de-retorno* Tipo de valor devuelto por la función o la palabra reservada `void` si la función no devuelve ningún valor.

*nombreFunción* Identificador o nombre de la función.

*listaDeParámetros* Lista de declaraciones de los parámetros de la función separados por comas.

*expresión* valor que devuelve la función.

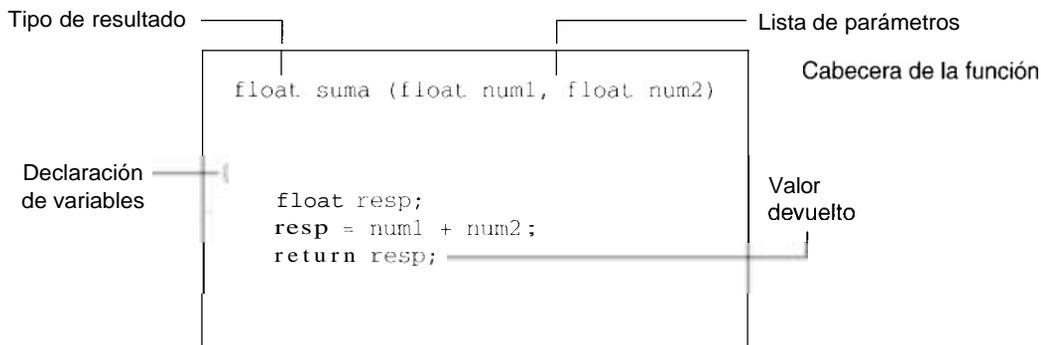


Figura 7.1. Estructura de una función.

Los aspectos más sobresalientes en el diseño de una función son:

- **Tipo de resultado.** Es el tipo de dato que devuelve la función C y aparece antes del nombre de la función.
- **Lista de parámetros.** Es una lista de parámetros *tipificados* (con tipos) que utilizan el formato siguiente:  
`tipo1 parámetro1, tipo2 parámetro2, ...`
- **Cuerpo de la función.** Se encierra entre llaves de apertura ( { ) y cierre ( } ). No hay punto y coma después de la llave de cierre.
- **Paso de parámetros.** Posteriormente se verá que el paso de parámetros en C se hace siempre por valor.
- **No se pueden declarar funciones anidadas.**
- **Declaración local.** Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- **Valor devuelto por la función.** Mediante la palabra reservada `return` se devuelve el valor de la función.

Una llamada a la función produce la ejecución de las sentencias del cuerpo de la función y un retorno a la unidad de programa llamadora después que la ejecución de la función se ha terminado, normalmente cuando se encuentra una sentencia `return`.

---

## Ejemplo 7.1

**Las funciones `cuadrado()` y `suma()` muestran dos ejemplos típicos de ellas.**

```

/* función que calcule los cuadrados de números enteros
 sucesivos a partir de un número dado (n), parámetro
 de la función y hasta obtener un cuadrado que sea
 mayor de 1000
*/
void cuadrado(int n)
{
 int cuadrado=0, q=0;
 while (q <= 1000) /*el cuadrado ha de ser menor de 1000 */
 {
 q = n*n;
 printf("El cuadrado de: %d es %d \n",n,q);
 n++;
 }
 return;
}
/*
 Calcula la suma de un número dado (parámetro) de elementos leídos de la
 entrada estándar (teclado).
*/
float suma (int num-elementos)
{
 int indice;
 float total = 0.0;

 printf("\n \t Introduce %d números reales\n",num_elementos);
 for (indice= 0; indice < num-elementos; indice++)
 {
 float x;
 scanf("%f",&x);
 }
}

```

```

 total += x;
}
return total;
}

```

### 7.2.1. Nombre de una función

Un nombre de una función comienza con una letra o un subrayado (`_`) y puede contener tantas letras, números o subrayados como desee. El compilador ignora, sin embargo, a partir de una cantidad dada (32 en Borland/Inprise C, 248 en Microsoft). C es sensible a mayúsculas, lo que significa que las letras mayúsculas y minúsculas son distintas a efectos del nombre de la función.

```

int max (int x, int y); /* nombre de la función max */
double media (double x1, double x2); /* nombre de la función media */
double MAX (int* m, int n); /* nombre de función MAX,
 distinta de max */

```

### 7.2.2. Tipo de dato de retorno

Si la función no devuelve un valor `int`, se debe especificar el tipo de dato devuelto (de retorno) por la función; cuando devuelve un valor `int`, se puede omitir ya que **por** defecto el C asume que todas las funciones son enteras, a pesar de ello siempre conviene especificar el tipo aun siendo de tipo `int`, para mejor legibilidad. El tipo debe ser uno de los tipos simples de C, tales como `int`, `char` o `float`, o un puntero a cualquier tipo C, o un tipo `struct`.

```

int max(int x, int y) /* devuelve un tipo int */
double media(double x1, double x2) /* devuelve un tipo double */
float func0() {...} /* devuelve un float */
char func1() {...} /* devuelve un dato char */
int *func3() {...} /* devuelve un puntero a int */
char *func4() {...} /* devuelve un puntero a char */
int func5() {...} /* devuelve un int (es opcional)*/

```

Si una función no devuelve un resultado, se puede utilizar el tipo `void`, que se considera como un tipo de dato especial. Algunas declaraciones de funciones que devuelven distintos tipos de resultados son:

```

int calculo_kilometraje(int litros, int kilometros);
char mayusculas(char car);
float DesvEst(void);
struct InfoPersona BuscarRegistro(int num_registro);

```

Muchas funciones no devuelven resultados. La razón es que se utilizan como *subrutinas* para realizar una tarea concreta. Una función que no devuelve un resultado, a veces se denomina *procedimiento*. Para indicar al compilador que una función no devuelve resultado, se utiliza el tipo de retorno `void`, como en este ejemplo:

```
void VisualizarResultados(float Total, int num-elementos);
```

Si se omite un tipo de retorno para una función, como en

```
VerResultados(float Total, int longitud);
```

el compilador supone que el tipo de dato devuelto es `int`. Aunque el uso de `int` es opcional, por razones de claridad y consistencia se recomienda su uso. Así, la función anterior se puede declarar también:

```
int VerResultados(float Total, int longitud);
```

### 7.2.3. Resultados de una función

Una función puede devolver un Único valor. El resultado se muestra con una sentencia `return` cuya sintaxis es:

```
return (expresión)
return;
```

El valor devuelto (*expresión*) puede ser cualquier tipo de dato excepto una función o un *array*. Se pueden devolver valores múltiples devolviendo un puntero o una estructura. El valor de retorno debe seguir las mismas reglas que se aplican a un operador de asignación. Por ejemplo, no se puede devolver un valor `int`, si el tipo de retorno es un puntero. Sin embargo, **si** se devuelve un `int` y el tipo de retorno es un `float`, se realiza la conversión automáticamente.

Una función puede tener cualquier número de sentencias `return`. Tan pronto como el programa encuentra cualquiera de las sentencias `return`, devuelve control a la sentencia llamadora. La ejecución de la función termina si no se encuentra ninguna sentencia `return`; en este caso, la ejecución continúa hasta la llave final del cuerpo de la función.

Si el tipo de retorno es `void`, la sentencia `return` se puede escribir como `return;` sin ninguna expresión de retorno, o bien, de modo alternativo se puede omitir la sentencia `return`.

```
void func1(void)
{
 puts("Esta función no devuelve valores");
}
```

El valor devuelto se suele encerrar entre paréntesis, pero su uso es opcional. En algunos sistemas operativos, como **DOS**, se puede devolver un resultado al entorno llamador. Normalmente el valor 0, se suele devolver en estos casos.

```
int main()
{
 puts("Prueba de un programa C, devuelve 0 al sistema ");
 return 0;
}
```

#### Consejo

Aunque no es obligatorio el uso de la sentencia `return` en la Última Línea, **se recomienda** su uso, ya que ayuda a recordar el retorno en ese punto a la función llamadora.

#### Precaución

Un error típico de programación es olvidar incluir la sentencia **return** o situarla dentro de una sección de código que no se ejecute. Si ninguna sentencia `return` se ejecuta, entonces el resultado que devuelve la función es impredecible y puede originar que su programa falle o produzca resultados incorrectos. Por ejemplo, suponga que se sitúa la sentencia `return` dentro de una sección de código que se ejecuta condicionalmente, tal como:

```

if (Total >= 0.0)
 return Total;

```

Si `Total` es menor que cero, no se ejecuta la sentencia `return` y el resultado de la función es un valor aleatorio (C puede generar el mensaje de advertencia "Function should return a value", que le ayudará a detectar este posible error).

#### 7.2.4. Llamada a una función

Las funciones, para poder ser ejecutadas, han de ser *llamadas* o *invocadas*. Cualquier expresión puede contener una *llamada a una función* que redirigirá el control del programa a la función nombrada. Normalmente la llamada a una función se realizará desde la función principal `main()`, aunque naturalmente también podrá ser desde otra función.

#### Nota

La función que llama a otra función se denomina *función llamadora* y la función controlada se denomina *función llamada*.

La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina (se alcanza la sentencia `return`, o la llave de cierre `}` si se omite `return`) el control del programa vuelve y retorna a la función `main()` o a la función llamadora si no es `main`.

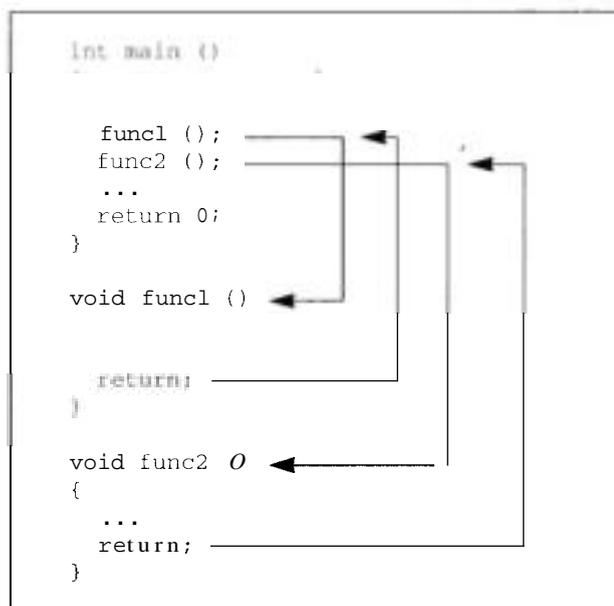


Figura 7.2. Traza de llamadas de funciones.

En el siguiente ejemplo se declaran dos funciones y se llaman desde la función `main()`.

```
#include <stdio.h>

void func1(void)
{
 puts ("Segunda función");
 return;
}

void func2(void)
{
 puts ("Tercera función");
 return;
}

int main()
{
 puts("Primera función llamada main()");
 func1(); /* Segunda función llamada */
 func2(); /* Tercera función llamada */
 puts ("main se termina");
 return 0; /* Devuelve control al sistema */
}
```

La salida de este programa es:

```
Primera función llamada main()
Segunda función
Tercera función
main se termina
```

Se puede llamar a una función y no utilizar el valor que se devuelve. En esta llamada a función: `func()`; el valor de retorno no se considera. El formato `func()` sin argumentos es el más simple. Para indicar que la llamada a una función no tiene argumentos se sitúa una palabra reservada `void` entre paréntesis en la declaración de la función y posteriormente en lo que se denominará *prototipo*; también, con paréntesis vacíos.

```
int main()

 func(); /* Llamada a la función */
 ...
}

void func(void) /* Declaración de la función */
{
 printf ("Función sin argumentos \n");
}
```

### **Precaución**

No se puede definir una función dentro de otra. Todo código de la función debe ser listado secuencialmente, a lo largo de todo el programa. Antes de que aparezca el código de una función, debe aparecer la llave de cierre de la función anterior.

---

**Ejemplo 7.2**

La función *max* devuelve el número mayor de dos enteros.

```

#include <stdio.h>
int max(int x, int y)
{
 if (x < y)
 return y;
 else
 return x;
}

int main()
{
 int m, n;
 do {
 scanf("%d %d", &m, &n);
 printf("Maximo de %d, %d es %d\n", max(m, n)); /*llamada a max*/
 }while(m != 0);
 return 0;
}

```

---



---

**Ejemplo 7.3**

Calcular la media aritmética de dos números reales.

```

#include <stdio.h>
double media(double x1, double x2)
{
 return(x1 + x2)/2;
}

int main()
{
 double num1, num2, med;
 printf("Introducir dos números reales:");
 scanf("%lf %lf", &num1, &num2);
 med = media(num1, num2);
 printf("El valor medio es %.4lf \n", med);
 return 0;
}

```

---

### 7.3. PROTOTIPOS DE LAS FUNCIONES

La *declaración* de una función se denomina *prototipo*. Los prototipos de una función contienen la cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma. Específicamente un prototipo consta de los siguientes elementos: nombre de la función, una lista de argumentos encerrados entre paréntesis y un punto y coma. En C no es estrictamente necesario que una función se declare o defina antes de su uso, no es necesario incluir el prototipo aunque si es recomendable para que el compilador pueda hacer chequeos en las llamadas a las funciones. Los prototipos de las funciones llamadas en un programa se incluyen en la cabecera del programa para que así sean reconocidas en todo el programa.

C recomienda que **se** declare una función si se llama a la función antes de que se defina.

### Sintaxis

|                                     |                                                                                                                                                                                              |
|-------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>tipo-retorno</i>                 | <i>nombre-función</i> ( <i>lista_de_declaración_parámetros</i> ) ;                                                                                                                           |
| <i>tipo-retorno</i>                 | Tipo del valor devuelto por la función o palabra reservada void si no devuelve un valor.                                                                                                     |
| <i>nombre-función</i>               | Nombre de la función.                                                                                                                                                                        |
| <i>lista_declaración_parámetros</i> | Lista de declaración de los parámetros de la función, separados por comas (los nombres de los parámetros son opcionales, pero es buena práctica incluirlos para indicar lo que representan). |

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función está siendo llamada correctamente, con respecto al número y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia.

```
double FahrACelsius(double tempFahr); /* prototipos válidos */
int max(int x, int y);
int longitud(int h, int a);
struct persona entrada(void);
char* concatenar(char* c1, char* c2);
double intensidad(double, double);
```

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la función `main()`. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los argumentos reales de la llamada a la función son los mismos que el número y tipo de argumentos formales en la función llamada. Si se detecta una inconsistencia, se visualiza un mensaje de error. Sin prototipos, un error puede ocurrir si un argumento con un tipo de dato incorrecto se pasa a una función. En programas complejos, este tipo de errores son difíciles de detectar.

En C, la diferencia entre los conceptos *declaración* y *definición* es preciso tenerla clara. Cuando una entidad se *declara*, se proporciona un nombre y se listan sus características. Una *definición* proporciona un nombre de entidad y reserva espacio de memoria para esa entidad. Una *definición* indica que existe un lugar en un programa donde «existe» realmente la entidad definida, mientras que una *declaración* es sólo una indicación de que algo existe en alguna posición.

Una *declaración* de la función contiene sólo la cabecera de la función y una vez declarada la función, la *definición* completa de la función debe existir en algún lugar del programa, antes o después de `main()`.

En el siguiente ejemplo se escribe una función `area()` de rectángulo. En la función `main()` se llama a `entrada()` para pedir la base y la altura; a continuación se llama a la función `area()`.

```
#include <stdio.h>

float area_rectangulo(float b, float a); /* declaración */
float entrada(); /* prototipo o declaración */

int main()
{
 float b, h;
 printf("\n Base del rectangulo: ");
 b = entrada();
 printf("\n Altura del rectangulo: ");
 h = entrada();
 printf("\n Area del rectangulo: %.2f", area_rectangulo(b,h));
}
```

```

 return 0;
}
/* devuelve número positivo */
float entrada()
{
 float m;
 do {
 scanf("%f",&m);
 } while (m<=0.0);
 return m;
}
/* calcula el area de un rectángulo */
float area_rectangulo(float b, float a)
{
 return (b*a);
}

```

En este otro ejemplo se declara la función media

```

#include <stdio.h>
double media (double x1, double x2); /*declaración de media*/

int main()
{
 ...
 med = media(num1, num2);
 ...
}

double media(double x1, double x2) /* definición */
{
 return (x1 + x2)/2;
}

```

- Declaraciones de una función

- Antes de **que una** función pueda ser invocada, debe ser **declarada**.
- Una declaración de una función contiene sólo la cabecera de la función (llamado también *prototipo*)

```

tipo_resultado nombre (tipo1 param1, tipo2 param2, ...);

```

- Los nombres de los parámetros se pueden omitir

```

char* copiar (char*, int);
char* copiar (char * buffer, int n);

```

La comprobación de tipos es una acción realizada por el compilador. El compilador conoce cuales son los tipos de argumentos que se han pasado una vez que se ha procesado un prototipo. Cuando se encuentra una sentencia de llamada a una función, el compilador confirma que el tipo de argumento en la llamada a la función es el mismo tipo que el del argumento correspondiente del prototipo. Si no son los mismos, el compilador genera un mensaje de error. Un ejemplo de prototipo:

```

int procesar(int a, char b, float c, double d, char *e);

```

El compilador utiliza sólo la información de los tipos de datos. Los nombres de los argumentos, aunque se aconsejan, no tienen significado; el propósito de los nombres es hacer la declaración de tipos más fácil para leer y escribir. La sentencia precedente se puede escribir también así:

```
int procesar(int, char, float, double, char *);
```

Si una función no tiene argumentos, se ha de utilizar la palabra reservada `void` como lista de argumentos del prototipo (también se puede escribir paréntesis vacíos).

```
int muestra(void);
```

## Ejemplos

```
1. /* prototipo de la función cuadrado */
double cuadrado(double);

int main()
{
 double x=11.5;
 printf("%6.2lf al cuadrado = %8.4lf \n",x,cuadrado(x));
 return 0;
}

double cuadrado(double n)
{
 return n*n;
}

2. /* prototipo de visualizar_nombre */
void visualizar_nombre(char*);

void main()
{
 visualizar_nombre("Lucas El Fuerte");
}

void visualizar_nombre(char* nom)
{
 printf("Hola %s \n",nom);
}
```

### 7.3.1. Prototipos con un número no especificado de parametros

Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por puntos suspensivos (...). Por ejemplo,

```
int muestras(int a, ...);
int printf(const char *formato, ...);
int scanf(const char *formato, ...);
```

Para implementar una función con lista variable de parámetros es necesario utilizar unas macros (especie de funciones en línea) que están definidas en el archivo de cabecera `stdarg.h`, por consiguiente lo primero que hay que hacer es incluir dicho archivo.

```
#include <stdarg.h>
```

En el archivo está declarado el tipo `va_list`, un puntero para manejar la lista de datos pasada a la función.

```
va_list puntero;
```

La función `va-start()` inicializa `puntero`, de tal forma que referencia al primer parámetro variable. El prototipo que tiene:

```
void va_start(va_list puntero, ultimofijo);
```

El segundo argumento es el último argumento fijo de la función que se está implementando. Así para la función `muestras(int a, ...)`:

```
va_start(puntero, a);
```

Con la función `va-arg()` se obtienen, consecutivamente, los sucesivos argumentos de la lista variable. El prototipo que tiene

```
tipo va_arg(va_list puntero, tipo);
```

Donde `tipo` es el tipo del argumento variable que es captado en ese momento, a su vez es el tipo de dato que devuelve `va-arg()`. Para la función `muestras()` si los argumentos variables son de tipo `int`:

```
int m;
m = va_arg(puntero, int);
```

La última llamada que hay que hacer en la implementación de estas funciones es a `va-end()`. De esta forma se queda el puntero preparado para siguientes llamadas. El prototipo que tiene `va-end()`:

```
void va_end(va_list puntero).
```

## Ejercicio 7.1

Una aplicación completa de una función con lista de argumentos variables es *maximo(int, ...)*, que calcula el máximo de *n* argumentos de tipo *double*, donde *n* es el argumento fijo que se utiliza.

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

void maximo(int n, ...);

int main(void)
{
 puts("\t\tPRIMERA BUSQUEDA DEL MAXIMO\n");
 maximo(6, 3.0, 4.0, -12.5, 1.2, 4.5, 6.4);
 puts("\n\t\tNUEVA BUSQUEDA DEL MAXIMO\n");
 maximo(4, 5.4, 17.8, 5.9, -17.99);
 return 0;
}

void maximo(int n, ...)
{
 double mx, actual;
 va_list puntero;
 int i;
 va_start(puntero, n);
 mx = actual = va_arg(puntero, double);
 printf("\t\tArgumento actual: %.2lf\n", actual);
 for (i=2; i<=n; i++)
 {
 actual = va_arg(puntero, double);
 printf("\t\tArgumento actual: %.2lf\n", actual);
 if (actual > mx)
 mx = actual;
 }
}
```

```

 mx = actual;
 }
}
printf("\t\tMáximo de la lista de %d números es %.2lf\n",n,mx);
va_end(puntero);
}

```

## 7.4. PARÁMETROS DE UNA FUNCIÓN

C siempre utiliza el método de *parámetros por valor* para pasar variables a funciones. Para que una función devuelva un valor a través de un argumento hay que pasar la dirección de la variable, y que el argumento correspondiente de la función sea un puntero, es la forma de conseguir en C un paso de *parámetro por referencia*. Esta sección examina el mecanismo que C utiliza para pasar parámetros a funciones y cómo optimizar el paso de parámetros, dependiendo del tipo de dato que se utiliza. Suponiendo que se tenga la declaración de una función `circulo` con tres argumentos

```
void circulo(int x, int y, int diametro);
```

Cuando se llama a `circulo` se deben pasar tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión, como en el siguiente ejemplo:

```
circulo(25, 40, vueltas*4);
```

### 7.4.1. Paso de parámetros por valor

*Paso por valor* (también llamado *paso por copia*) significa que cuando C compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro variable local, el cambio sólo afecta a la función y no tiene efecto fuera de ella.

La Figura 7.3 muestra la acción de pasar un argumento por valor. La variable real *i* no se pasa, pero el valor de *i*, 6, se pasa a la función receptora.

En la técnica de paso de parámetro por valor, la modificación de la variable (parámetro pasado) en la función receptora no afecta al parámetro argumento en la función llamadora.

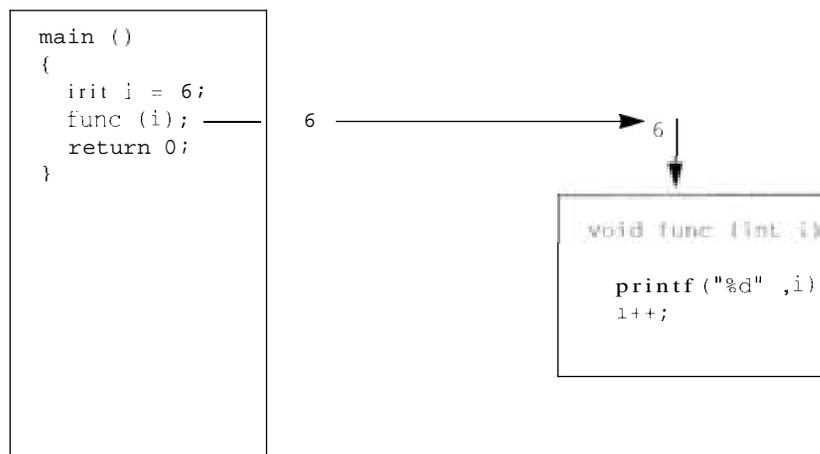


Figura 7.3. Paso de la variable *i* por valor.

## Nota

El método por defecto de pasar parámetros es por valor, a menos que se pasen arrays. Los arrays se pasan siempre por dirección.

El siguiente programa muestra el mecanismo de paso de parámetros por valor.

```

/*
 Muestra el paso de parámetros por valor
 Se puede cambiar la variable del parámetro en la función
 pero su modificación no puede salir al exterior
*/
#include <stdio.h>
void DemoLocal(int valor);
void main (void)
{
 int n = 10;
 printf("Antes de llamar a DemoLocal, n = %d\n",n);
 DemoLocal(n);
 printf("Después de llamada a DemoLocal, n = %d\n",n);
}
void DemoLocal(int valor)
{
 printf("Dentro de DemoLocal, valor = %d\n",valor);
 valor = 999;
 printf ("Dentro de DemoLocal, valor = %d\n",valor);
}

```

Al ejecutar este programa se visualiza la salida:

```

Antes de llamar a DemoLocal, n = 10
Dentro de DemoLocal, valor = 10
Dentro de DemoLocal, valor = 999
Después de llamar a DemoLocal, n = 10

```

### 7.4.2. Paso de parámetros por referencia

Cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por *referencia* o *dirección*.

En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (la variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado. Para pasar una variable por referencia, el símbolo & debe preceder al nombre de la variable y el parámetro variable correspondiente de la función debe declararse como puntero.

```

float x;
int y;
entrada(&x,&y);
. . .
void entrada(float* x, int* y)

```

C permite utilizar punteros para implementar parámetros por referencia, ya que por defecto en C el paso de parámetros es por valor.

```

/* método de paso por referencia, mediante punteros */
void intercambio(int* a, int* b)
{
 int aux = *a;
 *a = *b;
 *b = aux;
}

```

En la llamada siguiente, la función `intercambio()` utiliza las expresiones `*a` y `*b` para acceder a los enteros referenciados por las direcciones de las variables `i` y `j`:

```

int i = 3, j = 50;
printf("i = %d y j = %d \n", i, j);
intercambio(&i, &j);
printf("i = %d y j = %d \n", i, j);

```

La llamada a la función `intercambio()` debe pasar las direcciones de las variables intercambiadas. El operador `&` delante de una variable significa «dame la dirección de la variable».

```

double x;
&x ; /* dirección en memoria de x */

```

Una variable, o parámetro puntero se declara poniendo el asterisco (`*`) antes del nombre de la variable. Las variables `p`, `r` y `q` son punteros a distintos tipos.

```

char* p; /* variable puntero a char */
int * r; /* variable puntero a int */
double* q; /* variable puntero a double */

```

### 7.4.3. Diferencias entre paso de variables por valor y por referencia

Las reglas que se han de seguir cuando se transmiten variables por valor y por referencia son las siguientes:

- o los parámetros valor reciben copias de los valores de los argumentos que se les pasan;
- o la asignación a parámetros valor de una función nunca cambian el valor del argumento original pasado a los parámetros;
- o los parámetros para el paso por referencia (declarados con `*`, punteros) reciben la dirección de los argumentos pasados; a estos les debe de preceder del operador `&`, excepto los arrays;
- o en una función, las asignaciones a parámetros referencia (punteros) cambian los valores de los argumentos originales.

Por ejemplo, la escritura de una función `potrat()` para cambiar los contenidos de dos variables, requiere que los datos puedan ser modificados.

| <i>Paso por valor</i>                  | <i>Paso por referencia</i>               |
|----------------------------------------|------------------------------------------|
| <code>float a, b;</code>               | <code>float a, b;</code>                 |
| <code>potrat1(float x, float y)</code> | <code>potrat2(float* x, float* y)</code> |
| <code>{</code>                         | <code>{</code>                           |
| <code>}</code>                         | <code>}</code>                           |

Sólo en el caso de `potrat2` los valores de `a` y `b` se cambiarán. Veamos una aplicación completa de ambas funciones:

```

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

```

```

void potrat1(float, float);
void potrat2(float*, float*)

void main()
{
 float a, b;
 a = 5.0; b = 1.0e2;

 potrat1(a, b);
 printf("\n a = %.1f b = %.1f", a, b);
 potrat2(a, b);
 printf("\n a = %.1f b = %.1f", a, b);
}

void potrat1(float x, float y)
{
 x = x*x;
 y = sqrt(y);
}

void potrat2(float* x, float* y)
{
 *x = (*x)*(*x);
 *y = sqrt(*y);
}

```

La ejecución del programa producirá:

```

a = 5.0 b = 100.0
a = 25.0 b = 10.0

```

## Nota

Todos los parámetros en C se pasan por valor. C no tiene parámetros por referencia, hay que hacerlo con punteros y el operador &.

Se puede observar en el programa cómo se accede a los punteros, el operador \* precediendo al parámetro puntero devuelve el contenido.

### 7.4.4. Parámetros const de una función

Con el objeto de añadir seguridad adicional a las funciones, se puede añadir a una descripción de un parámetro el especificador `const`, que indica al compilador que sólo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación.

```

void f1(const int, const int*);
void f2(int, int const*);

void f1(const int x, const int* y)
{
 x = 10; /* error por cambiar un objeto constante*/
 y = 11; / error por cambiar un objeto constante*/
 y = &x; /* correcto */
}

void f2(int x, int const* y)
{
 x = 10; /* correcto */
 y = 11; / error */
 y = &x; /* correcto */
}

```

La Tabla 7.1 muestra un resumen del comportamiento de los diferentes tipos de parámetros.

Tabla 7.1. Paso de parámetros en C.

| Parámetro especificado como: | Item pasado por | Cambia item dentro de la función | Modifica parámetros al exterior |
|------------------------------|-----------------|----------------------------------|---------------------------------|
| int item                     | valor           | Si                               | NO                              |
| const int item               | valor           | NO                               | NO                              |
| int* item                    | por dirección   | Si                               | Si                              |
| const int* item              | por dirección   | No su contenido                  | NO                              |

## 7.5. FUNCIONES EN LINEA, MACROS CON ARGUMENTOS

Una función normal es un bloque de código que se llama desde otra función. El compilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. A continuación, el compilador genera código que sitúa cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

```
float fesp(float x)
{
 return (x*x + 2*x -1);
}
```

Las funciones en línea sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función es una expresión, **su** código es pequeño y se utiliza muchas veces en el programa. Realmente no son funciones, el preprocesador expande o sustituye la expresión cada vez que es llamada. Así la anterior función puede sustituirse:

```
#define fesp(x) (x*x + 2*x -1)
```

En este programa se realizan cálculos de la función para valores de x en un intervalo.

```
#include <stdio.h>
#define fesp(x) (x*x + 2*x -1)

void main()
{
 float x;
 for (x = 0.0; x <=6.5; x += 0.3)
 printf("\t f(%.1f) = %6.2f ", x, fesp(x));
}
```

Antes de que el compilador construya el código ejecutable de este programa, el preprocesador sustituye toda llamada a `fexp(x)` por la expresión asociada. Realmente es como si hubiéramos escrito

```
printf("\t f(%.1f) = %6.2f ", x, (x*x + 2*x -1));
```

Para una *macro con argumentos (función en línea)*, el compilador inserta realmente el código en el punto en que se llama, esta acción hace que el programa se ejecute más rápidamente, ya que no ha de ejecutar el código asociado con la llamada a la función.

Sin embargo, cada invocación a una macro puede requerir tanta memoria como se requiera para contener la expresión completa que representa. Por esta razón, el programa incrementa su tamaño, aunque es mucho más rápido en su ejecución. Si se llama a una macro diez veces en un programa, el compilador inserta diez copias de ella en el programa. Si la macrofunción ocupa 0.1K, el tamaño de su programa se incrementa en 1K (1024 bytes). Por el contrario, si se llama diez veces a la misma función

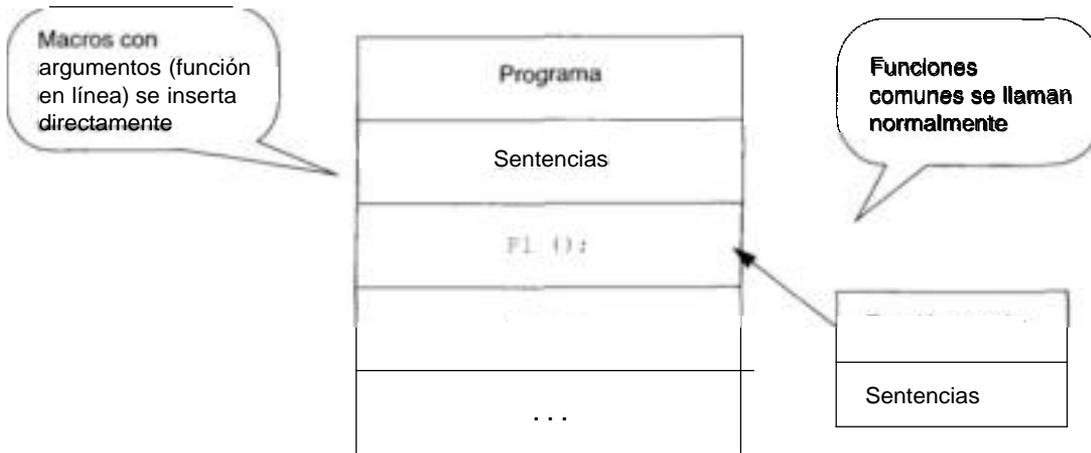


Figura 7.4. Código generado por una función fuera de línea.

con una función normal, y el código de llamada suplementario es 25 bytes por cada llamada, el tamaño se incrementa en una cantidad insignificante.

La Figura 7.5 ilustra la sintaxis general de una macro con argumentos.

```
#define NombreMacro(parámetros sin tipos) expresión-texto
```

**REGLA:** La definición de una macro sólo puede ocupar una línea. Se puede prolongar la línea con el carácter \ al final de la línea.

Figura 7.5. Código de una macro con argumentos.

La Tabla 7.2 resume las ventajas y desventajas de situar un código de una función en una macro o fuera de línea (función normal):

Tabla 7.2. Ventajas y desventajas de macros.

|                                    | Ventajas                  | Desventajas              |
|------------------------------------|---------------------------|--------------------------|
| <b>Macros (funciones en línea)</b> | Rápida de ejecutar.       | Tamaño de código grande. |
| <b>Funciones fuera de línea</b>    | Pequeño tamaño de código. | Lenta de ejecución.      |

### 7.5.1. Creación de macros con argumentos

Para crear una macro con argumentos utilizar la sintaxis:

```
#define NombreMacro(parámetros sin tipos) expresión-texto
```

La definición ocupará sólo una línea, aunque si se necesitan más texto, situar una barra invertida (\) al final de la primera línea y continuar en la siguiente, en caso de ser necesarias más líneas proceder de igual forma; de esa forma se puede formar una expresión más compleja. Entre el nombre de la macro y los paréntesis de la lista de argumentos no puede haber espacios en blanco. Por ejemplo, la función media de tres valores se puede escribir:

```
#define MEDIA3(x,y,z) ((x) + (y) + (z))/3.0
```

En este segmento de código se invoca a MEDIA3

```
double a = 2.9;
printf("\t %lf ", MEDIA3(a,4.5,7));
```

En esta llamada a MEDIA3 se pasan argumentos de tipo distinto. Es importante tener en cuenta que en las macros con argumentos *no hay comprobación de tipos*. Para evitar problemas de prioridad de operadores, es conveniente encerrar entre paréntesis cada argumento en la expresión de definición e incluso encerrar entre paréntesis toda la expresión.

En la siguiente macro, la definición de la expresión ocupa más de una línea.

```
#define FUNCION3(x) {
 if ((x) <-1.0) \
 (-(x)*(x) +3); \
 else if ((x)<=1) \
 (2*(x)+5); \
 else \
 ((x)*(x)-5); \
 }
```

Al tener la macro más de una sentencia, encerrarla entre llaves hace que sea una sola sentencia, aunque sea compuesta.

## Ejercicio 7.2

Una aplicación completa de una macro con argumentos es `VolCono()`, que calcula el volumen de la figura geométrica *Cono*.

$$(V = \frac{1}{3} \pi r^2 h)$$

```
#include <stdio.h>
#define Pi 3.141592

#define VOLCONO(radio,altura) ((Pi*(radio*radio)*altura)/3.0)

int main()
{
 float radio, altura, volumen;

 printf("\nIntroduzca radio del cono: ");
 scanf("%f",&radio);
 printf("Introduzca altura del cono: ");
 scanf("%f",&altura);
 volumen = VOLCONO(radio, altura);
 printf("\nEl volumen del cono es: %.2f",volumen);
 return 0;
}
```

## 7.6. ÁMBITO (ALCANCE)

El *ámbito* o *alcance* de una variable determina cuáles son las funciones que reconocen ciertas variables. Si una función reconoce una variable, la variable es *visible* en esa función. El ámbito es la zona de un programa en la que es visible una variable. Existen cuatro tipos de ámbitos: *programa*, *archivo/fuente*, *función* y *bloque*. Se puede designar una variable para que esté asociada a uno de estos ámbitos. Tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito.

Normalmente la posición de la sentencia en el programa determina el ámbito. Los especificadores de clases de almacenamiento, `static`, `extern`, `auto` y `register`, pueden afectar al ámbito. El siguiente fragmento de programa ilustra cada tipo de ámbito:

```
int i; /* Ámbito de programa */
static int j; /* Ámbito de archivo */
float func(int k) /* k, ámbito de función */
{
 int m; /* Ámbito de bloque */
 ...
}
```

### 7.6.1. Ámbito del programa

Las variables que tienen *ámbito de programa* pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman *variables globales*. Para hacer una variable global, declárela simplemente al principio de un programa, fuera de cualquier función.

```
int g, h; /* variables globales */
main()
{
 ...
}
```

Una variable global es visible («se conocen») desde su punto de definición en el archivo fuente. Es decir, si se define una variable global, cualquier línea del resto del programa, no importa cuantas funciones y líneas de código le sigan, podrá utilizar esa variable.

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

float ventas, beneficios; /* variables globales */
void f3(void)

}

void f1(void)
{
 ...
}

void main()
{
 ...
}
```

### Consejo

**Declare todas las variables en la parte superior de su programa. Aunque se pueden definir tales variables entre dos funciones, podría realizar cualquier cambio en su programa de modo más rápido, si sitúa las variables globales al principio del programa.**

### 7.6.2. Ámbito del archivo fuente

Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada `static` tiene *ámbito de archivo fuente*. Las variables con este ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente. Si un archivo fuente tiene más de una función, todas las funciones que siguen a la declaración de la variable pueden referenciarla. En el ejemplo siguiente, `i` tiene ámbito de archivo fuente:

```
static int i;
void func(void)
{
}
}
```

### 7.6.3. Ámbito de una función

Una variable que tiene ámbito de una función se puede referenciar desde cualquier parte de la función. Las variables declaradas dentro del cuerpo de la función se dice que son *locales* a la función. Las variables locales no se pueden utilizar fuera del ámbito de la función en que están definidas.

```
void calculo(void)
{
 double x, r, t ; /* Ámbito de la función */
 ...
}
```

### 7.6.4. Ámbito de bloque

Una variable declarada en un bloque tiene *ámbito de bloque* y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declaradas dentro de una función tienen ámbito de bloque de la función; no son visibles fuera del bloque. En el siguiente ejemplo, `i` es una variable local:

```
void funcl(int x)
{
 int i;
 for (i = x; i < x+10; i++)
 printf("i = %d \n", i*i);
}
```

Una variable local declarada en un bloque anidado sólo es visible en el interior de ese bloque.

```
float func(int j)
{
 if (j > 3)
 {
 int i;
 for (i = 0; i < 20; i++)
 funcl(i);
 }
 /* aquí ya no es visible i */
};
```

## 7.6.5. Variables locales

Además de tener un ámbito restringido, las variables locales son especiales por otra razón: existen en memoria sólo cuando la función está activa (es decir, mientras se ejecutan las sentencias de la función). Cuando la función no se está ejecutando, sus variables locales no ocupan espacio en memoria, ya que no existen. Algunas reglas que siguen las variables locales son:

- Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir la misma variable `test`. Cada variable es distinta y pertenece a su función específica.
- Las variables locales de las funciones no existen en memoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden compartir la misma memoria para sus variables locales (pero no al mismo tiempo).

## 7.7. CLASES DE ALMACENAMIENTO

Los especificadores de clases (tipos) de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes: `auto`, `extern`, `register`, `static` y `typedef`.

### 7.7.1. Variables automáticas

Las variables que se declaran dentro de una función se dice que son automáticas (`auto`), significando que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada `auto` es opcional.

```
auto int Total; es igual que int Total;
```

Normalmente no se especifica la palabra `auto`.

### 7.7.2. Variables externas

A veces se presenta el problema de que una función necesita utilizar una variable que *otra función* inicializa. Como las variables locales sólo existen temporalmente mientras se está ejecutando su función, no pueden resolver el problema. ¿Cómo se puede resolver entonces el problema? En esencia, de lo que se trata es de que una función de un archivo de código fuente utilice una variable definida en otro archivo. Una solución es declarar la variable local con la palabra reservada `extern`. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro lugar.

```
/* variables externas: parte 1 */
/* archivo fuente exter1.c */
#include <stdio.h>

extern void leerReal(void); /* función definida en otro archivo; en este
 caso no es necesario extern */

float f;

int main()
{
 leerReal();
 printf("Valor de f = %f", f);
 return 0;
}
```

```

/*variables externas: parte 2 */
/* archivo fuente exter2.c */
#include <stdio.h>

void leerReal(void)
{
 extern float f;

 printf("Introduzca valor en coma flotante: ");
 scanf("%f",&f);
}

```

En el archivo EXTER2.C la declaración externa de *f* indica al compilador que *f* se ha definido en otra parte (archivo). Posteriormente, cuando estos archivos se enlacen, las declaraciones se combinan de modo que se referirán a las mismas posiciones de memoria.

### 7.7.3. Variables registro

Otro tipo de variable C es la *variable registro*. Precediendo a la declaración de una variable con la palabra reservada *register*, se sugiere al compilador que la variable se almacene en uno de los registros hardware del microprocesador. La palabra *register* es una sugerencia al compilador y no una orden. La familia de microprocesadores 80x86 no tiene muchos registros hardware de reserva, por lo que el compilador puede decidir ignorar sus sugerencias. Para declarar una variable registro, utilice una declaración similar a:

```
register int k;
```

Una variable registro debe ser local a una función, nunca puede ser global al programa completo.

El uso de la variable *register* no garantiza que un valor se almacene en un registro. Esto sólo sucederá si existe un registro disponible. Si no existen registros suficientes, C ignora la palabra reservada *register* y crea la variable localmente como ya se conoce.

Una aplicación típica de una variable registro es como variable de control de un bucle. Guardando la variable de control de un bucle en un registro, se reduce el tiempo que la CPU requiere para buscar el valor de la variable de la memoria. Por ejemplo,

```
register int indice;
for (indice = 0; indice < 1000; indice++)...
```

### 7.7.4. Variables estáticas

Las variables estáticas son opuestas, en su significado, a las variables automáticas. Las *variables estáticas* no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable *static* se inicializa sólo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada *static*.

```

func_uno()
{
 int i;
 static int j = 25; /*j, k variables estáticas */
 static int k = 100;
 ...
}

```

Las variables estáticas se utilizan normalmente para mantener valores entre llamadas a funciones.

```
float ResultadosTotales(float valor)
{
 static float suma;

 suma = suma + valor;
 return suma;
}
```

En la función anterior se utiliza `suma` para acumular sumas a través de sucesivas llamadas a `ResultadosTotales`.

---

### Ejercicio 7.3

Una aplicación de una variable *static* en una función es la que nos permite obtener la serie de números de fibonacci. El ejercicio lo planteamos: dado un entero  $n$ , obtener los  $n$  primeros números de la serie de fibonacci.

#### Análisis

La secuencia de números de fibonacci: 0, 1, 1, 2, 3, 5, 8, 13..., se obtiene partiendo de los números 0, 1 y a partir de ellos cada número se obtiene sumando los dos anteriores:

$$a_n = a_{n-1} + a_{n-2}$$

La función `fibonacci` tiene dos variables estáticas,  $x$  e  $y$ . Se inicializan  $x$  a 0 e  $y$  a 1; a partir de esos valores se calcula el valor actual,  $y$ , se deja preparado  $x$  para la siguiente llamada. Al ser variables estáticas mantienen el valor entre llamada y llamada.

```
#include <stdio.h>
long int fibonacci();
int main()
{
 int n,i;
 printf("\nCuantos numeros de fibonacci ?: ");
 scanf("%d",&n);
 printf("\nSecuencia de fibonacci: 0,1");
 for (i=2; i<n; i++)
 printf(",%ld",fibonacci());
 return 0;
}
long int fibonacci()
{
 static int x = 0;
 static int y = 1;
 y = y + x;
 x = y - x;
 return y;
}
```

---

### Ejecución

Cuantos numeros de fibonacci ? 14

Secuencia de fibonacci: 0,1,1,2,3,5,8,13,21,34,55,89,144,233

## 7.8. CONCEPTO Y USO DE FUNCIONES DE BIBLIOTECA

Todas las versiones del lenguaje C ofrecen con una biblioteca estándar de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las *funciones estándar o predefinidas*, como así se denominan las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo *archivo de cabecera*.

Los nombres de los archivos de cabecera estándar utilizados en nuestro programa se muestran a continuación encerrados entre corchetes tipo ángulo:

|                               |                               |                               |                               |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| <code>&lt;assert.h&gt;</code> | <code>&lt;ctype.h&gt;</code>  | <code>&lt;errno.h&gt;</code>  | <code>&lt;float.h&gt;</code>  |
| <code>&lt;limits.h&gt;</code> | <code>&lt;math.h&gt;</code>   | <code>&lt;setjmp.h&gt;</code> | <code>&lt;signal.h&gt;</code> |
| <code>&lt;stdarg.h&gt;</code> | <code>&lt;stddef.h&gt;</code> | <code>&lt;stdio.h&gt;</code>  | <code>&lt;string.h&gt;</code> |
| <code>&lt;time.h&gt;</code>   |                               |                               |                               |

En los módulos de programa se pueden incluir líneas `#include` con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez.

Para utilizar una función o un macro, se debe conocer su número de argumentos, sus tipos y el tipo de sus valores de retorno. Esta información se proporcionará en los prototipos de la función. La sentencia `#include` mezcla el archivo de cabecera en su programa.

Algunos de los grupos de funciones de biblioteca más usuales son:

- o E/S estándar (para operaciones de Entrada/Salida);
- o matemáticas (para operaciones matemáticas);
- o rutinas estándar (para operaciones estándar de programas);
- o visualizar ventana de texto;
- o de conversión (rutinas de conversión de caracteres y cadenas);
- o de diagnóstico (proporcionan rutinas de depuración incorporada);
- o de manipulación de memoria;
- o control del proceso;
- o clasificación (ordenación);
- o directorios;
- o fecha y hora;
- o de interfaz;
- o diversas;
- o búsqueda;
- o manipulación de cadenas;
- o gráficos.

Se pueden incluir tantos archivos de cabecera como sean necesarios en sus archivos de programa, incluyendo sus propios archivos de cabecera que definen sus propias funciones.

En este capítulo se estudiarán las funciones más sobresalientes y más utilizadas en programación.

## 7.9. FUNCIONES DE CARÁCTER

El archivo de cabecera `<CTYPE.H>` define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones devuelven un resultado de valor verdadero (distinto de cero) o falso (cero).

Para utilizar cualquiera de las funciones (Tabla 7.3) no se olvide incluir el archivo de cabecera `CTYPE.H` en la parte superior de cualquier programa que haga uso de esas funciones.

Tabla 7.3. Funciones de caracteres.

| Función                          | Prueba (test) de                                                                           |
|----------------------------------|--------------------------------------------------------------------------------------------|
| <code>int isalpha(int c)</code>  | Letra mayúscula o minúscula.                                                               |
| <code>int isdigit(int c)</code>  | Dígito decimal.                                                                            |
| <code>int isupper(int c)</code>  | Letra mayúscula (A-Z).                                                                     |
| <code>int islower(int c)</code>  | Letra minúscula (a-z).                                                                     |
| <code>int isalnum(int c)</code>  | letra o dígito; <code>isalpha(c)    isdigit(c)</code>                                      |
| <code>int iscntrl(int c)</code>  | Carácter de control.                                                                       |
| <code>int isxdigit(int c)</code> | Dígito hexadecimal.                                                                        |
| <code>int isprint(int c)</code>  | Carácter imprimible incluyendo ESPACIO.                                                    |
| <code>int isgraph(int c)</code>  | Carácter imprimible excepto ESPACIO.                                                       |
| <code>int isspace(int c)</code>  | ESPACIO, AVANCE DE PÁGINA, NUEVA LÍNEA, RETORNO DE CARRO, TABULACIÓN, TABULACIÓN VERTICAL. |
| <code>int ispunct(int c)</code>  | Carácter imprimible no espacio, dígito o letra.                                            |
| <code>int toupper(int c)</code>  | Convierte a letras mayúsculas.                                                             |
| <code>int tolower(int c)</code>  | Convierte a letras minúsculas.                                                             |

### 7.9.1. Comprobación alfabética y de dígitos

Existen varias funciones que sirven para comprobar condiciones alfabéticas:

- **isalpha(c)**  
Devuelve verdadero (distinto de cero) si *c* es una letra mayúscula o minúscula. Se devuelve un valor falso si se pasa un carácter distinto de letra a esta función.
- **islower(c)**  
Devuelve verdadero (distinto de cero) si *c* es una letra minúscula. Se devuelve un valor falso (0), si se pasa un carácter distinto de una minúscula.
- **isupper(c)**  
Devuelve verdadero (distinto de cero) si *c* es una letra mayúscula, falso con cualquier otro carácter.

Las siguientes funciones comprueban caracteres numéricos:

- **isdigit(c)**  
Comprueba si *c* es un dígito de 0 a 9, devolviendo verdadero (distinto de cero) en ese caso, y falso en caso contrario.
- **isxdigit(c)**  
Devuelve verdadero si *c* es cualquier dígito hexadecimal (0 a 9, A a F, o bien a a f) y falso en cualquier otro caso.

Las siguientes funciones comprueban argumentos numéricos o alfabéticos:

- **isalnum(c)**  
Devuelve un valor verdadero, si *c* es un dígito de 0 a 9 o un carácter alfabético (bien mayúscula o minúscula) y falso en cualquier otro caso.

### Ejemplo 7.4

Leer un carácter del teclado y comprobar **si** es una letra.

```

/*
 Solicita iniciales y comprueba que es alfabética
*/
#include <stdio.h>
#include <ctype.h>
int main()
{
 char inicial;
 printf("¿Cuál es su primer carácter inicial? ");
 scanf("%c",&inicial);
 while (!isalpha(inicial))
 {
 puts("Carácter no alfabético ");
 printf("¿Cuál es su siguiente inicial? ");
 scanf("%c",&inicial);
 }
 puts(";Terminado!");
 return 0;
}

```

### 7.9.2. Funciones de prueba de caracteres especiales

Algunas funciones incorporadas a la biblioteca de funciones comprueban caracteres especiales, principalmente a efectos de legibilidad. Estas funciones son las siguientes:

- **isctr1(c)**  
Devuelve verdadero si *c* es un *carácter de control* (códigos ASCII 0 a 31) y falso en caso contrario.
- **isgraph(c)**  
Devuelve verdadero si *c* es un carácter imprimible (no de control) excepto espacio; en caso contrario, se devuelve falso.
- **isprint(c)**  
Devuelve verdadero si *c* es un carácter imprimible (código ASCII 32 a 127) incluyendo un espacio; en caso contrario, se devuelve falso.
- **ispunct(c)**  
Devuelve verdadero si *c* es cualquier carácter de puntuación (un carácter imprimible distinto de espacio, letra o dígito); falso, en caso contrario.
- **isspace(c)**  
Devuelve verdadero si *c* es carácter un espacio, nueva línea (`\n`), retorno de carro (`\r`), tabulación (`\t`) o tabulación vertical (`\v`).

### 7.9.3. Funciones de conversión de caracteres

Existen funciones que sirven para cambiar caracteres mayúsculas a minúsculas o viceversa.

- **tolower(c)**  
Convierte el carácter *c* a minúscula, si ya no lo es.
- **toupper(c)**  
Convierte el carácter *c* a mayúscula, si ya no lo es.

### Ejemplo 7.5

El programa MAYMIN1 .C comprueba si la entrada es una v o una H.

```

#include <stdio.h>
#include <ctype.h>

int main()
{
 char resp; /* respuesta del usuario */
 char c;

 printf("¿Es un varón o una hembra (V/H)? : ");
 scanf("%c",&resp);
 resp=toupper(resp);
 switch (resp)
 {
 case 'V':
 puts("Es un enfermero");
 break;
 case 'H':
 puts("Es una maestra");
 break;
 default:
 puts("No es ni enfermero ni maestra");
 break;
 }
 return 0;
}

```

## 7.10. FUNCIONES NUMÉRICAS

Virtualmente cualquier operación aritmética es posible en un programa C. Las funciones matemáticas disponibles son las siguientes:

- o matemáticas;
- o trigonométricas;
- o logarítmicas;
- o exponenciales;
- o aleatorias.

La mayoría de las funciones numéricas están en el archivo de cabecera MATH.H; las funciones **abs** y **labs** están definidas en MATH.H y STDLIB.H, y las rutinas **div** y **ldiv** en STDLIB.H.

### 7.10.1. Funciones matemáticas

Las funciones matemáticas usuales en la biblioteca estándar son:

- **ceil(x)**  
Redondea al entero más cercano.
- **fabs(x)**  
Devuelve el valor absoluto de x (un valor positivo).
- **floor(x)**  
Redondea por defecto al entero más próximo.

- **fmod(x, y)**  
Calcula el resto  $f$  en coma flotante para la división  $x/y$ , de modo que  $x = i*y+f$ , donde  $i$  es un entero,  $f$  tiene el mismo signo que  $x$  y el valor absoluto de  $f$  es menor que el valor absoluto de  $y$ .
- **pow(x, y)**  
Calcula  $x$  elevado a la potencia  $y$  ( $x^y$ ). Si  $x$  es menor que o igual a cero,  $y$  debe ser un entero. Si  $x$  es igual a cero,  $y$  no puede ser negativo.
- **pow10(x)**  
Calcula 10 elevado a la potencia  $x$  ( $10^x$ );  $x$  debe ser de tipo entero.
- **sqrt(x)**  
Devuelve la raíz cuadrada de  $x$ ;  $x$  debe ser mayor o igual a cero.

### 7.10.2. Funciones trigonométricas

La biblioteca de C incluye una serie de funciones que sirven para realizar cálculos trigonométricos. Es necesario incluir en su programa el archivo de cabecera `MATH.H` para utilizar cualquier función.

- **acos(x)**  
Calcula el arco coseno del argumento  $x$ . El argumento  $x$  debe estar entre  $-1$  y  $1$ .
- **asin(x)**  
Calcula el arco seno del argumento  $x$ . El argumento  $x$  debe estar entre  $-1$  y  $1$ .
- **atan(x)**  
Calcula el arco tangente del argumento  $x$ .
- **atan2(x, y)**  
Calcula el arco tangente de  $x$  dividido por  $y$ .
- **cos(x)**  
Calcula el coseno del ángulo  $x$ ;  $x$  se expresa en radianes.
- **sin(x)**  
Calcula el seno del ángulo  $x$ ;  $x$  se expresa en radianes.
- **tan(x)**  
Devuelve la tangente del ángulo  $x$ ;  $x$  se expresa en radianes.

#### **Regla**

**Si necesita pasar un ángulo expresado en grados a radianes, para poder utilizarlo con las funciones trigonométricas, multiplique los grados por  $\pi/180$ , donde  $\pi = 3.14159$ .**

### 7.10.3. Funciones logarítmicas y exponenciales

Las funciones logarítmicas y exponenciales suelen ser utilizadas con frecuencia no sólo en matemáticas, sino también en el mundo de la empresa y los negocios. Estas funciones requieren también el archivo de inclusión `MATH.H`.

- **exp(x), expl(x)**  
Calcula el exponencial  $e^x$ , donde  $e$  es la base de logaritmos naturales de valor 2.718282.  
`valor = exp(5.0);`  
Una variante de esta función es `expl`, que calcula  $e^x$  utilizando un valor `long double` (largo doble).

- **log(x), logl(x)**

La función **log** calcula el logaritmo natural del argumento **x** y **logl(x)** calcula el citado logaritmo natural del argumento **x** de valor `long double` (largo doble).

- **log10(x), log10l(x)**

Calcula el logaritmo decimal del argumento **x**, de valor real `double` en **log10(x)** y de valor real `long double` en **log10l(x)**; **x** ha de ser positivo.

#### 7.10.4. Funciones aleatorias

Los números aleatorios son de gran utilidad en numerosas aplicaciones y requieren un trato especial en cualquier lenguaje de programación. C no es una excepción y la mayoría de los compiladores incorporan funciones que generan números aleatorios. Las funciones usuales de la biblioteca estándar de C son: **rand**, **random**, **randomize** y **srand**. Estas funciones se encuentra en el archivo `STDLIB.H`.

- **rand(void)**

La función **rand** genera un número aleatorio. El número calculado por **rand** varía en el rango entero de 0 a `RAND-MAX`. La constante `RAND-MAX` se define en el archivo `STDLIB.H` en forma hexadecimal (por ejemplo, `7FFF`). En consecuencia, asegúrese incluir dicho archivo en la parte superior de su programa.

Cada vez que se llama a **rand()** en el mismo programa, se obtiene un número entero diferente. Sin embargo, si el programa se ejecuta una y otra vez, se devuelven el mismo conjunto de números aleatorios. Un método para obtener un conjunto diferente de números aleatorios es llamar a la función **srand()** o a la macro **randomize**.

La llamada a la función **rand()** se puede asignar a una variable o situar en la función de salida `printf()`.

```
test = rand();
printf("Este es un número aleatorio %d\n",rand());
```

- **randomize(void)**

La macro **randomize** inicializa el generador de números aleatorios con una semilla aleatoria obtenida a partir de una llamada a la función **time**. Dado que esta macro llama a la función **time**, el archivo de cabecera `TIME.H` se incluirá en el programa. No devuelve ningún valor.

```
/* programa para generar 10 números aleatorios */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

int main (void)
{
 int i;

 clrscr(); /* limpia la pantalla */
 randomize();
 for (i=1; i<=10; i++)
 printf("%d ",rand());

 return 0;
}
```

- **srand(semilla)**

La función **srand** inicializa el generador de números aleatorios. Se utiliza para fijar el punto de comienzo para la generación de series de números aleatorios; este valor se denomina *semilla*.

Si el valor de `semilla` es 1, se reinicializa el generador de números aleatorios. Cuando se llama a la función `rand` antes de hacer una llamada a la función `srand`, se genera la misma secuencia que si se hubiese llamado a la función `srand` con el argumento `semilla` tomando el valor 1.

- **random(num)**

La macro `random` genera un número aleatorio dentro de un rango especificado (0 y el límite superior especificado por el argumento `num`). Devuelve un número entero entre 0 y `num-1`.

```
/*
 programa para generar encontrar el mayor de 10 números aleatorios
 entre 0 y 1000
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#define TOPE 1000
#define MAX(x,y) ((x)>(y)?(x):(y))

int main(void)
{
 int mx,i;

 clrscr();
 randomize();
 mx = random(TOPE);
 for (i=2; i<=10; i++)
 {
 int y;
 y = random(TOPE);
 mx = MAX(mx,y);
 }
 printf("El mayor número aleatorio generado: %d", mx);
 return 0;
}
```

En este otro ejemplo de generación de números aleatorios, se fija la semilla en 50 y se genera un número aleatorio.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main(void)
{
 clrscr();
 srand(50);
 printf("Este es un número aleatorio: %d",rand());
 return 0;
}
```

## 7.11. FUNCIONES DE FECHA Y HORA

La familia de microprocesadores 80x86 tiene un sistema de reloj que se utiliza principalmente para controlar el microprocesador, pero se utiliza también para calcular la fecha y la hora.

El archivo de cabecera `TIME.H` define estructuras, macros y funciones para manipulación de fechas y horas. La fecha se guarda de acuerdo con el calendario gregoriano.

Las funciones `time`, `clock`, `_strdate` y `_strtime` devuelven la hora actual como el número de segundos transcurridos desde la medianoche del 1 de enero de 1970 (hora universal, GMT), el tiempo de CPU empleado por el proceso invocante, la fecha y hora actual, respectivamente.

La estructura de tiempo utilizada incluye los miembros siguientes:

```
struct tm
{
 int tm_sec; /* segundos */
 int tm_min; /* minutos */
 int tm_hour; /* horas */
 int tm_mday; /* día del mes 1 a 31 */
 int tm_mon; /* mes, 0 para Ene, 1 para Feb,... */
 int tm_year; /* año desde 1900 */
 int tm_wday; /* días de la semana desde domingo (0-6) */
 int tm_jday; /* día del año desde el 1 de Ene(0-365) */
 int tm_isdst; /* siempre 0 para gmtime */
};
```

- **clock(void)**

La función `clock` determina el tiempo de procesador, en unidades de click, transcurrido desde el principio de la ejecución del programa. Si no se puede devolver el tiempo de procesador se devuelve `-1`.

```
inicio = clock();
fin = clock();
```

- **time(hora)**

La función `time` obtiene la hora actual; devuelve el número de segundos transcurridos desde la medianoche (00:00:00) del 1 de enero de 1970. Este valor de tiempo se almacena entonces en la posición apuntada por el argumento *hora*. Si *hora* es un puntero nulo, el valor no se almacena. El prototipo de la función es:

```
time_t time(time_t *hora);
```

El tipo `time_t` está definido como tipo `long` en `time.h`.

- **localtime(hora)**

Convierte la fecha y hora en una estructura de tipo `tm`. Su prototipo es

```
struct tm *localtime(const time_t *t);
```

- **mktime(t)**

Convierte la fecha en formato de calendario. Toma la información del argumento `t` y determina los valores del día de la semana (`tm_wday`) y del día respecto al inicio del año, también conocido como fecha juliana (`tm_yday`). Su prototipo es

```
time_t mktime(struct tm *t);
```

La función devuelve `-1` en caso de producirse un error.

En este ejercicio se pide el año, mes y día; escribe el día de la semana y los días pasados desde el 1 de enero del año leído. Es utilizado un array de cadenas de caracteres, su estudio se hace en capítulos posteriores.

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

char *dias[] = { " ", "Lunes", "Martes", "Miercoles",
 "Jueves", "Viernes", "Sabado", "Domingo" };

int main(void)
{
```

```

 struct tm fecha;
 int anyo, mes, dia;

 /* Entrada: año, mes y día */
 printf("Año: ");
 scanf("%d", &anyo);
 printf("Mes: ");
 scanf("%d", &mes);
 printf("Día: ");
 scanf("%d", &dia);

 /* Asigna fecha a la estructura fecha, en formato establecido */
 fecha.tm_year = anyo - 1900;
 fecha.tm_mon = mes - 1;
 fecha.tm_mday = dia;
 fecha.tm_hour = 0;
 fecha.tm_min = 0;
 fecha.tm_sec = 1;
 fecha.tm_isdst = -1;

 /* mktime encuentra el día de la semana y el día del año.
 Devuelve -1 si error.
 */
 if (mktime(&fecha) == -1)
 {
 puts(" Error en la fecha.");
 exit(-1);
 }

 /* El domingo, la función le considera día 0 */
 if (fecha.tm_wday == 0)
 fecha.tm_wday = 7;

 printf("\nDía de la semana: %d; día del año: %d",
 fecha.tm_wday, fecha.tm_yday+1);

 /* Escribe el día de la semana */
 printf("\nEs el día de la semana, %s\n", dias[fecha.tm_wday]);
 return 0;
}

```

### Ejercicio 7.4

*Una aplicación de `clock()` para determinar el tiempo de proceso de un programa que calcula el factorial de un número.*

El factorial de  $n! = n*(n-1)*(n-2) \dots 2*1$ . La variable que vaya a calcular el factorial, se define de tipo `long` para poder contener un valor elevado. El número, arbitrariamente, va a estar comprendido entre 3 y 15. El tiempo de proceso va a incluir el tiempo de entrada de datos. La función `clock()` devuelve el tiempo en unidades de click, cada `CLK_TCK` es un segundo. El programa escribe el tiempo en ambas unidades.

```

/*
 En este ejercicio se determina el tiempo del procesador para
 calcular el factorial de un número requerido, entre 3 y 15.
*/

#include <time.h>
#include <stdio.h>

```

```

int main(void)
{
 float inicio, fin;
 int n, x;
 long int fact;

 inicio = clock();
 do {
 printf(" Factorial de (3 <x< 15): ");
 scanf("%d",&x);
 }while (x<=3 || x>=15);

 for (n=x,fact=1; x; x--)
 fact*=x;
 fin = clock();

 printf("\n Factorial de %d! = %ld",n, fact);
 printf("\n Unidades de tiempo de proceso: %f,\t En segundos: %f",
 (fin-inicio), (fin-inicio)/CLK_TCK);

 return 0;
}

```

---

## 7.12. FUNCIONES DE UTILIDAD

C incluyen una serie de funciones de utilidad que se encuentran en el archivo de cabecera `STDLIB.H` y que se listan a continuación.

- **abs(n), labs(n)**

```

int abs(int n)
long labs(long n)

```

devuelven el valor absoluto de *n*.

- **div(num, denom)**

```

div_t div(int num, int denom)

```

Calcula el cociente y el resto de *num*, dividido por *denom* y almacena el resultado en *quot* y *rem*, miembros `int` de la estructura `div_t`.

```

typedef struct
{
 int quot; /* cociente */
 int rem; /* resto */
} div_t;

```

El siguiente ejemplo calcula y visualiza el cociente y el resto de la división de dos enteros.

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 div_t resultado;

 resultado = div(16, 4);
 printf("Cociente %d", resultado.quot);
 printf("Resto %d", resultado.rem);
 return 0;
}

```

- `ldiv(num, denom)`

Calcula el cociente y resto de num dividido por denom, y almacena **los** resultados de quot y rem, miembros long de la estructura `ldiv_t`.

```
typedef struct
{
 long int quot; /* cociente */
 long int rem; /* resto */
} ldiv_t;

resultado = ldiv(1600L, 40L);
```

### 7.13. VISIBILIDAD DE UNA FUNCIÓN

El *ámbito* de un elemento es su visibilidad desde otras partes del programa y la *duración* de un elemento es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuando se crea y cuando se hace disponible. El ámbito de un elemento en C depende de donde se sitúe la definición y de los modificadores que le acompañan. En resumen, se puede decir que un elemento definido dentro de una función tiene *ámbito local* (alcance local), o si se define fuera de cualquier función, se dice que tiene un *ámbito global*. La Figura 7.6 resume el modo en que se ve afectado el ámbito por la posición en el archivo fuente.

Existen dos tipos de clases de almacenamiento en C: `auto` y `static`. Una variable `auto` es aquella que tiene una *duración automática*. No existe cuando el programa comienza la ejecución, se crea en algún punto durante la ejecución y desaparece en algún punto antes de que el programa termine la ejecución. Una variable `static` es aquella que tiene una *duración fija*. El espacio para el elemento de programación se establece en tiempo de compilación; existe en tiempo de ejecución y se elimina sólo cuando el programa desaparece de memoria en tiempo de ejecución.

Las variables con ámbito global se denominan *variables globales* y son las definidas externamente a la función (*declaración externa*). Las variables globales tienen el siguiente comportamiento y atributos:

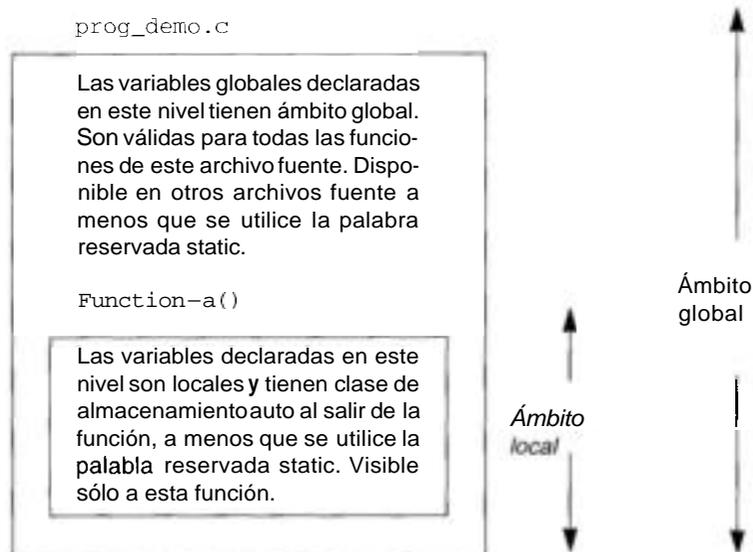


Figura 7.6. Ámbito de variable local y global.

- **Las variables globales tienen duración estática por defecto.** El almacenamiento se realiza en tiempo de compilación y nunca desaparece. Por definición, una variable global no puede ser una variable auto.
- **Las variables globales son visibles globalmente en el archivo fuente.** Se pueden referenciar por cualquier función, a continuación del punto de definición.
- **Las variables globales están disponibles, por defecto, a otros archivos fuente.** Esta operación se denomina *enlace externo*.

### 7.13.1. Variables locales frente a variables globales

Además de las variables globales, es preciso considerar las variables locales. Una *variable local* está definida solamente dentro del bloque o cuerpo de la función y no tiene significado (*vida*) fuera de la función respectiva. Por consiguiente, si una función define una variable como local, el ámbito de la variable está protegido. La variable no se puede utilizar, cambiar o borrar desde cualquier otra función sin una programación específica mediante el paso de valores (parámetros).

**Una *variable local* es una variable que se define dentro de una función.**

**Una *variable global* es una variable que puede ser utilizada por todas las funciones de un programa dado, incluyendo `main()`.**

Para construir variables globales en C, se deben definir fuera de la función `main()`. Para ilustrar el uso de variables locales y globales, examine la estructura de bloques de la Figura 7.7. Aquí la variable global es `x0` y la variable local es `x1`. La función puede realizar operaciones sobre `x0` y `x1`. Sin embargo, `main()` sólo puede operar con `x0`, ya que `x1` no está definida fuera del bloque de la función `funcion1()`. Cualquier intento de utilizar `x1` fuera de `funcion1()` producirá un error.

```

int x0 ; /* variable global */
funcion1 (...) /* prototipo funcional */

int main
{
 ...
 ...
 ...
}

funcion1 (...)
{
 int x1 /* variable local */
 ...
}

```

**Figura 7.7.** `x0` es global al programa completo, mientras que `x1` es local a la función `funcion1()`.

Examine ahora la Figura 7.8. Esta vez existen dos funciones, ambas definen `x1` como variable local. Nuevamente `x0` es una variable global. La variable `x1` sólo se puede utilizar dentro de las dos funciones. Sin embargo, cualquier operación sobre `x1` dentro de `funcion1()` no afecta al valor de `x1` en `funcion2()` y viceversa. En otras palabras, la variable `x1` de `funcion1()` se considera una variable independiente de `x1` en `funcion2()`.

Al contrario que las variables, *las funciones son externas por defecto*. Es preciso considerar la diferencia entre *definición* de una función y *declaración*. Si una declaración de variable comienza con la palabra reservada `extern`, no se considera definición de variable. Sin esta palabra reservada es una definición. Cada definición de variable es al mismo tiempo una declaración de variable. Se puede utilizar una variable sólo después de que ha sido declarada (en el mismo archivo). Únicamente las definiciones de variables asignan memoria y pueden, por consiguiente, contener inicializaciones. Una *variable sólo se define una vez*, pero se puede *declarar* tantas veces como se desee. Una declaración de variable al nivel global (externa a las funciones) es válida desde esa declaración hasta el final del archivo; una declaración en el interior de una función es válida sólo en esa función. En este punto, considérese que las definiciones y declaraciones de variables globales son similares a las funciones; la diferencia principal es que se puede escribir la palabra reservada `extern` en declaraciones de función.

```

int x0 ;
float funcion1(); /* prototipo funcion1 */
float funcion2(); /* prototipo funcion2 */

int main0
{
 ...

 float funcion1()
 {
 int x1 ; /* variable local */
 ...
 }

 float funcion2()
 {
 int x1 ; /*variable local*/
 }
}

```

**Figura 7.8.** `x0` es global al programa completo, `x1` es local tanto a `funcion1()` como a `funcion2()`, pero se tratan como variables independientes.

La palabra reservada `extern` se puede utilizar para notificar al compilador que la declaración del resto de la línea no está definida en el archivo fuente actual, pero está localizada en otra parte, en otro archivo. El siguiente ejemplo utiliza `extern`:

```

/* archivo con la funcion main(): programa.c */
int total;
extern int suma;
extern void f(void);
void main(void)

/*
 archivo con la definición de funciones y variable: modulo.c
*/
int suma;
void f(void)
...

```

Utilizando la palabra reservada **extern** se puede acceder a símbolos externos definidos en otros módulos. `suma` y la función `f()` se declaran externas.

**Las funciones son externas por defecto, al contrario que las variables.**

### 7.13.2. Variables estáticas y automáticas

Los valores asignados a las variables locales de una función se destruyen cuando se termina la ejecución de la función y no se puede recuperar su valor para ejecuciones posteriores de la función. Las variables locales se denominan *variables automáticas*, significando que se pierden cuando termina la función. Se puede utilizar `auto` para declarar una variable

```
auto int ventas;
```

aunque las variables locales se declaran automáticas por defecto y, por consiguiente, el uso de `auto` es opcional y, de hecho, no se utiliza.

Las *variables estáticas* (`static`), por otra parte, mantienen su valor después que una función se ha terminado. Una variable de una función, declarada como estática, mantiene un valor a través de ejecuciones posteriores de la misma función. Haciendo una variable local estática, su valor se retiene de una llamada a la siguiente de la función en que está definida. Se declaran las variables estáticas situando la palabra reservada `static` delante de la variable. Por ejemplo,

```
static int ventas = 10000;
static int dias = 500;
```

Este valor se almacena en la variable estática, sólo la primera vez que se ejecuta la función. Si su valor no está definido, el compilador almacena un cero en una variable estática por defecto.

El siguiente programa ilustra el concepto estático de una variable:

```

#include <stdio.h>
/* prototipo de la función */
void Ejemplo_estatica(int);

void main()
{
 Ejemplo_estatica(1);
 Ejemplo_estatica(2);
 Ejemplo_estatica(3);
}

/* Ejemplo del uso de una variable estática */

```

```

void Ejemplo_estatica(int Llamada)
{
 static int Cuenta;
 if (Llamada == 1)
 Cuenta = 1;
 printf("\n El valor de Cuenta en llamada n° %d es: %d",
 Llamada, Cuenta);
 ++Cuenta;
}

```

Al ejecutar el programa se visualiza:

```

El valor de Cuenta en llamada n° 1 es: 1
El valor de Cuenta en llamada n° 2 es: 2
El valor de Cuenta en llamada n° 3 es: 3

```

Si quita la palabra reservada `static` de la declaración de `Cuenta`, el resultado será:

```

El valor de Cuenta en llamada n° 1 es: 1
El valor de Cuenta en llamada n° 2 es: 1046

```

no se puede predecir cuál es el valor de `Cuenta` en llamadas posteriores a la primera.

*Las variables globales se pueden ocultar de otros archivos fuente utilizando el especificador de almacenamiento de clase `static`.*

Para hacer una variable global privada al archivo fuente (y, por consiguiente, no Útil a otros módulos de código) se le hace preceder por la palabra `static`. Por ejemplo, las siguientes variables se declaran fuera de las funciones de un archivo fuente:

```

static int m = 25;
static char linea_texto[80];
static int indice_linea;
static char bufer[MAXLOGBUF];
static char *pBuffer;

```

Las variables anteriores son privadas al archivo fuente. Observe este ejemplo:

```

#define OFF 0
#define ON 1
...
static unsigned char maestro = OFF;
...

main()

...
}

funcion-a()
{
...
}

```

`maestro` se puede utilizar tanto en `funcion-a()` como en `main()`, en este archivo fuente, pero no se puede declarar como **extern** a otro archivo fuente.

Se puede hacer también una declaración de función `static`. Por defecto, todas las funciones tienen enlace externo y son visibles a otros módulos de programa. Cuando se sitúa la palabra reservada `static` delante de la declaración de la función, el compilador hace privada la función al archivo fuente. Se puede, entonces, reutilizar el nombre de la función en otros módulos fuente del programa.

## 7.14. COMPILACIÓN SEPARADA

Hasta este momento, casi todos los ejemplos que se han expuesto en el capítulo se encontraban en un sólo archivo fuente. Los programas grandes son más fáciles de gestionar si se dividen en varios archivos fuente, también llamados *módulos*, cada uno de los cuales puede contener una o más funciones. Estos módulos se compilan y enlazan por separado posteriormente con un *enlazador*, o bien con la herramienta correspondiente del entorno de programación. Cuando se divide un programa grande en pequeños, los únicos archivos que se recompilan son los que se han modificado. El tiempo de compilación se reduce, dado que pequeños archivos fuente se compilan más rápido que los grandes. Los archivos grandes son difíciles de mantener y editar, ya que su impresión es un proceso lento que utilizará cantidades excesivas de papel.

La Figura 7.9 muestra cómo el enlazador puede construir un programa ejecutable, utilizando módulos objetos, cada uno de los cuales se obtiene compilando un módulo fuente.

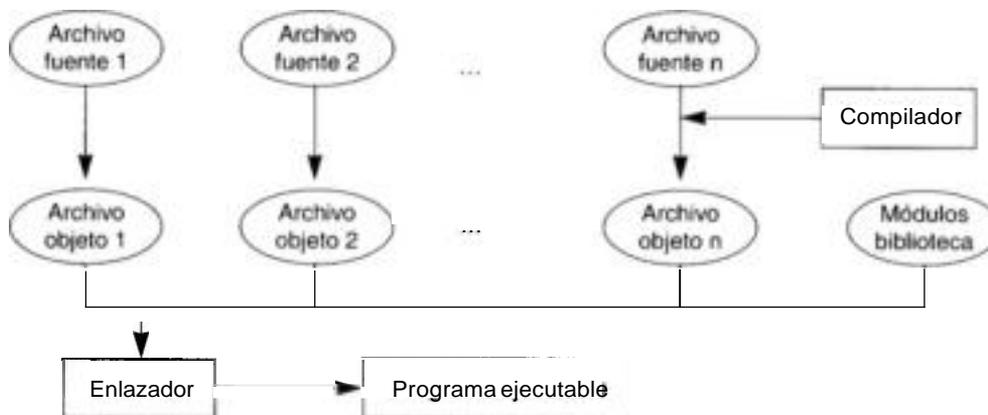


Figura 7.9. Compilación separada.

Cuando se tiene más de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por defecto. Si desea, por razones de legibilidad —no recomendable—, puede utilizar la palabra reservada `extern` con un prototipo de función y en la cabecera.

Se puede desear restringir la visibilidad de una función, haciéndola visible sólo a otras funciones en un archivo fuente. Una razón para hacer esto es tener la posibilidad de tener dos funciones con el mismo nombre en diferentes archivos. Otra razón es reducir el número de referencias externas y aumentar la velocidad del proceso de enlace.

Se puede hacer una función no visible al exterior de un archivo fuente utilizando la palabra reservada `static` con la cabecera de la función y la sentencia del prototipo de función. Se escribe la palabra `static` antes del tipo de valor devuelto por la función. Tales funciones no serán públicas al enlazador, de modo que otros módulos no tendrán acceso a ellas. La palabra reservada **static**, tanto para variables globales como para funciones, es útil para evitar conflictos de nombres y prevenir el uso accidental de ellos. Por ejemplo, imaginemos un programa muy grande que consta de muchos módulos, en el que se busca un error producido por una variable global; si la variable es estática, se puede restringir su búsqueda al módulo en que está definida; si no es así, se extiende nuestra investigación a los restantes módulos en que está declarada (con la palabra reservada `extern`).

Como regla general, son preferibles las variables locales a las globales. Si realmente es necesario o deseable que alguna variable sea **global**, es preferible hacerla estática, lo que significa que será «local» en relación al archivo en que está definida.

### Ejemplo 7.6

Supongamos dos módulos: *MODULO1* y *MODULO2*. En el primero se escribe la función *main()*, hace referencia a funciones y variables globales definidas en el segundo módulo.

```

/* MODULO1.C */
#include <stdio.h>

void main()
{
 void f(int i), g(void);
 extern int n; /* Declaración de n (no definición) */
 f(8);
 n++;
 g();
 puts ("Finde programa.");
}

/* MODULO2.C */

#include <stdio.h>
int n = 100; /* Definición de n (también declaración) */
static int m = 7;

void f(int i)
{
 n += (i+m);
}

void g(void)
{
 printf("n = %d\n",n);
}

```

*f* y *g* se definen en el módulo 2 y se declaran en el módulo 1. Si se ejecuta el programa, se produce la salida

```

n = 116
Fin de programa.

```

Se puede hacer una función invisible fuera de un archivo fuente utilizando la palabra reservada *static* con la cabecera y el prototipo de la función.

## 7.15. VARIABLES REGISTRO (*register*)

Una *variable registro* (*register*) es similar a una variable local, pero en lugar de ser almacenada en la pila, se almacena directamente en un registro del procesador (tal como *ax* o *bx*). Dado que el número

de registros es limitado y además están limitados en tamaño, el número de variables registro que un programa puede crear simultáneamente es muy restringido.

Para declarar una variable registro, se hace preceder a la misma con la palabra reservada `register` ;

```
register int k;
```

La ventaja de las variables registro es su mayor rapidez de manipulación. Esto se debe a que las operaciones sobre valores situados en los registros son normalmente más rápidas que cuando se realizan sobre valores almacenados en memoria. Su uso se suele restringir a segmentos de código mucha veces ejecutados. Las variables registro pueden ayudar a optimizar el rendimiento de un programa proporcionando acceso directo de la CPU a los valores claves del programa.

Una variable registro debe ser local a una función; nunca puede ser global al programa completo. El uso de la palabra reservada `register` no garantiza que un valor sea almacenado en un registro. Esto sólo sucederá si un registro está disponible (libre). Si no existen registros disponibles, C crea la variable como si fuera una variable local normal.

Una aplicación usual de las variables registro es como variable de control de bucles `for` o en la expresión condicional de una sentencia `while`, que se deben ejecutar a alta velocidad.

```
void usoregistro(void)
{
 register int k;
 puts("\n Contar con una variable registro. ');
 for (k = 1; k <= 100; k++)
 printf("%8d", k);
}
```

## 7.16. RECURSIVIDAD

Una *función recursiva* es una función que se llama a sí misma directa o indirectamente. La *recursividad* o *recursión directa* es el proceso por el que una función se llama a sí misma desde el propio cuerpo de la función. La *recursividad* o *recursión indirecta* implica más de una función.

La recursividad indirecta implica, por ejemplo, la existencia de dos funciones: `uno()` y `dos()`. Suponga que `main()` llama a `uno()`, y a continuación `uno()` llama a `dos()`. En alguna parte del proceso, `dos()` llama a `uno()` —una segunda llamada a `uno()`—. Esta acción es recursión indirecta, pero es recursiva, ya que `uno()` ha sido llamada dos veces, sin retornar nunca a su llamadora.

Un proceso recursivo debe tener una condición de terminación, ya que si no puede continuar indefinidamente.

Un algoritmo típico que conduce a una implementación recursiva es el cálculo del factorial de un número. El factorial de  $n$  ( $n!$ ).

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

En consecuencia, el factorial de 4 es igual a  $4*3*2*1$ , el factorial de 3 es igual a  $3*2*1$ . Así pues, el factorial de 4 es igual a 4 veces el factorial de 3. La Figura 7.10 muestra la secuencia de sucesivas invocaciones a la función factorial.

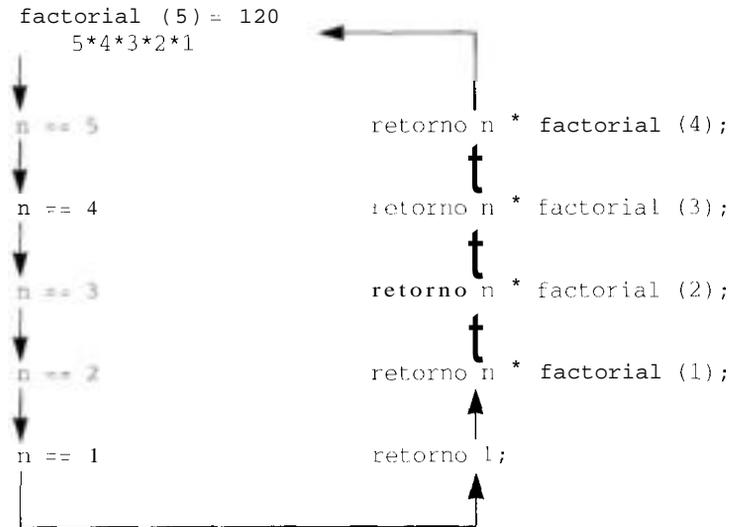


Figura 7.10. Llamadas a funciones recursivas para factorial(5).

### Ejemplo 7.7

Realizar el algoritmo de la función factorial.

La implementación de la función recursiva factorial es:

```
double factorial(int numero)
{
 if (numero > 1)
 return numero * factorial(numero-1);
 return 1;
}
```

### Ejemplo 7.8

Contar valores de 1 a 10 de modo recursivo.

```
#include <stdio.h>
void contar(int cima);
int main()
{
 contar(10);
 return 0;
}
void contar(int cima)
{
 if (cima > 1)
 contar(cima-1);
 printf("%d ", cima);
}
```

**Ejemplo 7.9**

**Determinar si un número entero positivo es par o impar; con dos funciones que se llaman mutuamente: recursividad indirecta.**

```
#include <stdio.h>

int par(int n);
int impar(int n);

int main(void)
{
 int n;

 /* Entrada: entero > 0 */
 do {
 printf("\nEntero > 0: ");
 scanf("%d", &n);
 } while (n<=0);

 /* Llamda a la función par() */
 if (par(n))
 printf("El numero %d es par.",n);
 else
 printf("El numero %d es impar.",n);
 return 0;
}

int par(int n)
{
 if (n == 0)
 return 1; /* es par */
 else
 return impar(n-1);
}

int impar(int n)
{
 if (n == 0)
 return 0; /* es impar */
 else
 return par(n-1);
}
```

La función `par()` llama a la función `impar()`, ésta a su vez llama a la función `par()`. La condición para terminar de hacer llamadas es que `n` sea cero; el cero se considera par.

## 7.17. RESUMEN

Las funciones son la base de la construcción de programas en C. Se utilizan funciones para subdividir problemas grandes en tareas más pequeñas. El encapsuamiento de las características en funciones, hace los programas más fáciles de mantener. El uso de funciones ayuda al programador a reducir el tamaño de su programa, ya que se puede llamar repetidamente y reutilizar el código dentro de una función.

En este capítulo habrá aprendido lo siguiente:

- el concepto, declaración, definición y uso de una función;
- las funciones que devuelven un resultado lo hacen a través de la sentencia `return`;
- los parámetros de funciones se pasan por valor, para un paso por referencia se utilizan punteros;
- el modificador `const` se utiliza cuando se desea que los parámetros de la función sean valores de sólo lectura;
- el concepto y uso de prototipos, cuyo uso es recomendable en C;
- la ventaja de utilizar macros con argumentos, para aumentar la velocidad de ejecución;
- el concepto de *ámbito* o *alcance* y *visibilidad*, junto con el de *variable global* y *local*;
- clases de almacenamiento de variables en memoria: `auto`, `extern`, `register` y `static`.

La biblioteca estándar C de funciones en tiempo de ejecución incluye gran cantidad de funciones. Se agrupan por categorías, entre las que destacan:

- manipulación de caracteres;
- numéricas;
- tiempo y hora;
- conversión de datos;
- búsqueda y ordenación;
- etc.

Tenga cuidado de incluir el archivo de cabecera correspondiente cuando desee incluir funciones de biblioteca en sus programas.

Una de las características más sobresalientes de C que aumentan considerablemente la potencia de los programas es la posibilidad de manejar las funciones de modo eficiente, apoyándose en la propiedad que les permite ser compiladas por separado.

Otros temas tratados han sido:

- *Ámbito* o las *reglas de visibilidad* de funciones y variables.

- En entorno de un programa tiene cuatro tipos de ámbito: de programa, archivo fuente, función y bloque. Una variable está asociada a uno de esos ámbitos y es invisible (no accesible) desde otros ámbitos.
- Las *variables globales* se declaran fuera de cualquier función y son visibles a todas las funciones. Las variables locales se declaran dentro de una función y sólo pueden ser utilizadas por esa función.

```
int i; /* variable global,
 ámbito de programa
 */
static int j /* ámbito de archivo
 */

main()
{
 int d, e; /* variable local,
 ámbito de función */
 ...
}

func (int j)
{
 if (j > 3)
 {
 int i; /* ámbito de bloque */
 for (i = 0; i < 20; i++)
 func2(i);
 }
 /* i ya no es visible */
}
```

- *Variables automáticas* son las variables, por defecto, declaradas localmente en una función.
- *Variables estáticas* mantienen su información, incluso después que la función ha terminado.

Cuando se llama de nuevo la función, la variable se pone al valor que tenía cuando se llamó anteriormente.

- *Funciones recursivas* son aquellas que se pueden llamar a sí mismas.
- Las *variables registro* se pueden utilizar cuando se desea aumentar la velocidad de procesamiento de ciertas variables.

## 7.18. EJERCICIOS

- 7.1. Escribir una función que tenga un argumento de tipo entero y que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.
- 7.2. Escribir una función lógica de dos argumentos enteros, que devuelva *true* si uno divide al otro y *false* en caso contrario.
- 7.3. Escribir una función que convierta una temperatura dada en grados Celsius a grados Fahrenheit. La fórmula de conversión es:
- $$F = \frac{9}{5}C + 32$$
- 7.4. Escribir una función lógica *Digito* que determine si un carácter es uno de los dígitos de 0 a 9.

- 7.5. Escribir una función lógica *Vocal* que determine si un carácter es una vocal.
- 7.6. Escribir una función *Redondeo* que acepte un valor real *Cantidad* y un valor entero *Decimales* y devuelva el valor *Cantidad* redondeado al número especificado de *Decimales*. Por ejemplo, *Redondeo* (20.563,2) devuelve 20.56.
- 7.7. Determinar y visualizar el número más grande de dos números dados, mediante un subprograma.
- 7.8. Escribir un programa recursivo que calcule los *N* primeros números naturales.

## 7.19. PROBLEMAS

- 7.1. Escribir un programa que solicite del usuario un carácter y que sitúe ese carácter en el centro de la pantalla. El usuario debe poder a continuación desplazar el carácter pulsando las letras A (arriba), B (abajo), I (izquierda), D (derecha) y F (fin) para terminar.
- 7.2. Escribir una función que reciba una cadena de caracteres y la devuelva en forma inversa (ñola' se convierte en 'aloh').
- 7.3. Escribir una función que determine si una cadena de caracteres es un palíndromo (un palíndromo es un texto que se lee igual en sentido directo y en inverso: radar).

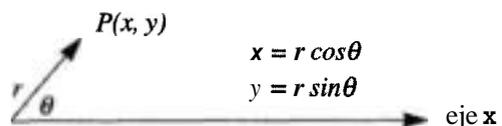
- 7.4. Escribir un programa mediante una función que acepte un número de día, mes y año y lo visualice en el formato

dd/mm/aa

Por ejemplo, los valores 8, 10 y 1946 se visualizan como

8/10/46

- 7.5. Escribir un programa que utilice una función para convertir coordenadas polares a rectangulares.



7.6. Escribir un programa que lea un entero positivo y a continuación llame a una función que visualice sus factores primos.

7.7. Escribir un programa, mediante funciones, que visualice un calendario de la forma:

| L  | M  | M  | J  | V  | S  | D  |
|----|----|----|----|----|----|----|
|    |    | 1  | 2  | 3  | 4  | 5  |
| 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 |    |    |    |

El usuario indica únicamente el mes y el año.

7.8. Escribir un programa que lea los dos enteros positivos  $n$  y  $b$  que llame a una función *CambiarBase* para calcular y visualizar la representación del número  $n$  en la base  $b$ .

7.9. Escribir un programa que permita el cálculo del *mdc* (máximo común divisor) de dos números por el algoritmo de Euclides. (Dividir  $a$  entre  $b$ , se obtiene el cociente  $q$  y el resto  $r$  si es cero  $b$  es el *mdc*, si no se divide  $b$  entre  $r$ , y así sucesivamente hasta encontrar un resto cero, el último divisor es el *mdc*.) La función *mdc()* devolverá el máximo común divisor.

7.10. Escribir una función que devuelva el inverso de un número dado (1234, inverso 4321).

7.11. Calcular el coeficiente del binomio con una función factorial.

$$\binom{m}{n} = \frac{m!}{n! (m-n)!} \quad \text{donde} \quad m! = \begin{cases} 1 & \text{si } m = 0 \\ 1,2,3 \dots m & \text{si } m \neq 0 \end{cases}$$

7.12. Escribir una función que permita calcular la serie:

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n \cdot (n + 1) \cdot (2n + 1)}{6} = n \cdot (n + 1) \cdot (2 \cdot n + 1) / 6$$

7.13. Escribir un programa que lea dos números  $x$  y  $n$  y en una función calcule la suma de la progresión geométrica.

$$1 + x + x^2 + x^3 + \dots + x^n$$

7.14. Escribir un programa que encuentre el valor mayor, el valor menor y la suma de los datos de entrada. Obtener la media de los datos mediante una función.

7.15. Escribir una función que acepte un parámetro  $x(x \neq 0)$  y devuelva el siguiente valor:

$$\frac{1}{x^2 \left( \frac{e^{1.435}}{x} - 1 \right)}$$

7.16. Escribir una función con dos parámetros,  $x$  y  $n$ , que devuelva lo siguiente:

$$x + \frac{x^n}{n} - \frac{x^{n+1}}{n+2} \quad \text{si } x \geq 0$$

$$x + \frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} \quad \text{si } x < 0$$

7.17. Escribir una función que tome como parámetros las longitudes de los tres lados de un triángulo ( $a$ ,  $b$  y  $c$ ) y devuelva el área del triángulo.

$$\text{Área} = \sqrt{p(p-a)(p-b)(p-c)}$$

$$\text{donde } p = \frac{a+b+c}{2}$$

7.18. Escribir un programa mediante funciones que realicen las siguientes tareas:

- a) Devolver el valor del día de la semana en respuesta a la entrada de la letra inicial (mayúscula o minúscula) de dicho día.
- b) Determinar el número de días de un mes.

7.19. Escribir un programa que lea una cadena de hasta diez caracteres que representa a un número en numeración romana e imprima el formato del número romano y su equivalente en numeración arábiga. Los caracteres romanos y sus equivalentes son:

|   |      |   |    |
|---|------|---|----|
| M | 1000 | L | 50 |
| D | 500  | X | 10 |
| C | 100  | V | 5  |
|   |      | I | 1  |

Compruebe su programa para los siguientes datos:

LXXXVI (86), CCCXIX (319), MCCLIV (1254).

7.20. Escriba una función que calcule cuántos puntos de coordenadas enteras existen dentro de un triángulo del que se conocen las coordenadas de sus tres vértices.

7.21. Escribir un programa que mediante funciones determine el área del círculo correspondiente a la circunferencia circunscrita de un triángulo del que conocemos las coordenadas de los vértices.

7.22. Dado el valor de un ángulo escribir una función que muestre el valor de todas las funciones trigonométricas correspondientes al mismo.

## CAPÍTULO 8

# **ARRAYS (LISTAS Y TABLAS)**

---

### CONTENIDO

- 8.1. *Arrays.*
- 8.2. Inicialización de un array.
- 8.3. *Arrays* de caracteres y cadenas de texto.
- 8.4. *Arrays* multidimensionales.
- 8.5. Utilización de arrays *como* parámetros:
- 8.6. Ordenación de listas.
- 8.7. Búsqueda en listas.
- 8.8. *Resumen.*
- 8.9. *Ejercicios.*
- 8.10. *Problemas.*

---

## INTRODUCCIÓN

En capítulos anteriores se han descrito las características de los tipos de datos básicos o simples (carácter, entero y coma flotante). Asimismo, se ha aprendido a definir y utilizar constantes simbólicas utilizando `const`, `#define` y el tipo `enum`. En este capítulo continuaremos el examen de los restantes tipos de datos de C, examinando especialmente el tipo `array` (lista o tabla), la estructura, la unión.

En este capítulo aprenderá el concepto y tratamiento de los arrays. Un array almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. El array es muy importante por diversas razones. Una operación muy importante es almacenar secuencias o cadenas de texto. Hasta el momento C proporciona datos de un solo carácter; utilizando el tipo `array`, se puede crear una variable que contenga un grupo de caracteres.

---

## CONCEPTOS CLAVE

- Array.
- Array de caracteres.
- Arrays bidimensionales.
- Arrays multidimensionales.
- Cadena de texto.
- Declaración de un *array*.
- Inicialización de un *array*.
- Lista, tabla.
- Ordenación y búsqueda.
- Parámetros de tipo *array*.

## 8.1. ARRAYS

Un *array* (lista o tabla) es una secuencia de datos del mismo tipo. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3, etc. El tipo de elementos almacenados en el array puede ser cualquier tipo de dato de C, incluyendo estructuras definidas por el usuario, como se describirá más tarde. Normalmente el array se utiliza para almacenar tipos tales como *char*, *int* o *float*.

Un array puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de personas que residen en cada una de las diecisiete comunidades autónomas españolas. Cada ítem del array se denomina *elemento*.

Los elementos de un array se numeran, como ya se ha comentado, consecutivamente 0, 1, 2, 3, ... Estos números se denominan *valores índice* o *subíndice* del array. El término «subíndice» se utiliza ya que se especifica igual que en matemáticas, como una secuencia tal como  $a_0, a_1, a_2, \dots$ . Estos números localizan la posición del elemento dentro del array, proporcionando *acceso directo* al array.

Si el nombre del array es *a*, entonces  $a[0]$  es el nombre del elemento que está en la posición 0,  $a[1]$  es el nombre del elemento que está en la posición 1, etc. En general, el elemento *i-ésimo* está en la posición *i-1*. De modo que si el array tiene *n* elementos, sus nombres son  $a[0], a[1], \dots, a[n-1]$ . Gráficamente se representa así el array *a* con seis elementos.

|   |      |      |      |      |      |      |
|---|------|------|------|------|------|------|
| a | 25.1 | 34.2 | 5.25 | 7.45 | 6.09 | 7.54 |
|---|------|------|------|------|------|------|

Figura 8.1. Array de seis elementos.

El array *a* tiene 6 elementos:  $a[0]$  contiene 25.1,  $a[1]$  contiene 34.2,  $a[2]$  contiene 5.25,  $a[3]$  contiene 7.45,  $a[4]$  contiene 6.09 y  $a[5]$  contiene 7.54. El diagrama de la Figura 8.1 representa realmente una región de la memoria de la computadora, ya que un array se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

En C los índices de un array siempre tienen como límite inferior 0, como índice superior el tamaño del array menos 1.

### 8.1.1. Declaración de un array

Al igual que con cualquier tipo de variable, se debe declarar un array antes de utilizarlo. Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el *tamaño* o *longitud* del array. Para indicar al compilador el *tamaño* o *longitud* del array se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La *sintaxis* para declarar un array de una dimensión determinada es:

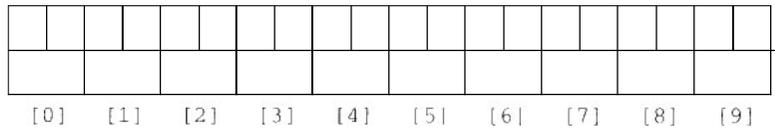
```
tipo nombreArray[numeroDeElementos];
```

Por ejemplo, para crear un array (lista) de diez variables enteras, se escribe:

```
int numeros[10];
```

Esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. En C los enteros ocupan, normalmente, 2 bytes, de modo que un array de diez enteros ocupa 20 bytes de memoria. La Figura 8.2 muestra el esquema de un array de diez elementos; cada elemento puede tener su propio valor.

Array de datos enteros: a



Un array de enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza dos bytes. Se accede a cada elemento de array mediante un índice que comienza en cero. Así, el elemento quinto (`a[4]`) del array ocupa los bytes 9º y 10º.

**Figura 8.2.** Almacenamiento de un array en memoria.

Se puede acceder a cada elemento del array utilizando un índice en el nombre del array. Por ejemplo, `printf("%d \n", numeros[4]);`

visualiza el valor del elemento 5 del array. Los arrays siempre comienzan en el elemento 0. Así pues, el array `numeros` contiene los siguientes elementos individuales:

|            |            |            |            |
|------------|------------|------------|------------|
| numeros[0] | numeros[1] | numeros[2] | numeros[3] |
| numeros[4] | numeros[5] | numeros[6] | numeros[7] |
| numeros[8] | numeros[9] |            |            |

Si por ejemplo, se quiere crear un array de números reales y su tamaño es una constante representada por un parámetro

```
#define N 20
float vector[N];
```

Para acceder al elemento 3 y leer un valor de entrada:

```
scanf("%f ", &vector[2]);
```

### Precaución

**C no comprueba que los índices del array están dentro del rango definido. Así, por ejemplo, se puede intentar acceder a `numeros[12]` y el compilador no producirá ningún error, lo que puede producir un fallo en su programa, dependiendo del contexto en que se encuentre el error.**

### 8.1.2. Subíndices de un array

El índice de un array se denomina, con frecuencia, *subíndice del array*. El término procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

|         |                   |            |
|---------|-------------------|------------|
| numeros | <i>equivale a</i> | numeros[0] |
| numeros | <i>equivule a</i> | numeros[3] |

El método de numeración del elemento *i-ésimo* con el índice o subíndice *i-1* se denomina *indexación basada en cero*. Su uso tiene el efecto de que el índice de un elemento del array es siempre el mismo que el número de «pasos» desde el elemento inicial `a[0]` a ese elemento. Por ejemplo, `a[3]` está a 3 pasos o posiciones del elemento `a[0]`. La ventaja de este método se verá de modo más evidente al tratar las relaciones entre arrays y punteros.

**Ejemplos**

|                                                    |                                                                                                               |
|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>int edad[5];</code>                          | <i>Array edad contiene 5 elementos: el primero, edad[0] y el último, edad[4].</i>                             |
| <code>int pesos[25], longitudes[100];</code>       | <i>Declara 2 arrays de enteros.</i>                                                                           |
| <code>float salarios[25];</code>                   | <i>Declara un array de 25 elementos float.</i>                                                                |
| <code>double temperaturas[50];</code>              | <i>Declara un array de 50 elementos double.</i>                                                               |
| <code>char letras[15];</code>                      | <i>Declara un array de caracteres.</i>                                                                        |
| <code>#define MX 120<br/>char buffer[MX+1];</code> | <i>Declara un array de caracteres de tamaño MX+1, el primer elemento es buffer[0] y el último buffer[MX].</i> |

En los programas se pueden referenciar elementos del array utilizando fórmulas para los subíndices. Mientras que el subíndice puede evaluar a un entero, se puede utilizar una constante, una variable o una expresión para el subíndice. Así, algunas referencias individuales a elementos son:

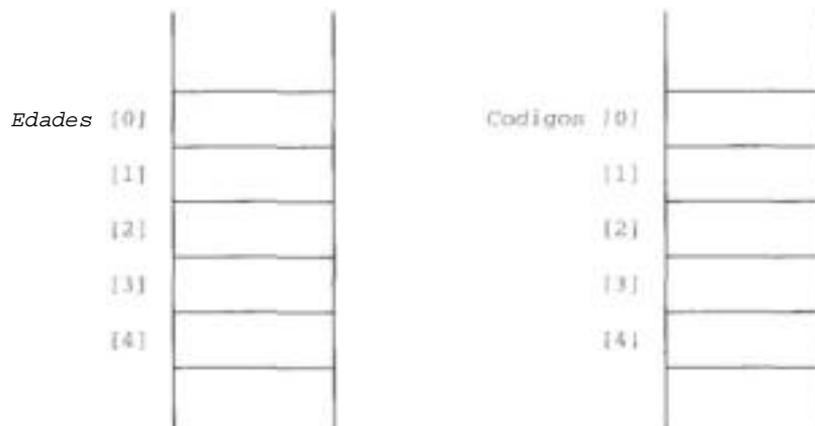
```
edad[4]
ventas[total+5]
bonos[mes]
salario[mes[i]*5]
```

**8.1.3. Almacenamiento en memoria de los arrays**

Los elementos de los arrays se almacenan en bloques contiguos. Así, por ejemplo, los arrays

```
int edades[5];
char codigos[5];
```

se representan gráficamente en memoria en la Figura 8.3.



**Figura 8.3.** Almacenamiento en memoria de arrays.

## Nota

Todos **los** subíndices de **los** arrays comienzan con **0**.

## Precaución

C permite asignar valores fuera de rango a **los** subíndices. Se debe tener cuidado no hacer esta acción, debido a **que se sobrescribirían datos o código**.

Los arrays de caracteres funcionan de igual forma que los arrays numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte. **Así**, por ejemplo, un array llamado `nombre` se puede representar en la Figura 8.4.

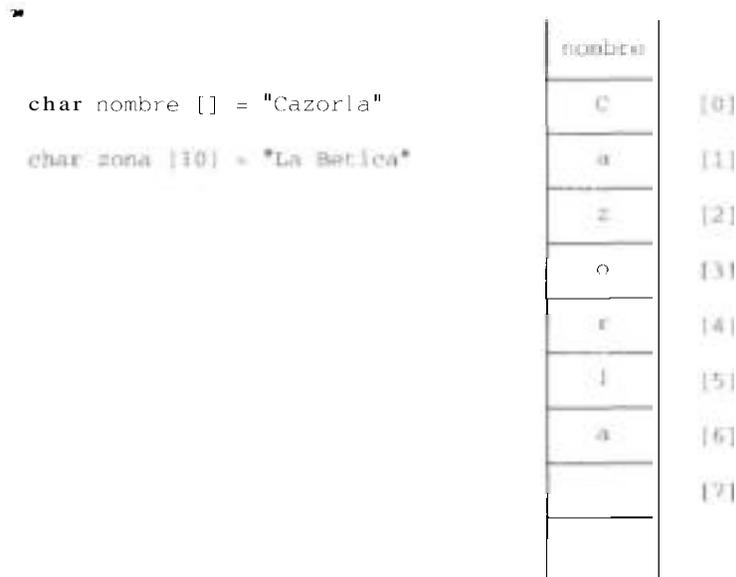


Figura 8.4. Almacenamiento de un arrays de caracteres en memoria.

A tener en cuenta, en las cadenas de caracteres el sistema siempre inserta un último carácter (nulo) para indicar fin de cadena.

### 8.1.4. El tamaño de los arrays

El operador `sizeof` devuelve el número de bytes necesarios para contener su argumento. Si se usa `sizeof` para solicitar el tamaño de un array, esta función devuelve el número de bytes reservados para el array completo.

Por ejemplo, supongamos que se declara un array de enteros de 100 elementos denominado `edades`; si se desea conocer el tamaño del array, se puede utilizar una sentencia similar a:

```
n = sizeof(edades);
```

donde  $n$  tomará el valor 200. Si se desea solicitar el tamaño de un elemento individual del array, tal como

```
n = sizeof(edades[6]);
```

$n$  almacenará el valor 2 (número de bytes que contienen un entero).

### 8.1.5. Verificación del rango del índice de un array

C, al contrario que otros lenguajes de programación —por ejemplo, Pascal—, no verifica el valor del índice de la variable que representa al array. Así, por ejemplo, en Pascal si se define un array  $a$  con índices 0 a 5, entonces  $a[6]$  hará que el programa se «rompa» en tiempo de ejecución.

---

#### Ejemplo 8.1

*Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array.* ■

```
double suma(const double a[], const int n)
{
 double S = 0.0;
 if (n * sizeof(double) > sizeof(a))
 return 0;
 for (int i = 0; i < n; i++)
 S += a[i];
 return S;
}
```

---

## 8.2. INICIALIZACIÓN DE UN ARRAY

Se deben asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de enteros `precios`, se puede escribir:

```
precios[0] = 10;
precios[1] = 20;
precios[3] = 30;
precios[4] = 40;
...
```

La primera sentencia fija `precios[0]` al valor 10, `precios[1]` al valor 20, etc. Sin embargo, este método no es práctico cuando el array contiene muchos elementos. El método utilizado, normalmente, es inicializar el array completo en una sola sentencia.

Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros[6] = {10, 20, 30, 40, 50, 60};
int n[] = {3, 4, 5} /* Declara un array de 3 elementos */
char c[] = {'L', 'u', 'i', 's'}; /* Declara un array de 4 elementos */
```

El array `numeros` tiene 6 elementos, `n` tiene 3 elementos y el array `c` tiene 4 elementos.

En C los arrays de caracteres, las cadenas, se caracterizan por tener un carácter final que indica el fin de la cadena, es el carácter nulo. Lo habitual es inicializar un array de caracteres (una variable cadena) con una constante cadena.

```
char s[] = "Puesta del Sol";
```

## Nota

**C puede dejar los corchetes vacíos, sólo cuando se asignan valores al array, tal como**

```
int cuenta[] = {15, 25, -45, 0, 50};
```

**El compilador asigna automáticamente cinco elementos a cuenta**

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el número de elementos del array es pequeño. Por ejemplo, para inicializar un array (lista) de 10 enteros a los valores 10 a 1, y a continuación visualizar dichos valores en un orden inverso, se puede escribir:

```
int cuenta[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
for (i = 9; i >= 0; i--)
 printf("\n cuenta descendente %d = %d", i, cuenta[i]);
```

Se pueden asignar constantes simbólicas como valores numéricos, de modo que las sentencias siguientes son válidas:

```
#define ENE 31
#define FER 28
#define MAR 31
...
int meses[12] = {ENE, FEB, MAR, ABR, MAY, JUN,
 JUL, AGO, SEP, OCT, NOV, DIC};
```

Pueden asignarse valores a un array utilizando un bucle `for` `while`/`do-while`, y éste suele ser el sistema más empleado normalmente. Por ejemplo, para inicializar todos los valores del array `numeros` al valor 0, se puede utilizar la siguiente sentencia:

```
for (i = 0; i <= 5; i++)
 numeros[i] = 0;
```

debido a que el valor del subíndice *i* varía de 0 a 5, cada elemento del array `numeros` se inicializa y establece a cero.

---

## Ejemplo 8.2

*El programa INICIALI.C lee ocho enteros; a continuación visualiza el total de los números.*

```
#include <stdio.h>
#define NUM 8
int main()
{
 int nums[NUM];
 int i;
 int total = 0;
```

```

for (i = 0; i < NUM; i++)
{
 printf("Por favor, introduzca el número: ");
 scanf("%d",&nums[i]);
}
printf("\nLista de números: ");
for (i = 0; i < NUM; i++)
{
 printf("%d ",nums[i]);
 total += nums[i];
}

printf("\nLa suma de los números es %d",total);
return 0;
}

```

Las variables globales que representan arrays se inicializan a 0 por defecto. Por ello, la ejecución del siguiente programa visualiza 0 para los 10 valores del array:

```

int lista[10];
int main()
{
 int j;
 for (j = 0; j <= 9; j++)
 printf("\n lista[%d] = %d",j,lista[j]);
 return 0;
}

```

A Así, por ejemplo, en

```

int Notas[5];
void main()
{
 static char Nombres[5];
}

```

**Si se define un array globalmente o un array estático y no se proporciona ningún valor de inicialización, el compilador inicializará el array con un valor por defecto (cero para arrays de elementos enteros y reales --coma flotante— y carácter nulo para arrays de caracteres).**

el array de enteros se ha definido globalmente y el array de caracteres se ha definido como un array local estático de `main()`. Si se ejecuta ese segmento de programa, se obtendrán las siguientes asignaciones a los elementos de los arrays:

| Notas | Nombres  |
|-------|----------|
| [0] 0 | 101 '\0' |
| [1] 0 | [1] '\0' |
| [2] 0 | [2] '\0' |
| [3] 0 | [3] '\0' |
| [4] 0 | [4] '\0' |

### 8.3. ARRAYS DE CARACTERES Y CADENAS DE TEXTO

Una cadena de texto es un conjunto de caracteres, tales como «ABCDEFGH». C soporta cadenas de texto utilizando un array de caracteres que contenga una secuencia de caracteres:

```
char cadena[] = "ABCDEFGH";
```

Es importante comprender la *diferencia* entre un array de caracteres y una cadena de caracteres. Las *cadena*s contienen un carácter nulo al final del array de caracteres.

**Las cadenas se deben almacenar en arrays de caracteres, pero no todos los arrays de caracteres contienen cadenas.**

Examine la Figura 8.5. donde se muestra una cadena de caracteres y un array de caracteres.

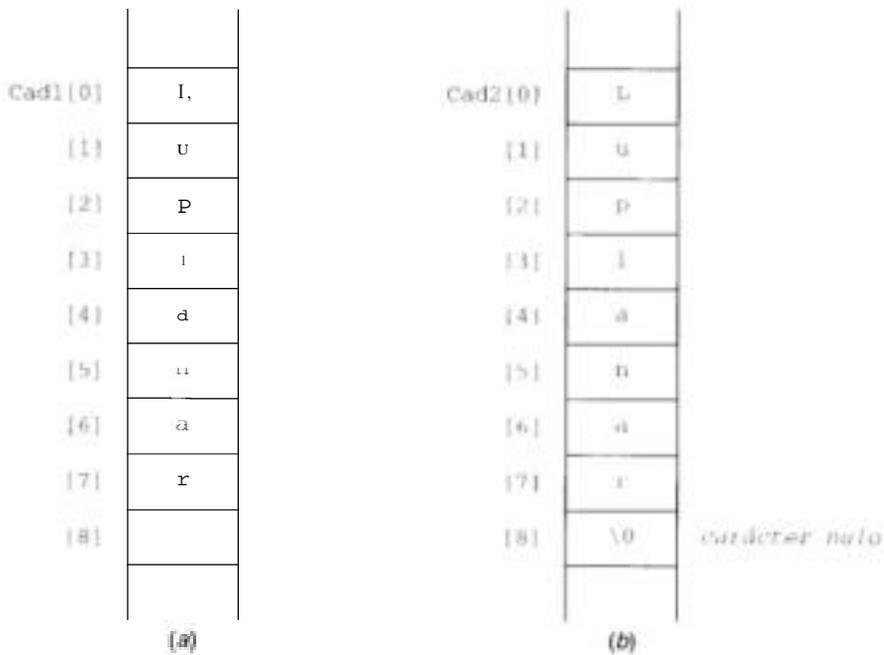


Figura 8.5. (a) Array de caracteres; (b) cadena.

|        |   |   |   |   |   |   |    |
|--------|---|---|---|---|---|---|----|
| Cadena | A | B | C | D | E | F | \0 |
|--------|---|---|---|---|---|---|----|

```
Cadena[3] = 'D';
Cadena[4] = 'E';
Cadena[5] = 'F';
Cadena[6] = '\0';
```

Sin embargo, no se puede asignar una cadena a un array del siguiente modo:

```
Cadena = "ABCDEF";
```

Para copiar una constante cadena o copiar una variable de cadena a otra variable de cadena se debe utilizar la función de la biblioteca estándar —posteriormente se estudiará— `strcpy()` («copiar cadenas»). `strcpy()` permite copiar una constante de cadena en una cadena. Para copiar el nombre "Abracadabra" en el array `nombre`, se puede escribir

```
strcpy(nombre, "Abracadabra"); /*Copia Abracadabra en nombre */
```

`strcpy()` añade un carácter nulo al final de la cadena. A fin de que no se produzcan errores en la sentencia anterior, se debe asegurar que el array de caracteres `nombre` tenga elementos suficientes para contener la cadena situada a su derecha.

### Ejemplo 8.3

*Rellenar los elementos de un array con números reales positivos procedentes del teclado.*

```
#include <stdio.h>
/* Constantes y variables globales */
#define MAX 10
float muestra[MAX];
void main()
{
 int i;
 printf("\nIntroduzca una lista de %d elementos positivos.\n",MAX);
 for (i = 0; i < MAX; muestra[i]>0?++i:i)
 scanf("%f",&muestra[i]);
}
```

En el bucle principal, sólo se incrementa `i` si `muestra[i]` es positivo: `muestra[i]>0?++i:i`. Con este incremento condicional se consigue que todos los valores almacenados sean positivos.

### Ejemplo 8.4

*Visualizar el array `muestra` después de introducir datos en el mismo, separándolos con el tabulador.*

```
#include <stdio.h>
#define MAX 10
float muestra[MAX];
void main()
{
 int i;
 printf("\nIntroduzca una lista de %d elementos positivos.\n",MAX);
 for (i = 0; i < MAX; muestra[i]>0?++i:i)
 scanf("%f",&muestra[i]);
 printf("\nDatos leídos del teclado: ");
 for (i = 0, i < MAX; ++i)
 printf("%f\t",muestra[i]);
}
```

## 8.4. ARRAYS MULTIDIMENSIONALES

Los arrays vistos anteriormente se conocen como arrays *unidimensionales* (una sola dimensión) y se caracterizan por tener un solo subíndice. Estos arrays se conocen también por el término *listas*. Los *arrays multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear arrays de tantas dimensiones como requieran sus aplicaciones, esto es, tres, cuatro o más dimensiones.

Un array de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (Fig. 8.6).

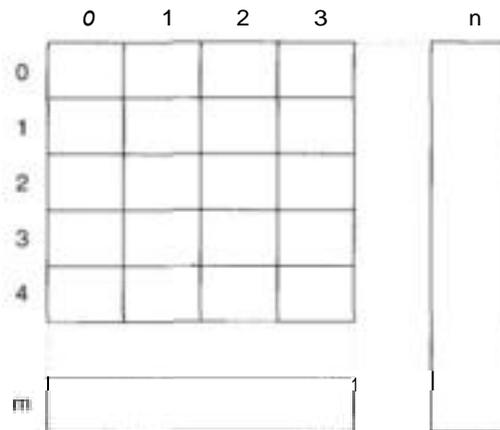


Figura 8.6. Estructura de un array de dos dimensiones.

Obsérvese que en el array bidimensional de la Figura 8.6, si las filas se etiquetan de 0 a  $m$  y las columnas de 0 a  $n$ , el número de elementos que tendrá el array será el resultado del producto  $(m+1) \times (n+1)$ . El sistema de localizar un elemento será por las coordenadas representadas por su número de fila y su número de columna  $(a, b)$ . La sintaxis para la declaración de un array de dos dimensiones es:

`<tipo de datoElemento> <nombre array> [<NúmeroDeFilas>] [<NúmeroDeColumnas>]`

Algunos ejemplos de declaración de tablas:

```
char Pantalla[25][80];
int puestos[6][8];
int equipos[4][301];
int matriz[4][2];
```

### Atención

**Al contrario que otros lenguajes, C requiere que cada dimensión esté encerrada entre corchetes. La sentencia**

```
int equipos[4, 301
```

**no es válida.**

Un array de dos dimensiones en realidad es un *array de arrays*. Es decir, es un array unidimensional, y cada elemento no es un valor entero, o de coma flotante o carácter, sino que cada elemento es otro array.

Los elementos de los arrays se almacenan en memoria de modo que el subíndice más próximo al nombre del array es la fila y el otro subíndice, la columna. En la Tabla 8.1 se representan todos los elementos y sus posiciones relativas en memoria del array, `int tabla[4][2]`; suponiendo que cada entero ocupa 2 bytes.

**Tabla 8.1.** Un array bidimensional.

| Elemento                 | Posición relativa de memoria |
|--------------------------|------------------------------|
| <code>tabla[0][0]</code> | 0                            |
| <code>tabla[0][1]</code> | 2                            |
| <code>tabla[1][0]</code> | 4                            |
| <code>tabla[1][1]</code> | 6                            |
| <code>tabla[2][0]</code> | 8                            |
| <code>tabla[2][1]</code> | 10                           |
| <code>tabla[3][0]</code> | 12                           |
| <code>tabla[3][1]</code> | 14                           |

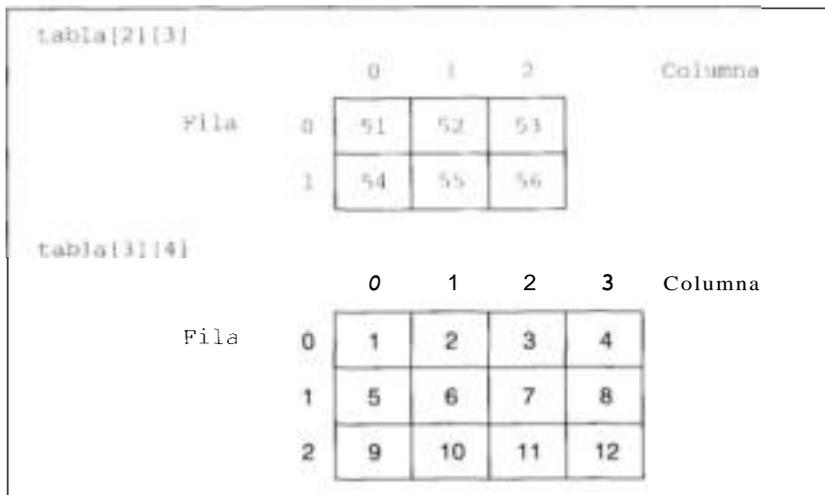
### 8.4.1. Inicialización de arrays multidimensionales

Los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves, como en los ejemplos siguientes:

```
1. int tabla[2][3] = {51, 52, 53, 54, 55, 56};
```

o bien en los formatos mas amigables:

```
int tabla[2][3]= { {51, 52, 53},
 {54, 55, 56} };
int tabla[2][3]= {{51, 52, 53}, {54, 55, 56}};
int tabla[2][3]= {
 {51, 52, 53},
 {54, 55, 56}
};
```



**Figura 8.7.** Tablas de dos dimensiones.

```
2. int tabla:[3][4] = {
 {1, 2, 3, 4},
 {15, 6, 7, 8},
 {9, 10, 11, 12}
};
```

### Consejo

Los arrays multidimensionales (a menos que sean globales) no se inicializan a valores específicos a menos que *se* les asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o más elementos, pero no todos, C rellena el resto con ceros o valores nulos ('\`\0`'). Si se desea inicializar a cero un array multidimensional, utilice una sentencia tal como ésta:

```
float ventas[3][4] = {0.,0.,0.,0., 0.,0.,0.,0., 0.,0.,0.,0.};
```



Figura 8.8. Almacenamiento en memoria de tabla [3][4].

### 8.4.2. Acceso a los elementos de los arrays bidimensionales

Se puede acceder a los elementos de arrays bidimensionales de igual forma que a los elementos de un array unidimensional. La diferencia reside en que en los elementos bidimensionales deben especificarse los índices de la fila y la columna.

El formato general para asignación directa de valores a los elementos es:

*inserción de elementos*

```
<nombre array>[índice fila][índice columna]=valor elemento;
```

```
extracción de elementos
<variable> = <nombre array> [índice fila][índice columna];
```

Algunos ejemplos de inserciones:

```
Tabla[2][3] = 4.5;
Resistencias[2][4] = 50;
AsientosLibres[5][12] = 5;
```

y de extracción de valores:

```
Ventas = Tabla[1][1];
Dia = Semana[3][6];
```

### 8.4.3. Lectura y escritura de elementos de arrays bidimensionales

Las funciones de entrada o salida se aplican de igual forma a los elementos de un array bidimensional. Por ejemplo,

```
int tabla[3][4];
double resistencias[4][5];

scanf("%d",&tabla[2][3]);
printf("%4d",tabla[1][1]);
scanf("%lf",&resistencias[2][4]);

if (asientosLibres[3][1])
 puts("VERDADERO");
else
 puts("FALSO");
```

### 8.4.4. Acceso a elementos mediante bucles

Se puede acceder a los elementos de arrays bidimensionales mediante bucles anidados. Su sintaxis es:

```
int IndiceFila, IndiceCol;

for (IndiceFila = 0; IndiceFila < NumFilas; ++IndiceFila)
 for (IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
 Procesar elemento[IndiceFila][IndiceCol];
```

---

#### Ejemplo 8.9

*Define una tabla de discos, rellena la tabla con datos de entrada y se muestran por pantalla.*

```
float discos[2][4];
int fila, col;

for (fila = 0; fila < 2; fila++)
{
 for (col = 0; col < 4; col++)
 {
 scanf("%f",&discos[fila][col]);
 }
}

/* Visualizar la tabla */
```

```

for (fila = 0; fila < 2; fila++)
{
 for (col = 0; col < 4; col++)
 {
 printf("\n Pts %.1f \n", discos [fila][col]);
 }
}

```

---

## Ejercicio 8.1

### *Lectura y visualización de un array de dos dimensiones.*

La función leer() lee un array (una tabla) de dos dimensiones y la función visualizar() presenta la tabla en la pantalla.

```

#include <stdio.h>
/* prototipos */
void leer(int a[][5]);
void visualizar(const int a[][5]);

int main()
{
 int a[3][5];
 leer(a);
 visualizar(a);
 return 0;
}

void leer(int a[][5])
{
 int i,j;

 puts("Introduzca 15 números enteros, 3 por fila");
 for (i = 0; i < 3; i++)
 {
 printf("Fila %d: ",i);
 for (j = 0; j < 5; j++)
 scanf("%d",&a[i][j]);
 }
}

void visualizar (const int a[][5])
{
 int i,j;
 for (i = 0; i < 3; i++)
 {
 for (j = 0; j < 5; j++)
 printf(" %d",a[i][j]);
 printf("\n");
 }
}

```

---

### **Ejecución**

La traza (ejecución) del programa:

Introduzca 15 números enteros, 3 por fila

|         |    |    |    |    |    |
|---------|----|----|----|----|----|
| Fila 0: | 45 | 75 | 25 | 10 | 40 |
| Fila 1: | 20 | 14 | 36 | 15 | 26 |
| Fila 2: | 21 | 15 | 37 | 16 | 27 |
|         | 45 | 75 | 25 | 10 | 40 |
|         | 20 | 14 | 36 | 15 | 26 |
|         | 21 | 15 | 37 | 16 | 27 |

### 8.4.5. Arrays de más de dos dimensiones

C proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos del mundo real requieren más de dos o tres dimensiones. El medio más fácil de dibujar un array de tres dimensiones es imaginar un cubo tal como se muestra en la Figura 8.10.

Un array tridimensional se puede considerar como un conjunto de arrays bidimensionales combinados juntos para formar, en profundidad, una tercera dimensión. El cubo se construye con filas (dimensión vertical), columnas (dimensión horizontal) y planos (dimensión en profundidad). Por consiguiente, un elemento dado se localiza especificando su plano, fila y columna. Una definición de un array tridimensional equipos es:

```
int equipos[3][15][10];
```

Un ejemplo típico de un array de tres dimensiones es el modelo *libro*, en el que cada página del libro es un array bidimensional construido por filas y columnas. Así, por ejemplo, cada página tiene cuarenta y cinco líneas que forman las filas del array y ochenta caracteres por línea, que forman las columnas del array. Por consiguiente, si el libro tiene quinientas páginas, existirán quinientos planos y el número de elementos será  $500 \times 80 \times 45 = 1.800.000$ .

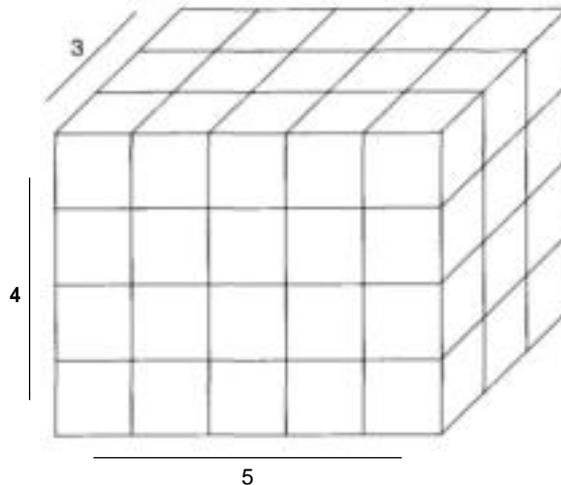


Figura 8.10. Un array de tres dimensiones (4 x 5 x 3).

### 8.4.6. Una aplicación práctica

El array `libro` tiene tres dimensiones `[PAGINAS] [LINEAS] [COLUMNAS]`, que definen el tamaño del array. El tipo de datos del array es `char`, ya que los elementos son caracteres.

¿Cómo se puede acceder a la información del libro? El método más fácil es mediante bucles anidados. Dado que el libro se compone de un conjunto de páginas, el bucle más externo será el bucle de página; y el bucle de columnas el bucle más interno. Esto significa que el bucle de filas se insertará entre los bucles página y columna. El código siguiente permite procesar el array

```
int pagina, linea, columna;

for (pagina = 0; pagina < PAGINAS; ++pagina)
 for (linea = 0; linea < LINEAS; ++linea)
 for (columna = 0; columna < COLUMNAS; ++columna)
 <procesar Libro [pagina][linea][columna]>
```

## Ejercicio 8.2

*Comprobar si una matriz de números enteros es simétrica respecto a la diagonal principal.*

La matriz se genera internamente, con la función `random()` y argumento **N(8)** para que la matriz tenga valores de 0 a 7. El tamaño de la matriz se pide como dato de entrada. La función `simetrica()` determina si la matriz es simétrica. La función `main()` genera matrices hasta encontrar una que sea simétrica y la escribe en pantalla.

```
/*
 Determina si una matriz es simétrica. La matriz se genera con números
 aleatorios de 0 a 7. El programa itera hasta encontrar una matriz
 simétrica.
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define N 8

void genmat(int a[][N], int n);
int simetrica(int a[][N], int n);
void escribemat(int a[][N], int n);

int main(void)
{
 int a[N][N]; /* define matriz de tamaño máximo N */
 int n, i, j;
 int es-sim;

 randomize();
 do {
 printf("\nTamaño de cada dimensión de la matriz, máximo %d: ", N);
 scanf("%d", &n);
 } while (n < 2 || n > N);

 do {
 gen-mat (a, n);
 es-sim = simetrica(a, n);

 if (es-sim)
 {
 puts("\n\Encontrada matriz simétrica.\n");
 escribe_mat(a, n);
 }
 } while (!es_sim);

 return 0;
}
```

```

}
void genmat(int a[][N], int n)
{
 int i,j;

 for (i=0; i<n; i++)
 for (j=0; j<n; j++)
 a[i][j]= random(N);
}

int simetrica(int a[][N], int n)
{
 int i,j;
 int es-simetrica;

 for (es_simetrica=1,i=0; i<n-1 && es-simetrica; i++)
 {
 for (j=i+1; j<n && es-simetrica; j++)
 if (a[i][j] != a[j][i])
 es_simetrica= 0;
 }
 return es-simetrica;
}

void escribemat(int a[][N], int n)
{
 int i,j;
 puts("\tMatriz analizada");
 puts("\t--- - ----~\n");
 for (i=0; i<n; i++)
 { putchar('\t');
 for (j=0; j<n; j++)
 printf("%d %c",a[i][j],(j==n-1 ?'\n ':' '));
 }
}

```

---

## 8.5. UTILIZACIÓN DE ARRAYS COMO PARÁMETROS

En C *todos los arrays se pasan por referencia* (dirección). Esto significa que cuando se llama a una función y se utiliza un array como parámetro, se debe tener cuidado de no modificar los arrays en una función llamada. C trata automáticamente la llamada a la función como si hubiera situado el operador de dirección & delante del nombre del array. La Figura 8.11 ayuda a comprender el mecanismo. Dadas las declaraciones

```

#define MAX 100
double datos[MAX];

```

se puede declarar una función que acepte un array de valores double como parámetro. La función SumaDeDatos() puede tener el prototipo:

```

double SumaDeDatos(double datos[MAX]);

```

Incluso mejor si se dejan los corchetes en blanco y se añade un segundo parámetro que indica el tamaño del array:

```

double SumaDeDatos(double datos[], int n);

```

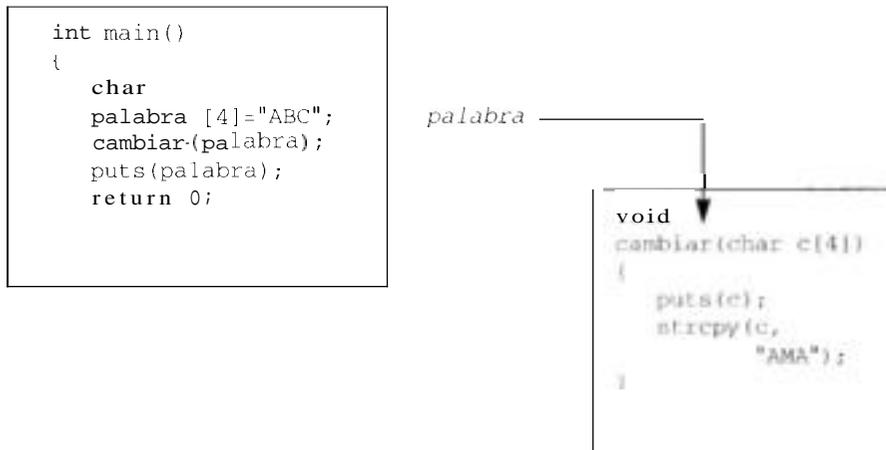


Figura 8.11. Paso de un array por dirección.

A la función `SumaDeDatos` se pueden entonces pasar argumentos de tipo array junto con un entero `n`, que informa a la función sobre cuantos valores contiene el array. Por ejemplo, esta sentencia visualiza la suma de valores de los datos del array:

```
printf("\nSuma = %lf", SumaDeDatos (datos, MAX));
```

La función `SumaDeDatos` no es difícil de escribir. Un simple bucle `for` suma los elementos del array y una sentencia `return` devuelve el resultado de nuevo al llamador:

```
double SumaDeDatos(double datos[], int n)
{
 double suma = 0;
 while (n > 0)
 suma += datos[--n];
 return suma;
}

```

El código que se utiliza para pasar un array a una función incluye el tipo de elemento del array y su nombre. El siguiente ejemplo incluye dos funciones que procesan arrays. En ambas listas de parámetros, el array `a[]` se declara en la lista de parámetros tal como

```
double a[]
```

El número real de elementos se pasa mediante una variable entera independiente. Cuando se pasa un array a una función, se pasa realmente *sólo* la dirección de la celda de memoria donde comienza el array. Este valor se representa por el nombre del array `a`. La función puede cambiar entonces el contenido del array accediendo directamente a las celdas de memoria en donde se almacenan los elementos del array. Así, aunque el nombre del array se pasa por valor, sus elementos se pueden cambiar como si se hubieran pasado por referencia.

## Ejemplo 8.5

*Paso de arrays a funciones. En el ejemplo se lee un array y se escribe.*

El array tiene un tamaño máximo, `L`, aunque el número real de elementos es determinado en la función `leerArray()`. El segundo argumento es, por tanto, un puntero para así poder transmitir por referencia y obtener dicho dato de la función.

```

#include <stdio.h>
#define L 100
void leerArray(double a[], int*);
void imprimirArray (const double [], int);

int main()
{
 double a[L];
 int n;

 leerArray (a, &n);
 printf("El array a tiene %d elementos, estos son\n ",n);
 imprimirArray (a, n);

 return 0;
}

void leerArray(double a[], int* num)
{
 int n = 0;
 puts("Introduzca datos. Para terminar pulsar 0.\n");
 for (; n < L; n++)
 {
 printf("%d: ",n);
 scanf("%lf",&a[n]);
 if (a[n] == 0) break;
 };
 *num = n;
}

void imprimirArray(const double a[],int n)
{
 int i = 0;
 for (; i < n; i++)
 printf("\t%d: %lf\n",i,a[i]);
}

```

### Ejecución

Introduzca datos. Para terminar pulsar 0.

```

0: 31.31
1: 15.25
2: 44.77
3: 0

```

El array tiene tres elementos, éstos son:

```

0: 31.31
1: 15.25
2: 44.77

```

**Ejercicio 8.2**

Escribir una función que calcule el máximo de los primeros  $n$  elementos de un array especificado.

```
double maximo(const double a[],int n)

 double mx;
 int i;
 mx = a[0];
 for (i = 1; i < n; i++)
 mx = (a[i]>mx ? a[i]: mx);

 return mx;
}
```

**8.5.1. Precauciones**

Cuando se utiliza una variable array como argumento, la función receptora puede no conocer cuántos elementos existen en el array. Sin su conocimiento una función no puede utilizar el array. Aunque la variable array puede apuntar al comienzo de él, no proporciona ninguna indicación de donde termina el array.

La función `SumaDeEnteros()` suma los valores de todos los elementos de un array y devuelve el total.

```
int SumaDeEnteros(int *ArrayEnteros)
{
 ...
}

int main()
{
 int lista[5] = {10, 11, 12, 13, 14};
 SumaDeEnteros (lista);
 ...
}
```

Aunque `SumaDeEnteros()` conoce donde comienza el array, no conoce cuántos elementos hay en el array; en consecuencia, no sabe cuántos elementos hay que sumar.

**Se pueden utilizar dos métodos alternativos para permitir que una función conozca el número de argumentos asociados con un array que se pasa como argumento de una función:**

- **situar un valor de señal al final del array, que indique a la función que se ha de detener el proceso en ese momento;**
- **pasar un segundo argumento que indica el número de elementos del array.**

Todas las cadenas utilizan el primer método ya que terminan en nulo. Una segunda alternativa es pasar el número de elementos del array siempre que se pasa el array como un argumento. El array y el número de elementos se convierten entonces en una pareja de argumentos que se asocian con la función llamada. La función `SumaDeEnteros()`, por ejemplo, se puede actualizar así:

```
int SumaDeEnteros(int ArrayEnteros[], int NoElementos)
{
 ...
}
```

El segundo argumento, `NoElementos`, es un valor entero que indica a la función `SumaDeEnteros()` cuantos elementos se procesarán en el array `ArrayEnteros`. Este método suele ser el utilizado para arrays de elementos que no son caracteres.

### Ejemplo 8.6

*Este programa introduce una lista de 10 números enteros y calcula su suma y el valor máximo.*

```
#include <stdio.h>
int SumaDeEnteros(const int ArrayEnteros[], int NoElementos);
int maximo(const int ArrayEnteros[], int NoElementos);

int main()
{
 int items[10];
 int Total, i;

 puts("Introduzca 10 números, seguidos por return");
 for (i = 0; i < 10; i++)
 scanf("%d",&Items[i]);

 printf("Total = %d \n",SumaDeEnteros(Items,10));
 printf("Valor máximo: %d \n",maximo(Items,10));
 return 0;
}

int SumaDeEnteros(const int ArrayEnteros[], int NoElementos)
{
 int i, Total = 0;
 for (i = 0; i < NoElementos; i++)
 Total += ArrayEnteros[i];

 return Total;
}

int maximo(const int ArrayEnteros[], int NoElementos)
{
 int mx;
 int i;

 mx = ArrayEnteros[0];
 for (i = 1; i < NoElementos; i++)
 mx = (ArrayEnteros[i]>mx ? ArrayEnteros[i] : mx);

 return mx;
}
```

El siguiente programa muestra cómo se pasa un array de enteros a una función de ordenación, `ordenar()`.

```
#include <stdio.h>
void ordenar(int[],int); /* prototipo de ordenar */

int main()
{
 int ListaEnt[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 10};
 int i;
 int LongLista = sizeof(ListaEnt) / sizeof(int);

 ordenar(ListaEnt,LongLista);
}
```

```

 for (i = 0; i < LongLista; i++)
 printf("%d ", ListaEnt[i]);
 return 0;
}

void ordenar(int lista[], int numElementos)
{
 /* cuerpo de la función ordenar el array */
}

```

Como C trata las cadenas como arrays de caracteres, las reglas para pasar arrays como argumentos a funciones se aplican también a cadenas. El siguiente ejemplo de una función de cadena que convierte los caracteres de **sus** argumentos a mayúsculas, muestra el paso de parámetros tipo cadena.

```

void convierte_mayus(char cad[])
{
 int i = 0;
 int intervalo = 'a'-'A';
 while (cad[i])
 {
 cad[i] = (cad[i]>='a' && cad[i]<='z') ? cad[i]-intervalo: cad[i];
 i++;
 }
}

```

La función `conviertemayus()` recibe una cadena, un array de caracteres cuyo último carácter es el nulo (0). El bucle termina cuando se alcance el fin de cadena (nulo, condición *false*). La condición del operador ternario determina si el carácter es minúscula, en cuyo caso resta a dicho carácter el intervalo que hay entre las minúsculas y las mayúsculas.

## 8.5.2. Paso de cadenas como parámetros

La técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo utilizan el primer método dado anteriormente para controlar el tamaño de un array. Las cadenas son arrays de caracteres. Cuando una cadena se pasa a una función, tal como `strlen()` (véase capítulo de tratamiento de cadenas), la función conoce que se ha almacenado el final del array cuando ve un valor de 0 en un elemento del array.

Las cadenas utilizan siempre un 0 para indicar que es el último elemento del array de caracteres. Este 0 es el carácter nulo del código de caracteres ASCII.

Considérese estas declaraciones de una constante y una función que acepta un parámetro cadena y un valor de su longitud.

```

#define MAXLON 128
void FuncDemo(char s[], int long);

```

El parámetro `s` es un array de caracteres de longitud no especificada. El parámetro `long` indica a la función cuántos bytes ocupa (que puede ser diferente del número de caracteres almacenados en `s`). Dadas las declaraciones siguientes:

```

char presidente[MAXLON] = "Manuel Martinez";
FuncDemo(presidente, MAXLON);

```

la primera línea declara e inicializa un array de caracteres llamado `presidente`, capaz de almacenar hasta `MAXLON-1` caracteres más un byte de terminación, carácter nulo. La segunda línea, pasa la cadena a la función.

## 8.6. ORDENACIÓN DE LISTAS

La *ordenación* de arrays es otra de las tareas usuales en la mayoría de los programas. La ordenación o clasificación es el procedimiento mediante el cual se disponen los elementos del array en un orden especificado, tal como orden alfabético u orden numérico.

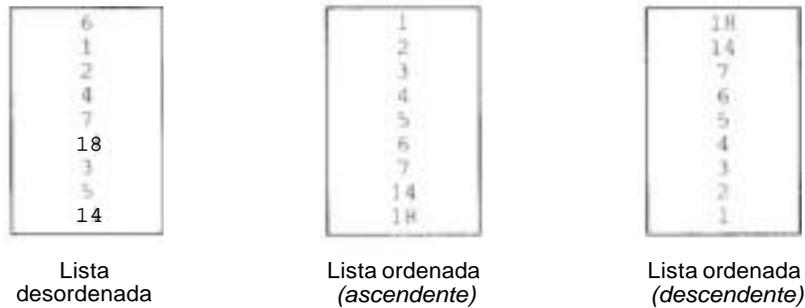


Figura 8.12. Lista de números desordenada y ordenada en orden ascendente y en orden descendente.

Un diccionario es un ejemplo de una lista ordenada alfabéticamente, y una agenda telefónica o lista de cuentas de un banco es un ejemplo de una lista ordenada numéricamente. El orden de clasificación u ordenación puede ser *ascendente* o *descendente*.

Existen numerosos algoritmos de ordenación de arrays: *inserción*, *burbuja*, *selección*, *rápido* (*quick sort*), *fusión* (*merge*), *montículo* (*heap*), *shell*, etc.

### 8.6.1. Algoritmo de la burbuja

La ordenación por burbuja es uno de los métodos más fáciles de ordenación. El método (algoritmo) de ordenación es muy simple. Se compara cada elemento del array con el siguiente (por parejas), si no están en el orden correcto, se intercambian entre sí sus valores. El valor más pequeño *flota* hasta la parte superior del array como si fuera una burbuja en un vaso de refresco con gas.

La Figura 8.13 muestra una lista de números, antes, durante las sucesivas comparaciones y a la terminación del algoritmo de la burbuja. Se van realizando diferentes pasadas hasta que la lista se encuentra ordenada totalmente en orden ascendente.

|                           |    |    |    |    |   |
|---------------------------|----|----|----|----|---|
| <i>Lista desordenada:</i> | 6  | 4  | 10 | 2  | 8 |
| <i>Primera pasada</i>     | 6  | 4  | 4  | 4  |   |
|                           | 4  | 6  | 6  | 6  |   |
|                           | 10 | 10 | 2  | 2  |   |
|                           | 2  | 2  | 10 | 8  |   |
|                           | 8  | 8  | 8  | 10 |   |
| <i>Segunda pasada</i>     | 4  | 4  |    |    |   |
|                           | 6  | 2  |    |    |   |
|                           | 2  | 6  |    |    |   |
|                           | 8  | 8  |    |    |   |
|                           | 10 | 10 |    |    |   |

|                       |    |    |
|-----------------------|----|----|
| <i>Tercera pasada</i> | 4  | 2  |
|                       | 2  | 4  |
|                       | 6  | 6  |
|                       | 8  | 8  |
|                       | 10 | 10 |
| <i>Cuarta pasada</i>  | 2  |    |
|                       | 4  |    |
|                       | 6  |    |
|                       | 8  |    |
|                       | 10 |    |

Figura 8.13. Secuencias de ordenación.

La ordenación de arrays requiere siempre un intercambio de valores, cuando éstos no se encuentran en el orden previsto. Si, por ejemplo, en la primera pasada 6 y 4 no están ordenados se han de intercambiar sus valores. Suponiendo que el array se denomina lista:

```
lista[0] 6
lista[1] 4
lista[2] 10
lista[3] 2
lista[4] 8
```

para intercambiar dos valores, se necesita utilizar una tercera variable auxiliar que contenga el resultado inmediato. Así, por ejemplo, si las dos variables son `lista[0]` y `lista[1]`, el siguiente código realiza el intercambio de dos variables:

```
aux = lista[0];
lista[0] = lista[1];
lista[1] = aux;
```

### Ejemplo 8.7

La función `intercambio` intercambia los valores de **dos** variables `x` e `y`

El algoritmo de intercambio utiliza una variable auxiliar

```
aux = x;
x = y;
y = aux;
```

La función `intercambio` sirve para intercambiar dos elementos `x` e `y` que se pasan a ella. Al tener que pasar por referencia, los argumentos de la función son punteros.

```
void intercambio(float* x, float* y)
{
 float aux;
 aux = *x;
 *x = *y;
 *y = aux;
}
```

Una llamada a esta función:

```
float r, v;

intercambio(&r, &v);
```

**Ejemplo 8.8**

El programa siguiente ordena una lista de números reales y a continuación los imprime.

```
#include <stdio.h>
/* prototipos */
void imprimir(float a[], int n);
void intercambio(float* x, float* y);

void ordenar (float a[], int n);

int main()

 float a[10]={25.5,34.1,27.6,15.24, 3.27, 5.14, 6.21,7.57,4.61, 5.4};

 imprimir(a,10);
 ordenar (a,10);
 imprimir(a,10);
 return 0;
}

void imprimir(float a[], int n)
{
 int i = 0;

 for (; i < n-1; i++) {
 printf("%f,%c",a[i],((i+1)%10==0 ? '\n' : ' '));
 }
 printf("%f \n",a[n-1]);
}

void intercambio(float* x, float* y)
{
 float aux;
 aux = *x;
 *x = *y;
 *y = aux;
}

/* ordenar burbuja */

void ordenar (float a[], int n)
{
 int i,j;
 for (i = n-1; i>0; i--)
 for (j = 0; j < i; j++)
 if (a[j] > a[j+1])
 intercambio(&a[j],&a[j+1]);
}

```

**8.7. BÚSQUEDA EN LISTAS**

Los *arrays* (listas y tablas) son uno de los medios principales por los cuales se almacenan los datos en programas C. Debido a esta causa, existen operaciones fundamentales cuyo tratamiento es imprescindible conocer. Estas operaciones esenciales son: la *búsqueda* de elementos y la ordenación o clasificación de las listas.

La *búsqueda* de un elemento dado en un array (lista o tabla) es una aplicación muy usual en el desarrollo de programas en C. Dos algoritmos típicos **que** realizan esta tarea son la *búsqueda secuencial*

o *en serie* y la *búsqueda binaria* o *dicotómica*. La búsqueda secuencial es el método utilizado para listas no ordenadas, mientras que la búsqueda binaria se utiliza en arrays que ya están ordenados.

### 8.7.1. Búsqueda secuencial

Este algoritmo busca el elemento dado, recorriendo secuencialmente el array desde un elemento al siguiente, comenzando en la primera posición del array y se detiene cuando se encuentra el elemento buscado o bien se alcanza el final del array.

Por consiguiente, el algoritmo debe comprobar primero el elemento almacenado en la primera posición del array, a continuación el segundo elemento y así sucesivamente, hasta que se encuentra el elemento buscado o se termina el recorrido del array. Esta tarea repetitiva se realizará con bucles, en nuestro caso con el bucle **while**.

#### Algoritmo *BusquedaSec*

Se utiliza una variable lógica, en C tipo `int`, denominada *Encontrado*, que indica si el elemento se encontró en la búsqueda. La variable *Encontrado* se inicializa a *falso(0)* y se activa a *verdadero(1)* cuando se encuentra el elemento. Se utiliza un operador **and** (en C **&&**), que permita evaluar las dos condiciones de terminación de la búsqueda: elemento encontrado o no haya más elementos (índice del array excede al último valor válido del mismo).

Cuando el bucle se termina, el elemento o bien se ha encontrado, o bien no se ha encontrado. Si el elemento se ha encontrado, el valor de *Encontrado* será *verdadero* y el valor del índice será la posición del array (índice del elemento encontrado). Si el elemento no se ha encontrado, el valor de *Encontrado* será *falso* y se devuelve el valor `-1` al programa llamador.

#### *BusquedaSec*

```

inicio
 Poner Encontrado = falso
 Poner Indice = primer indice del array
mientras (Elemento no Encontrado) y (Indice < Ultimo) hacer
 si (A[indice] = Elemento) entonces
 Poner Encontrado a Verdadero
 si no
 Incrementar Indice
fin-mientras

si (Encontrado) entonces
 retorno (Indice)
si no
 retorno (-1)
fin-si
€in

```

El algoritmo anterior implementado como una función para un array *Lista* es:

```

enum {FALSE, TRUE};

int BusquedaSec(int Lista[MAX], int Elemento)
{
 int Encontrado = FALSE;
 int i = 0;

 /* Búsqueda en la lista hasta que se encuentra el elemento
 o se alcanza el final de la lista.
 */

```

```

while ((!Encontrado) && (i <= MAX-1))
{
 Encontrado = ((A[i] == Elemento)?TRUE:i++);
}
/*Si se encuentra el elemento se devuelve la posición en la lista. */
if (Encontrado)
 return (i);
else
 return (-1);
}

```

En el bucle `while` se ha utilizado el operado condicional `?:` para asignar `TRUE` si *se* encuentra el elemento, o bien incrementar el índice `i`.

### Ejemplo 8.9

*El siguiente programa busca todas las ocurrencias de un elemento y la posición que ocupa en una matriz. La posición viene dada por fila y columna; la matriz se genera con números aleatorios de 0 a 49.*

La función de búsqueda devuelve 0 si no encuentra al elemento, 1 si lo encuentra. Tiene el argumento de la matriz y dos parámetros para devolver la **fila** y columna, por lo que tendrán que ser de tipo puntero para poder devolver dicha información. La búsqueda se **hará** a partir de la fila y columna de la última coincidencia.

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define F 8
#define C 10
#define N 50

void escribemat(int a[][C]);
void genmat(int a[][C]);
int buscar(int a[][C],int* fila,int* col,int elemento);

int main()
{
 int a[F][C];
 int item,nf,nc,esta;
 int veces = 0;

 randomize();
 genmat(a);
 printf("\n Elemento a buscar: ");
 scanf("%d",&item);

 do {
 esta = buscar(a,&nf,&nc,item);
 if (esta)
 {
 veces = veces+1;
 printf("\n coincidencia %d: Fila %d, Columna %d\n",veces,nf,nc);
 }
 }while (esta);
 escribe_mat(a);
 printf("\nNúmero de coincidencias del elemento %d : %d",

```

```

 item,veces);
 return 0;
I
/* Búsqueda lineal en toda la matriz */
int buscar(int a[][C],int* fila,int* col,int elemento)

 static int x = 0, y = -1;
 int i,j,encontrado;

 /* avanza al siguiente elemento(fila,columna) */
 if (y == C-1) /* ultima columna */
 I
 y = 0;
 x = x+1;
 I
 else
 y = y+1;
 encontrado = 0;
 while (!encontrado && (x < F))
 {
 encontrado = (a[x][y] == elemento);
 if (!encontrado) /* avanza a siguiente elemento */
 if (y == C-1)
 {
 y = 0;
 x = x+1;
 I
 else
 y = y+1;
 }
 /* ultimo valor de x e y */
 *fila = x;
 *col = y;
 return encontrado;
 }
void gen-mat(int a[][C])
{
 int i,j;
 for (i=0; i<F; i++)
 for (j=0; j<C; j++)
 a[i][j]= random(N);
I
void escribemat(int a[][C])
{
 int i,j;
 puts("\t\tMatriz analizada");
 puts("\t\t--- - -----\n");
 for (i=0; i<F; i++)
 { putchar('\t');
 for (j=0; j<C; j++)
 printf("%d %c",a[i][j],(j==C-1 ?'\n ':' '));
 }
I

```

**Ejemplo 8.10**

En este programa se quiere buscar la fila de una matriz real que tiene la máxima suma de sus elementos en valor absoluto, La matriz se genera con números aleatorios, las dimensiones de la matriz se establecen con una constante predefinida.

Para determinar la suma de una fila se define la función `sumar()`, se le pasa la dirección del primer elemento de la fila para tratar cada fila como una array unidimensional. Para generar números aleatorios de tipo real, se divide el número que devuelve la función `rand()` entre 100.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define F 6
#define C 10
#define V 100.0

void escribe_mat (float mt[][C]);
void gen_mat (float mt[][C]);
float sumar(float v[]);
int maximo(float mt[][C]);

int main()
{
 float mat[F][C];
 int fila;

 randomize();
 gen_mat (mat);
 escribe_mat (mat);

 fila = maximo(mat);
 printf("\n\nFila cuya suma de elementos es mayor: %d",fila);
 return 0;
}

void gen_mat (float mat[][C])
{
 int i,j;
 for (i=0; i<F; i++)
 for (j=0; j<C; j++)
 mat[i][j]= rand()/V;
}

void escribe_mat(float mat[][C])
{
 int i,j;
 puts("\n\t\tMatriz analizada\n");
 puts("\t\t--- - -----");
 for (i=0; i<F; i++)
 {
 for (j=0; j<C; j++)
 printf("%.2f%c",mat[i][j],(j==C-1 ?'\n ':' '));
 }
}

float sumar (float v[])
{
 int i;
 float s;
```

```

 for (s=0.0,i=0; i<C; i++)
 s += v[i];
 return s;
}

int maximo(float mt[][C])
{
 float mx;
 int i,f;

 mx = sumar(&mt[0][0]); /* dirección de primera fila */
 printf("\nSuma fila %d %.2f",0,mx);
 for (f=0,i=1; i<F; i++)
 {
 float t;
 t = sumar(&mt[i][0]);
 printf("\nSuma fila %d %.2f",i,t);
 if (t > mx)
 {
 mx = t;
 f = i;
 }
 }
 return f;
}

```

## 8.8. RESUMEN

En este capítulo se analizan los tipos agregados de C *arrays*. Después de leer este capítulo debe tener un buen conocimiento de los conceptos fundamentales de los tipos agregados.

Se describen y analizan los siguientes conceptos:

- Un *array* es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.
- Existen arrays de una dimensión, de dos dimensiones..., y multidimensionales.
- En C los arrays se definen especificando el tipo de dato del elemento, el nombre del array y el tamaño de cada dimensión del array. Para acce-

der a los elementos del array se deben utilizar sentencias de asignación directas, sentencias de lectura/escritura o bucles (mediante las sentencias `for`, `while` o `do-while`).

```
int total_meses[12];
```

- Los arrays de caracteres contienen cadenas de textos. En C se terminan las cadenas de caracteres situando un carácter nulo ("0") como último byte de la cadena.
- C soporta arrays multidimensionales.

```
int ventas_totales[12][50];
```

### 8.9. EJERCICIOS

Para los Ejercicios 8.1 a 8.5, suponga las declaraciones:

```
int i, j, k;
int Primero[21], Segundo[21];
int Tercero[6][12];
```

Determinar la salida de cada segmento de programa (en los casos que se necesite, se indica debajo el archivo de datos de entrada correspondiente).

```
8.1. for (i=1; i<=6; i++)
 scanf("%d",&Primero[i]);
 for(i= 3; i>0; i--)
 printf("%4d",Primero[2*i]);

 3 7 4 - 1 0 6
```

```
8.2. scanf("%d",&k);
 for(i=3; i<=k;)
 scanf("%d",&Segundo[i++]);
 j= 4;
 printf("%d %d\n",Segundo[k],Segundo[j+1]);

 6 3 0 1 9
```

```
8.3. for(i= 0; i<10;i++)
 Primero[i] = i + 3;
 scanf("%d %d",&j,&k);
 for(i= j; i<=k;)
 printf("%d\n",Primero[i++]);

 7 2 3 9
```

```
8.4. for(i=0, i<12; i++)
 scanf("%d",&Primero[i]);
 for(j=0; j<6;j++)
 Segundo[j]=Primero[2*j] + j;
 for(k=3; k<=7,k++)
 printf("%d %d \n",Primero [k+1],Segundo [k-1]);

 2 7 3 4 9 - 4
 6 -5 0 5 -8 1
```

```
8.5. for(j= 0; j<7;)
 scanf("%d",&Primero[j++]);
 i = 0;
 j = 1;
 while ((j< 6) && (Primero[j-1]
 <Primero[j]))
```

```
{
 i++,j++;
}
for(k= -1; k<j+2;)
 printf("%d",Primero[++k]);
.....
20 60 70 10 0 40
30 90
```

```
8.6. for(i= 0; i< 3; i++)
 for(j= 0; j<12; j++)
 Tercero[i][j] = i+j+1;
 for(i= 0; i< 3;i++)
 {
 j = 2;
 while (j < 12)
 {
 printf("%d %d %d \n",i,j,
 Tercero [i][j]);
 j+=3;
 }
 }
```

8.7. Escribir un programa que lea el array

```
4 7 1 3 5
2 0 6 9 7
3 1 2 6 4
```

y lo escriba como

```
4 2 3
7 0 1
1 6 2
3 9 6
5 7 4
```

8.8. Dado el array

```
4 7 - 5 4 9
0 3 -2 6 -2
1 2 4 1 1
6 1 0 3 - 4
```

escribir un programa que encuentre la suma de todos los elementos que no pertenecen a la diagonal principal.

8.9. Escribir una función que intercambie la fila *i-ésima* por la *j-ésima* de un array de dos dimensiones, *mxn*.

### 8.10. PROBLEMAS

**Nota:** todos los programas que se proponen deben hacerse descomponiendo el problema en módulos, que serán funciones en C.

- 8.1. Escribir un programa que convierta un número romano (en forma de cadena de caracteres) en número arábigo.

*Reglas de conversión*

|   |      |
|---|------|
| M | 1000 |
| D | 500  |
| C | 100  |
| L | 50   |
| X | 10   |
| V | 5    |
| I | 1    |

- 8.2. Escribir un programa que permita visualizar el triángulo de Pascal:

```

 1 2 1
 1 3 3 1
 1 4 6 4 1
 1 5 10 1 0 5 1
 1 6 15 20 15 6 1

```

En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un array de una sola dimensión.

- 8.3. Escribir una función que invierta el contenido de  $n$  números enteros. El primero se vuelve el último; el segundo, el penúltimo, etc.
- 8.4. Escribir una función a la cual se le proporcione una fecha (día, mes, año), así como un número de días a añadir a esta fecha. La función calcula la nueva fecha y se visualiza.
- 8.5. Un número entero es primo si ningún otro número primo más pequeño que él es divisor suyo. A continuación escribir un programa que rellene una tabla con los 80 primeros números primos y los visualice.
- 8.6. Escribir un programa que visualice un cuadrado mágico de orden impar  $n$  comprendido entre 3 y 11; el usuario debe elegir el valor de

$n$ . Un cuadrado mágico se compone de números enteros comprendidos entre 1 y  $n$ . La suma de los números que figuran en cada fila, columna y diagonal son iguales.

*Ejemplo*

|   |   |   |
|---|---|---|
| 8 | 1 | 6 |
| 3 | 5 | 7 |
| 4 | 9 | 2 |

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha, y así sucesivamente. El cuadrado es cíclico: la línea encima de la primera es, de hecho, la última y la columna a derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla situada encima del número que acaba de ser situado.

- 8.7. El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. **Sugerencia:** almacenar una lista de palabras en un array y seleccionar palabras aleatoriamente.
- 8.8. Escribir un programa que lea las dimensiones de una matriz, lea y visualice la matriz y a continuación encuentre el mayor y menor elemento de la matriz y sus posiciones.
- 8.9. Si  $x$  representa la media de los números  $x_1, x_2, \dots, x_n$ , entonces la **varianza** es la media de los cuadrados de las desviaciones de los números de la media.

$$\text{Varianza} = \frac{1}{n} \sum_{i=1}^n (x_i - x)^2$$

Y la **desviación estándar** es la raíz cuadrada de la varianza. Escribir un programa que lea una lista de números reales, los cuente y a continuación calcule e imprima su media, varianza y desviación estándar. Utilizar funciones para calcular la media, **varianza** y **desviación estándar**.

- 8.10. Escribir un programa para leer una matriz A y formar la matriz transpuesta de A. El programa debe de escribir ambas matrices.
- 8.11. Escribir una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por -5 y devolver al punto donde fue llamado el vector modificado y el número de entradas modificadas.
- 8.12. Los resultados de las últimas elecciones a alcalde en el pueblo x han sido los siguientes:

| Distrito | Candidato A | Candidato B | Candidato C | Candidato D |
|----------|-------------|-------------|-------------|-------------|
| 1        | 194         | 48          | 206         | 45          |
| 2        | 180         | 20          | 320         | 16          |
| 3        | 221         | 90          | 140         | 20          |
| 4        | 432         | 50          | 821         | 14          |
| 5        | 820         | 61          | 946         | 18          |

Escribir un programa que haga las siguientes tareas:

- a) Imprimir la tabla anterior con cabeceras incluidas.
  - b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos. Asimismo, visualizar el candidato más votado.
  - c) Si algún candidato recibe más del 50 por ciento de los datos, el programa imprimirá un mensaje declarándole ganador.
  - d) Si ningún candidato recibe más del 50 por ciento de los datos, el programa debe imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.
- 8.13. Escribir un programa que lea una colección de cadenas de caracteres de longitud arbitraria. Por cada cadenas leída, su programa hará lo siguiente:
- a) Imprimir la longitud de la cadena.
  - b) Contar el número de ocurrencia de palabras de cuatro letras.
  - c) Sustituir cada palabra de cuatro letras por una cadena de cuatro asteriscos e imprimir la nueva cadena.
- 8.14. Una agencia de venta de vehículos automóviles distribuye quince modelos diferentes y tie-

ne en su plantilla diez vendedores. Se desea un programa que escriba un informe mensual de las ventas por vendedor y modelo, así como el número de automóviles vendidos por cada vendedor y el número total de cada modelo vendido por todos los vendedores. Asimismo, para entregar el premio al mejor vendedor, necesita saber cuál es el vendedor que más coches ha vendido.

| vendedor \ modelo | 1  | 2 | 3  | 4... | 15 |
|-------------------|----|---|----|------|----|
| 1                 | 4  | 8 | 1  |      | 4  |
| 2                 | 12 | 4 | 25 |      | 14 |
| 3                 | 15 | 3 | 4  |      | 7  |
| 10                |    |   |    |      |    |

- 8.15. Diseñar un programa que determine la frecuencia de aparición de cada letra mayúscula en un texto escrito por el usuario (fin de lectura, el punto o el retorno de carro, ASCII 13).
- 8.16. Escribir un programa que lea una cadena de caracteres y la visualice en un cuadro.
- 8.17. Escribir un programa que lea una frase, sustituya todas las secuencias de dos o más blancos por un solo blanco y visualice la frase restante.
- 8.18. Escribir un programa que lea una frase y a continuación visualice cada palabra de la frase en columna, seguido del número de letras que compone cada palabra.
- 8.19. Escribir un programa que desplace una palabra leída del teclado desde la izquierda hasta la derecha de la pantalla.
- 8.20. Escribir un programa que lea una línea de caracteres, y visualice la línea de tal forma que las vocales sean sustituidas por el carácter que más veces se repite en la línea.
- 8.21. Escribir un programa que calcule la frecuencia de aparición de las vocales de un texto proporcionado por el usuario. Esta solución debe presentarse en forma de histograma.

- 8.22.** Escribir un programa que lea una serie de cadenas, a continuación determine si la cadena es un identificador válido según la sintaxis de C. **Sugerencias:** utilizar las siguientes funciones: longitud (tamaño del identificador en el rango permitido); primero (determinar si el nombre comienza con un símbolo permitido); restantes (comprueba si los restantes son caracteres permitidos).
- 8.23.** Escriba una función **sort** que ordene un conjunto de  $n$  cadenas en orden alfabético.
- 8.24.** Diseñar un programa que determine la media del número de horas trabajadas durante todos los días de la semana para cada uno de los empleados de la Universidad.
- 8.25.** Escriba una función que ordene en sentido descendente los  $n$  primeros elementos de un **array** de cadenas basado en las longitudes de las cadenas. Por ejemplo, 'bibi' vendrá antes que 'Ana'.
- 8.26.** Se introduce una frase por teclado. Se desea imprimir cada palabra de la frase en líneas diferentes y consecutivas.
- 8.27.** Escribir un programa que determine si una frase o una palabra es un palíndromo. Un palíndromo es una cadena de caracteres que se lee de igual forma en ambos sentidos; por ejemplo: ana.
- 8.28.** Escribir un programa que tenga como entrada una palabra y  $n$  líneas. Se quiere determinar el número de veces que se encuentra la palabra en las  $n$  líneas.
- 8.29.** Se dice que una matriz tiene un **punto** de silla si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales, y calcule la posición de un **punto** de silla (si es que existe).
- 8.30.** Escribir un programa en el que se genere aleatoriamente un vector de 20 números enteros. El vector ha de quedar de tal forma que la suma de los 10 primeros elementos sea mayor que la suma de los 10 últimos elementos. Mostrar el vector original y el vector con la distribución indicada.

## CAPÍTULO 9

# ESTRUCTURAS Y UNIONES

---

### CONTENIDO

- |                                                            |                                 |
|------------------------------------------------------------|---------------------------------|
| <b>9.1.</b> Estructuras.                                   | <b>9.6.</b> Uniones.            |
| <b>9.2.</b> Acceso a estructuras.                          | <b>9.7.</b> Enumeraciones.      |
| <b>9.3.</b> Estructuras anidadas.                          | <b>9.8.</b> Campos de bit.      |
| <b>9.4.</b> Arrays de estructuras.                         | <b>9.9.</b> <i>Resumen.</i>     |
| <b>9.5.</b> Utilización de estructuras<br>como parámetros. | <b>9.10.</b> <i>Ejercicios.</i> |
|                                                            | <b>9.11.</b> <i>Problemas.</i>  |

---

## INTRODUCCIÓN

Este capítulo examina estructuras, uniones, enumeraciones y tipos definidos por el usuario que permite a un programador crear nuevos tipos de datos. La capacidad para **crear** nuevos tipos es una característica importante y potente de C y libera a un programador de restringirse al uso de los tipos ofrecidos por el lenguaje. Una **estructura** contiene múltiples variables, que pueden ser de tipos diferentes. La estructura es importante para la creación de programas potentes, tales como bases de datos u otras aplicaciones que requieran grandes cantidades de datos. Por otra parte, se analizará el concepto de **unión**, otro tipo de dato no tan importante como las estructuras array y estructura, pero si necesarias en algunos *casos*.

Un tipo de dato enumerado es una colección de miembros con nombre que tienen valores enteros equivalentes. Un **typedef** es de hecho no un nuevo tipo de dato sino simplemente un sinónimo de un tipo existente.

---

## CONCEPTOS CLAVE

- Estructura.
- Estructuras anidadas.
- Selector de campos.
- **struct**.
- **sizeof**.
- **union**.
- **typedef**.
- Operadores de bits,

## 9.1. ESTRUCTURAS

Los arrays son estructuras de datos que contienen un número determinado de elementos (**su tamaño**) y todos los elementos han de ser del mismo tipo de datos; es una estructura de datos homogénea. Esta característica supone una gran limitación cuando se requieren grupos de elementos con tipos diferentes de datos cada uno. Por ejemplo, si se dispone de una lista de temperaturas, es muy útil un array; sin embargo, si se necesita una lista de información de clientes que contengan elementos tales como el nombre, la edad, la dirección, el número de la cuenta, etc., los arrays no son adecuados. La solución a este problema es utilizar un tipo de dato registro, en C llamado *estructura*.

Los componentes individuales de una estructura se llaman *miembros*. Cada miembro (elemento) de una estructura puede contener datos de un tipo diferente de otros miembros. Por ejemplo, se puede utilizar una estructura para almacenar diferentes tipos de información sobre una persona, tal como nombre, estado civil, edad y fecha de nacimiento. Cada uno de estos elementos se denominan *nombre del miembro*.

**Una estructura es una colección de uno o más tipos de elementos denominados miembros, cada uno de los cuales puede ser un tipo de dato diferente.**

Una estructura puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado *nombre* del miembro. Supongamos que se desea almacenar los datos de una colección de discos compactos (**CD**) de música. Estos datos pueden ser:

- Título.
- Artista.
- Número de canciones.
- Precio.
- Fecha de compra.

La estructura CD contiene cinco miembros. Tras decidir los miembros, se debe decidir cuáles son los tipos de datos para utilizar por los miembros. Esta información se representa en la tabla siguiente:

| Nombre miembro      | Tipo de dato                      |
|---------------------|-----------------------------------|
| Título              | Array de caracteres de tamaño 30. |
| Artista             | Array de caracteres de tamaño 25. |
| Número de canciones | Entero.                           |
| Precio              | Coma flotante.                    |
| Fecha de compra     | Array de caracteres de tamaño 8.  |

La Figura 9.1 contiene la estructura CD, mostrando gráficamente los tipos de datos dentro de la estructura. Obsérvese que cada miembro es un tipo de dato diferente.

|                     |                                   |
|---------------------|-----------------------------------|
| Título              | Ay, ay, ay, cómo se aleja el sol. |
| Artista             | NO me pises la sandalias.         |
| Número de canciones | 10                                |
| Precio              | 2222.25                           |
| Fecha de compra     | 8-10-1999                         |

**Figura 9.1.** Representación gráfica de una estructura CD.

### 9.1.1. Declaración de una estructura

Una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar. El formato de la declaración es:

```
struct <nombre de la estructura>
{
 <tipo de dato miembro> <nombre miembro>
 <tipo de dato miembro> <nombre miembro>
 ...
 <tipo de dato miembro.> <nombre miembro>
};
```

La declaración de la estructura CD es

```
struct coleccion_CD
{
 char titulo[30];
 char artista[25];
 int num-canciones;
 float precio;
 char fecha-compra[8];
}
```

---

#### Ejemplo

```
struct complejo
{
 float parte-real, parte-imaginaria;
};
```

En este otro ejemplo se declara el tipo estructura venta ;

```
struct venta
{
 char vededor[30];
 unsigned int codigo;
 int inids_articulos;
 float precio-unit;
};
```

---

### 9.1.2. Definición de variables de estructuras

Al igual que a los tipos de datos enumerados, a una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura. Del mismo modo que sucede en otras situaciones, en C existen dos conceptos similares a considerar, *declaración y definición*. La diferencia técnica es la siguiente, una declaración especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria; la declaración especifica un nuevo tipo de dato: `struct <nombre-estructura>`. Por consiguiente, cada definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado.

Las variables de estructuras se pueden definir de dos formas: 1) listándolas inmediatamente después de la llave de cierre de la declaración de la estructura, o 2) listando el tipo de la estructura creado seguida por las variables correspondientes en cualquier lugar del programa antes de utilizarlas. La definición y declaración de la estructura `colecciones-CD` se puede hacer por cualquiera de los dos métodos:

```

1. struct colecciones-CD
 {
 char titulo[30];
 char artista[25];
 int num-canciones;
 float precio;
 char fecha_compra[8];
 } cd1, cd2, cd3;

2. struct colecciones_CD cd1, cd2, cd3;

```

### Otros ejemplos de definición/declaración

Considérese un programa que gestione libros y procese los siguientes datos: título del libro, nombre del autor, editorial y año de publicación. Una estructura `info-libro` podría ser:

```

struct info_libro
{
 char titulo[60];
 char autor[30];
 char editorial[30];
 int anyo;
};

```

La definición de la estructura se puede hacer así:

```

1. struct info_libro
 {
 char titulo[60];
 char autor[30];
 char editorial[30];
 int anyo;
 } librol, libro2, libro3;

2. struct info_libro librol, libro2, libro3;

```

Ahora se nos plantea una aplicación de control de los participantes en una carrera popular, cada participante se representa por los datos: nombre, edad, sexo, categoría, club y tiempo. El registro se representa con la estructura `corredor`:

```

struct corredor
{
 char nombre[40];
 int edad;
 char sexo;
 char categoria[20];
 char club[26];
 float tiempo;
};

```

La definición de variables estructura se puede hacer así:

```

struct corredor vl, sl, cl;

```

### 9.1.3. Uso de estructuras en asignaciones

Como una estructura es un tipo de dato similar a un `int` o un `char`, se puede asignar una estructura a otra. Por ejemplo, se puede hacer que `libro3`, `libro4` y `libro5` tengan los mismos valores en sus miembros que `librol`. Por consiguiente, sería necesario realizar las siguientes sentencias:

```

libro3 = libro1;
libro4 = libro1;
libro5 = libro1;

```

De modo alternativo se puede escribir

```

libro4 = libro5 = libro6 = libro1;

```

#### 9.1.4. Inicialización de una declaración de estructuras

Se puede inicializar una estructura de dos formas. Se puede inicializar una estructura dentro de la sección de código de su programa, o bien se puede inicializar la estructura como parte de la definición. Cuando se inicializa una estructura como parte de la definición, se especifican los valores iniciales, entre llaves, después de la definición de variables estructura. El formato general en este caso:

```

struct <tipo> <nombre variable estructura> =
{ valor miembro,
 valor miembro ,
 ...
 valor miembro
};

struct info-libro
{
 char titulo[60];
 char auto[30];
 char editorial[30];
 int anyo;
} libro1 = {"Maravilla del saber ", "Lucas Garcia", "McGraw-Hill", 1999);

```

Otro ejemplo podría ser:

```

struct coleccion_CD
{
 char titulo[30];
 char artista[25];
 int num-canciones;
 float precio;
 char fecha_compra[8];
} cd1 = {
 "El humo nubla tus ojos",
 "Col Porter",
 15,
 2545,
 "02/6/99"
};

```

Otro ejemplo con la estructura corredor:

```

struct corredor vl = {
 "Salvador Rapido",
 29,
 'V',
 "Senior",
 'Independiente',
 0.0
};

```

### 9.1.5. El tamaño de una estructura

El operador `sizeof` se aplica sobre un tipo de datos, o bien sobre una variable. Se puede aplicar para determinar el tamaño que ocupa en memoria una estructura. El siguiente programa ilustra el uso del operador `sizeof` para determinar el tamaño de una estructura:

```
#include <stdio.h>

/* declarar una estructura Persona */
struct persona
{
 char nombre[30];
 int edad;
 float altura;
 float peso;
};

void main()
{
 struct persona mar;
 printf("Sizeof (persona): %d \n", sizeof (mar));
}
```

Al ejecutar el programa se produce la salida:

```
Sizeof (persona) : 40
```

El resultado se obtiene determinando el número de bytes que ocupa la estructura

| Persona      | Miembros dato | Tamaño (bytes) |
|--------------|---------------|----------------|
| nombre[30]   | char(1)       | 30             |
| edad         | int(2)        | 2              |
| altura       | float(4)      | 4              |
| peso         | float(4)      | 4              |
| <i>Total</i> |               | <b>40</b>      |

## 9.2. ACCESO A ESTRUCTURAS

Cuando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información de la estructura. Se puede acceder a los miembros de una estructura de una de estas dos formas: 1) utilizando el operador punto (`.`), o bien 2) utilizando el operador puntero `->`.

### 9.2.1. Almacenamiento de información en estructuras

Se puede almacenar información en una estructura mediante inicialización, asignación directa o lectura del teclado. El proceso de inicialización ya se ha examinado, veamos ahora la asignación directa y la lectura del teclado.

#### *Acceso a una estructura de datos mediante el operador punto*

La asignación de datos a los miembros de una variable estructura se hace mediante el operador punto. La sintaxis en C es:

```
<nombre variable estructura> . <nombre miembro> = datos;
```

Algunos ejemplos:

```
strcpy(cd1.titulo,"Granada");
cd1.precio = 3450.75;
cd1.num_canciones = 7;
```

El operador punto proporciona el camino directo al miembro correspondiente. Los datos que se almacenan en un miembro dado deben ser del mismo tipo que el tipo declarado para ese miembro. En el siguiente ejemplo se lee del teclado los datos de una variable estructura corredor:

```
struct corredor cr;
printf("Nombre: ");
gets(cr.nombre);
printf("edad: ");
scanf("%d",&cr.edad);
printf("Sexo: ");
scanf("%c",&cr.sexo);
printf("Club: ");
gets(cr.club);
if (cr.edad <= 18)
 cr.categoria = "Juvenil";
elseif (cr.edad <= 40)
 cr.categoria = "Senior";
else
 cr.categoria = "Veterano";
```



### ***Acceso a una estructura de datos mediante el operador puntero***

El operador puntero, `->`, sirve para acceder a los datos de la estructura a partir de un puntero. Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura. A continuación, utilice simplemente el operador puntero para apuntar a un miembro dado.

La asignación de datos a estructuras utilizando el operador puntero tiene el formato:

```
<puntero estructura> -> <nombre miembro> = datos;
```

Así, por ejemplo, una estructura estudiante

```
struct estudiante
{
 char Nombre[41];
 int Num_Estudiente;
 int Anyo-de-matricula;
 float Nota;
};
```

Se puede definir `ptr_est` como un puntero a la estructura

```
struct estudiante *ptr_est;
struct estudiante mejor;
```

A los miembros de la estructura estudiante se pueden asignar datos como sigue (siempre y cuando la estructura ya tenga creado su espacio de almacenamiento, por ejemplo, con `malloc()`; o bien, tenga la dirección de una variable estructura).

```
ptr_est = &mejor; /* ptr_est tiene la dirección(apunta a) mejor */
strcpy(ptr_est->Nombre, "Pepe alomdra");
ptr_est -> Num_Estudiente = 3425;
ptr_est -> Nota = 8.5;
```

## Nota

Previamente habría que crear espacio de almacenamiento en memoria; por ejemplo, con la función `malloc()`.

### 9.2.2. Lectura de información de una estructura

Si ahora se desea introducir la información en la estructura basta con acceder a los miembros de la estructura con el operador punto o flecha (puntero). Se puede introducir la información desde el teclado o desde un archivo, o asignar valores calculados.

Así, si `z` es una variable de tipo estructura complejo, se lee parte real, parte imaginaria y se calcula el módulo:

```
struct complejo
{
 float pr;
 float pi;
 float modulo;
};
struct complejo z;

printf("\nParte real: ");
scanf("%f",&z.pr);
printf("\nParte imaginaria: ");
scanf("%f",&z.pi);
/* calculo del módulo */
z.modulo = sqrt(z.pr*z.pr + z.pi*z.pi);
```

### 9.2.3. Recuperación de información de una estructura

Se recupera información de una estructura utilizando el operador de asignación o una sentencia de salida (`printf()`, `puts()`, ...). Igual que antes, se puede emplear el operador punto o el operador flecha (puntero). El formato general toma uno de estos formatos:

1. *<nombre variable> = <nombre variable estructura>.<nombre miembro>;*

o bien

*<nombre variable> = <puntero de estructura> -> <nombre miembro>;*

2 .para salida:

*printf(" ",<nombre variable estructura>.<nombre miembro>);*

o bien

*printf(" ",<puntero de estructura>-> <nombre miembro>);*

Algunos ejemplos del uso de la estructura complejo:

```
float x,y;
struct complejo z;
struct complejo *pz;
```

```

pz = &z;
x = z.pr;
y = z.pi;
...
printf("\nNúmero complejo (%.1f,%.1f), módulo: %.2f",
 pz->pr,pz->pi,pz->modulo);

```

### 9.3. ESTRUCTURAS ANIDADAS

Una estructura puede contener otras estructuras llamadas *estructuras anidadas*. Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes sólo una vez en su propia estructura y a continuación utilizar esa estructura como un miembro de otra estructura. Consideremos las siguientes dos definiciones de estructuras:

```

struct empleado
{
 char nombre-emp[30];
 char direccion[25];
 char ciudad[20];
 char provincia[20];
 long int cod-postal;
 double salario;
};

struct clientes

 char nombre_cliente[30];
 char direccion[25];
 char ciudad[20];
 char provincia[20];
 long int cod-postal;
 double saldo;
};

```

Estas estructuras contienen muchos datos diferentes, aunque hay datos que están solapados. Así, se podría disponer de una estructura, `info_dir`, que contuviera los miembros comunes.

```

struct info-dir

 char direccion[25];
 char ciudad[20];
 char provincia[20];
 long int cod-postal;
};

```

Esta estructura se puede utilizar como un miembro de las otras estructuras, es decir, *anidarse*.

```

struct empleado
{
 char nombre-emp[30];
 struct info-dir direccion_emp;
 double salario;
};

struct clientes
{

```

```

char nombre_cliente[30];
struct info-dir direccion_clien;
double saldo;
};

```

Gráficamente se podrían mostrar estructuras anidadas en la Figura 9.2.

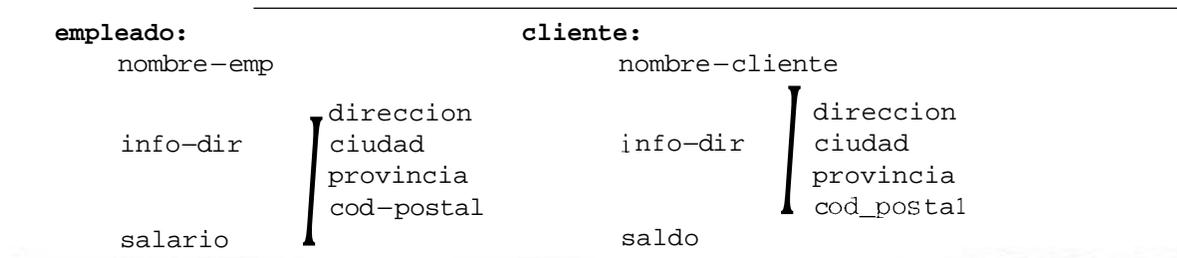


Figura 9.2. Estructuras anidadas.

### 9.3.1. Ejemplo de estructuras anidadas

Se desea diseñar una estructura que contenga información de operaciones financieras. Esta estructura debe constar de un número de cuenta, una cantidad de dinero, el tipo de operación (depósito=0, retirada de fondos=1, puesta al día=2 o estado de la cuenta=3) y la fecha y hora en que la operación se ha realizado. A fin de realizar el acceso correcto a los campos día, mes y año, así como el tiempo (la hora y minutos) en que se efectuó la operación, se define una estructura fecha y una estructura tiempo. La estructura registro-operacion tiene como miembros una variable (un campo) de tipo fecha, otra variable del tipo tiempo y otras variables para representar los otros campos. La representación del tipo de operación se hace con una variable entera, aunque el tipo apropiado es un tipo enumerado (descrito en siguientes apartados). A continuación se declara estos tipos, se escribe una función que lee una operación financiera y devuelve la operación leída. La fecha y hora es captada del sistema.

```

#include <stdio.h>
#include <dos.h>

struct registro-operation entrada();
struct fecha
{
 unsigned int mes, dia, anyo;
};
struct tiempo
{
 unsigned int horas, minutos;
};
struct registro-operacion
{
 long numero-cuenta;
 float cantidad;
 int tipo-operacion;
 struct fecha f;
 struct tiempo t;
};

int main()
{
 struct registro-operacion w;

```

```

 w = entrada();

 printf("\n\n Operación realizada\n ");
 printf("\t%d\n",w.numero-cuenta);
 printf("\t%d-%d-%d\n",w.f.dia,w.f.mes,w.f.anyo);
 printf("\t%d:%d\n",w.t.horas,w.t.minutos);

 return 0;
}

struct registro-operacion entrada()
{
 struct time t;
 struct date d;
 struct registro-operacion una;
 printf("\nNúmero de cuenta: ");
 scanf("%d",&una.numero_cuenta);
 puts("\n\tTipo de operación");
 puts("Deposito(0)");
 puts("Retirada de fondos(1)");
 puts("Puesta al día(2)");
 puts("Estado de la cuenta(3)");
 scanf("%d",&una.tipo_operacion);

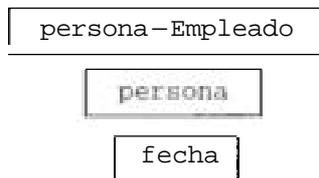
 /* Fecha y tiempo del sistema */
 gettime(&t);
 una.t.horas = t.ti_hour;
 una.t.minutos = t.ti_min;

 getdate(&d);
 una.f.anyo = d.da_year;
 una.f.mes = d.da_mon;
 una.f.dia = d.da-day;
 return una;
}

```

## Ejercicio 9.1

Se desea registrar una estructura `PersonaEmpleado` que contenga como miembros los datos de una persona empleado que a su vez tenga los datos de la fecha de nacimiento. En un programa se muestra el uso de la estructura, se define una función para dar entrada a los datos de la estructura y otra función para dar salida a los datos de una estructura persona. A la función de entrada se transmite por dirección (&p) la variable estructura, por lo que el argumento correspondiente tiene que ser un puntero(\*p) y el acceso a los campos se hace con el selector ->



```

#include <stdio.h>
struct fecha
{
 unsigned int dia, mes, anyo;
};

```

```

struct persona {
 char nombre[20];
 unsigned int edad;
 int altura;
 int peso;
 struct fecha fec;
};

struct persona-empleado
{
 struct persona unapersona;
 unsigned int salario;
 unsigned int horas_por_semana;
};

/* prototipos de funciones */

void entrada(struct persona-empleado *p);
void muestra(struct persona-empleado up);

void main()
{
 /* define una variable persona-empleado */
 struct persona-empleado p;

 /* llamada a entrada() transmitiendo la direccion */
 entrada(&p);

 /* salida de los datos almacenados */
 muestra(p);
}

void entrada(struct persona-empleado *p)
{
 printf("\nIntroduzca su nombre: ");
 gets(p->unapersona.nombre);
 printf ("introduzca su edad: ");
 scanf("%d",&p->unapersona.edad);
 printf ("Introduzca su altura: ");
 scanf("%d",&p->unapersona.altura);
 printf ('Introduzca su peso: ');
 scanf("%d",&p->unapersona.peso);
 printf("Introduzca su fecha de nacimiento: ");
 scanf("%d %d %d",&p->unapersona.fec.dia,
 &p->unapersona.fec.mes,
 &p->unapersona.fec.anyo);
 printf ("Introduzca su salario:");
 scanf("%d",&p->salario);
 printf ("introduzca numero de horas:");
 scanf ("%d",&p->horas_por_semana);
}

void muestra(struct persona-empleado up)
{
 puts("\n\n\tDatos de un empleado");
 puts("\n\n\t-----");
 printf("Nombre: %s \n",up.unapersona.nombre);
 printf("Edad: %d \n",up.unapersona.edad);
 printf("fecha de nacimiento: %d-%d-%d\n",

```

```

 up.unapersona.fec.dia,
 up.unapersona.fec.mes,
 up.unapersona.fec.anyo);
printf("Altura: %d \n", up.unapersona.altura);
printf("Peso: %d \n", up.unapersona.peso);
printf("Numero de horas: %d \n", up.horas_por_semana);

```

1

**El acceso a miembros dato de estructuras anidadas requiere el uso de múltiples operadores punto.**

**Ejemplo:** acceso al día del mes de la fecha de nacimiento de un empleado.

```
up.unapersona.fec.dia
```

Las estructuras se pueden anidar a cualquier grado. También es posible inicializar estructuras anidadas en la definición. El siguiente ejemplo inicializa una variable Luis de tipo `struct persona`.

```
struct persona Luis { "Luis", 25, 1940, 40, {12, 1, 70}};
```

## 9.4. ARRAYS DE ESTRUCTURAS

Se puede crear un array de estructuras tal como se crea un array de otros tipos. Los arrays de estructuras son idóneos para almacenar un archivo completo de empleados, un archivo de inventario, o cualquier otro conjunto de datos que se adapte a un formato de estructura. Mientras que los arrays proporcionan un medio práctico de almacenar diversos valores del mismo tipo, los arrays de estructuras le permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras.

Muchos programadores de C utilizan arrays de estructuras como un método para almacenar datos en un archivo de disco. Se pueden introducir y calcular sus datos de disco en arrays de estructuras y a continuación almacenar esas estructuras en memoria. Los arrays de estructuras proporcionan también un medio de guardar datos que se leen del disco.

La declaración de un array de estructuras `info-libro` se puede hacer de un modo similar a cualquier array, es decir,

```
struct info-libro libros[100];
```

asigna un array de 100 elementos denominado `libros`. Para acceder a los miembros de cada uno de los elementos estructura se utiliza una notación de array. Para inicializar el primer elemento de `libros`, por ejemplo, su código debe hacer referencia a los miembros de `libros[0]` de la forma siguiente:

```
strcpy(libros[0].titulo, "C++ a su alcance");
strcpy(libros[0].autor, "Luis Joyanes");
strcpy(libros[0].editorial, "McGraw-Hill");
libros[0].anyo = 1999;
```

También puede inicializarse un array de estructuras en el punto de la declaración encerrando la lista de inicializadores entre llaves, ( ). Por ejemplo,

```
struct info-libro libros[3] = { "C++ a su alcance", "Luis Joyanes",
 "McGraw-Hill", 1999, "Estructura de datos", "Luis Joyanes",
 "McGraw-Hill", 1999, "Problemas en Pascal", "Angel Hermoso",
 "McGraw-Hill", 19971;
```

En el siguiente ejemplo se declara una estructura que representa a un número racional, un array de números racionales es inicializado con valores al azar.

```

struct racional
{
 int N,
 int D;
};
struct racinal rs[4] = { 1,2, 2,3, -4,7, 0,1};

```

### 9.4.1. Arrays como miembros

Los miembros de las estructuras puede ser asimismo arrays. En este caso, será preciso extremar las precauciones cuando se accede a los elementos individuales del array.

Considérese la siguiente definición de estructura. Esta sentencia declara un array de 100 estructuras, cada estructura contiene información de datos de empleados de una compañía.

```

struct nomina
{
 char nombre[30];
 int dependientes;
 char departamento[10];
 float horas_dias[7]; /* array de tipo float */
 float salario;
} empleado[100]; /* Un array de 100 empleados */

```

---

#### Ejemplo 9.1

Una librería desea catalogar su inventario de libros. El siguiente programa crea un array de 100 estructuras, donde cada estructura contiene diversos tipos de variables, incluyendo arrays.

```

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct inventario
{
 char titulo[25];
 char fecha_pub[20];
 char autor[30];
 int num;
 int pedido;
 float precio-venta;
};

int main()
{
 struct inventario libro[100];
 int total = 0;
 char resp, b[21];

 do {
 printf("Total libros %d \n", (total+1));
 printf("¿Cuál es el título?: ");
 gets(libro[total].titulo);

 printf("¿Cuál es la fecha de publicación?: ");
 gets(libro[total].fecha_pub);
 }

```

```

printf("¿Quién es el autor?");
gets(libro[total].autor);

printf("¿Cuántos libros existen?: ");
scanf("%d",&libro[total].num);

printf("¿Cuántos ejemplares existen pedidos?: ");
scanf("%d",&libro[total].pedido);

printf("¿Cuál es el precio de venta?: ");
gets(b);
libro[total].precio_venta = atof(b); /* conversión a real */
fflush(stdin);

printf("\n ¿Hay más libros? (S/N)");
scanf("%c",&resp);
fflush(stdin);
resp = toupper(resp); /* convierte a mayúsculas */
if (resp == 'S')
{
 total++;
 continue;
}
} while (resp == 'S');
return 0;
)

```

## 9.5. UTILIZACIÓN DE ESTRUCTURAS COMO PARÁMETROS

C permite pasar estructuras a funciones, bien por valor o bien por referencia utilizando el operador &. Si la estructura es grande, el tiempo necesario para copiar un parámetro struct a la pila puede ser prohibitivo. En tales casos, se debe considerar el método de pasar la dirección de la estructura.

El listado siguiente muestra un programa que pasa la dirección de una estructura a una función para entrada de datos. La misma variable estructura la pasa por valor a otra función para salida de los campos.

```

#include <stdio.h>

/* Define el tipo estructura info_persona */
struct info-persona {
 char nombre[20];
 char calle[30];
 char ciudad[25];
 char provincia[25];
 char codigopostal[6];
};

/* prototipos de funciones */
void entrad_pna(struct info_persona* pp);
void ver_info(struct info_persona p);

void main(void)
{
 struct info-persona reg-dat;
 /* Pasa por referencia la variable */
 entrad_pna(®_dat);
 /* Pasa por valor */
 ver_info(reg_dat);
}

```

```

printf("\nPulsa cualquier carácter para continuar\n");
getchar();
}

void entrada_pna(struct info_persona* pp)
{
 puts("\n Entrada de los datos de una persona\n");
 /* Para acceder a los campos se utiliza el selector -> */
 printf ("Nombre: "); gets(pp->nombre);
 printf ("Calle: "); gets(pp->calle);
 printf ("Ciudad: "); gets(pp->ciudad);
 printf ("Provincia: "); gets(pp->provincia);
 printf ("Código postal: "); gets(pp->codigopostal);
}

void ver_info(struct info_persona p)
{
 puts("\n\tInformación relativa a la persona");
 puts(p.nombre);
 puts(p.calle);
 puts(p.ciudad);
 puts(p.provincia);
 puts(p.codigopostal);
}

```

**Si se desea pasar la estructura por referencia, necesita situar un operador de referencia & antes de reg-dat en la llamada a la función entrada\_pna(). El parámetro correspondiente debe de ser tipo puntero struct info\_persona\* pp. El acceso a miembro de dato de la estructura a partir de un puntero requiere el uso del selector ->.**

## 9.6. UNIONES

Las uniones son similares a las estructuras en cuanto que agrupa a una serie de variables, pero la forma de almacenamiento es diferente y, por consiguiente, efectos diferentes. Una estructura (`struct`) permite almacenar variables relacionadas juntas y almacenadas en posiciones contiguas en memoria. Las uniones, declaradas con la palabra reservada `union`, almacenan también miembros múltiples en un paquete; sin embargo, en lugar de situar sus miembros unos detrás de otros, en una unión, todos los miembros se solapan entre sí en la misma posición. El tamaño ocupado por una unión se determina así: es analizado el tamaño de cada variable de la unión, el mayor tamaño de variable será el tamaño de la unión. La sintaxis de una unión es la siguiente:

```

union nombre {
 tipo1 miembro1;
 tipo2 miembro2;
 ...
};

```

Un ejemplo:

```

union PruebaUnion
{
 float Item1;
 int Item2;
};

```

La cantidad de memoria reservada para una unión es igual a la anchura de la variable más grande. En el tipo `union`, cada uno de los miembros dato comparten memoria con los otros miembros de la unión. La cantidad total de memoria utilizada por la unión `comparte` es de 8 bytes, ya que el elemento `double` es el miembro dato mayor de la unión.

```
union comparte
{
 char letra;
 int elemento;
 float precio;
 double z;
};
```

Una razón para utilizar una unión es ahorrar memoria. En muchos programas se deben tener varias variables, pero no necesitan utilizarse todas al mismo tiempo. Considérese la situación en que se necesitan tener diversas cadenas de caracteres de entrada. Se pueden crear varios arrays de cadenas de caracteres, tales como las siguientes:

```
char linea_ordenes[80];
char mensaje_error[80];
char ayuda[80];
```

Estas tres variables ocupan 240 bytes de memoria. Sin embargo, si su programa no necesita utilizar las tres variables simultáneamente, ¿por qué no permitirle compartir la memoria utilizando una unión? Cuando se combinan en el tipo `union` `frases`, estas variables ocupan un total de sólo 80 bytes.

```
union frases {
 char linea_ordenes[80];
 char mensaje_error[80];
 char ayuda[80];
} cadenas, *pc;
```

Para referirse a los miembros de una unión, se utiliza el operador punto (`.`), o bien el operador `->` si se hace desde un puntero a unión. **Así:**

```
cadenas.ayuda;
cadenas.mensaje_error;
pc -> mensaje_error;
```

## 9.7. ENUMERACIONES

Un **enum** es un tipo definido por el usuario con constantes de nombre de tipo entero. En la declaración de un tipo `enum` se escribe una lista de identificadores que internamente se asocian con las constantes enteras 0, 1, 2, etc.

### Formato

1. `enum`

```
{
 enumerador , enumerador , ...enumerador
};
```
2. `enum nombre`

```
{
 enumerador , enumerador , ...enumerador
};
```

En la declaración del tipo `enum` pueden asociarse a los identificadores valores constantes en vez de la asociación que por defecto se hace (0, 1, 2, etc.). Para ello se utiliza este formato:

```
3. enum nombre
{
 enumerador, = expresión_constante,
 enumerador = expresión-constante,
 ...
 enumerador,= exprsesión-constante,
};
```

## Ejemplo 9.2

### Usos típicos de `enum`

```
enum Interruptor
{
 ENCENDIDO,
 APAGADO
};

enum Boolean
{
 FALSE,
 TRUE
};
```

### Ejemplo

```
enum
{
 ROJO, VERDE, AZUL
};
```

define tres constantes `ROJO`, `VERDE` y `AZUL` de valores iguales a 0, 1 y 2, respectivamente. Los miembros datos de un `enum` se llaman enumeradores y la constante entera por defecto del primer enumerador de la lista de los miembros datos es igual a 0. Obsérvese que, al contrario que `struct` y `union`, los miembros de un tipo `enum` se separan por el operador coma. El ejemplo anterior es equivalente a la definición de las tres constantes, `ROJO`, `VERDE` y `AZUL`, tal como:

```
const int ROJO = 0;
const int VERDE = 1;
const int AZUL = 2;
```

En la siguiente declaración de tipo enumerado se le da un nombre al tipo

```
enum dias-semana
{
 LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
};
```

Una variable de tipo `enum dias-semana` puede tomar los valores especificados en la declaración del tipo. El siguiente bucle está controlado por una variable del tipo enumerado.

```
enum dias-semana dia;

for (dia = LUNES; dia <= DOMINGO; dia++)
{
 ...
}
```

```
 printf("%d ", dia);
}
```

La ejecución del bucle escribiría en pantalla: 0 1 2 3 4 5 6.

A los enumeradores se pueden asignar valores constantes o expresiones constantes durante la declaración:

```
enum Hexaedro
{
 VERTICES = 8,
 LADOS = 12,
 CARAS = 6
}
```

---

## Ejercicio 9.2

El siguiente programa muestra el uso *de* la enumeración *boolean*. El programa lee un texto y cuenta las vocales leídas. La función `vocal()` devuelve `TRUE` si el carácter de entrada es vocal.

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

enum boolean
{
 FALSE, TRUE
};

enum boolean vocal(char c);

void main()
{
 char car;
 int numvocal = 0;

 puts("\nIntroduce un texto. Para terminar: INTRO ");
 while ((car = getchar()) != '\n')
 {
 if (vocal(tolower(car)))
 numvocal++;
 }
 printf("\n Total de vocales leidas: %d\n", numvocal);
}

enum boolean vocal(char c)

switch (c)
{
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u':
 return TRUE;
 default:
 return FALSE;
}
}
```

---

### 9.7.1. sizeof de tipos de datos estructurados

El tamaño en bytes de una estructura, de una unión o de un tipo enumerado se puede determinar con el operador `sizeof`.

El siguiente programa extrae el tamaño de una estructura (`struct`), de una unión (`union`) con miembros dato idénticos, y de un tipo enumerado.

```

/* declara una union */
union tipo-union
{
 char c;
 int i;
 float f;
 double d;
};

/* declara una estructura */
struct tipo-estructura
{
 char c;
 int i;
 float f;
 double d;
};

/* declara un tipo enumerado */
enum monedas
{
 PESETA,
 DURO,
 CINCODUROS,
 CIEN
};
...

printf("\nsizeof (tipo-estructura): %d\n",
 sizeof(struct tipo-estructura));
printf("\nsizeof (tipo-union): %d\n",
 sizeof(union tipo-union));
printf("\nsizeof (monedas): %d\n",
 sizeof(enum monedas));

```

La salida que se genera con estos datos:

```

sizeof(tipo_estructura):15
sizeof(tipo_union): 8
sizeof(monedas): 2

```

### 9.7.2. typedef

Un **typedef** permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o de un tipo ya existente.

---

**Ejemplo**

Uso de `typedef` para declarar un nuevo nombre, `Longitud`, de tipo de dato `double`.

```

...
typedef double Longitud;
...
Longitud Distancia (const struct Pto* p, const struct Pto* p2)
{
 ...
 Longitud longitud = sqrt(r-cua);
 return longitud;
}

```

---

Otros ejemplos:

```

typedef char* String;
typedef const char* string;

```

Puede declararse un tipo estructura o un tipo unión y a continuación asociar el tipo estructura a un nombre con `typedef`.

---

**Ejemplo**

Declaración del *tipo de dato complejo* y asociación a `complex`.

```

struct complejo
{
 float x,y;
};
typedef struct complejo complex;
/* definición de un array de complejos */
complex v[12];

```

**La ventaja de `typedef` es que permite dar nombres de tipos de datos más acordes con lo que representan en una determinada aplicación.**

## 9.8. CAMPOS DE BIT

El lenguaje C permite realizar operaciones con los bits de una palabra. Ya se han estudiado los operadores de manejo de bits: `>>`, `<<`, ... Con los campos de bits, C permite acceder a un número de bits de una palabra entera. Un campo de bits es un conjunto de bits adyacentes dentro de una palabra entera. La sintaxis para declarar campos de bits se basa en la declaración de estructuras. El formato general:

```

struct identificador-campo {
 tipo nombre1: longitud1;
 tipo nombre2: longitud2;
 tipo nombre3: longitud3;

 tipo nombren: longitudn;
};

```

tipo ha de ser entero, `int`; generalmente `unsigned int`  
 longitud es el número de bits consecutivos que se toman

### Ejemplo 9.3

*En este ejemplo se declara un campo de bits para representar en formato comprimido el día, mes año (los dos últimos dígitos) y si el año es bisiesto.*

```
struct fecha {
 unsigned dia: 5;
 unsigned mes: 4;
 unsigned año: 7;
 unsigned bisiesto: 1;
};
```

### Ejemplo 9.4

*El siguiente ejemplo muestra cómo puede utilizarse campos de bits para representar si están o no conectados diversos componentes eléctricos. Cada componente se representa con un flag, con un bit; cuando esté puesto a cero es que no está conectado, cuando esté puesto a uno está conectado.*

```
struct componentes {
 unsigned diodo: 1;
 unsigned resistencia: 1;
 unsigned amperimetro: 1;
 unsigned transistor: 1;
 unsigned condensador: 1;
 unsigned inductancia: 1;
};
```

Los campos individuales se referencian como cualquier otro miembro de una estructura: selector punto (.). Por ejemplo,

```
struct componentes ct;
ct.diodo = 1;
if (ct.amperimetro)
{
}
```

**Al** declarar campos de bits, la suma de los bits declarados puede exceder el tamaño de un entero; en ese caso se emplea la siguiente posición de almacenamiento entero. No está permitido que un campo de bits solape los límites entre dos `int`.

**Al** declarar una estructura puede haber miembros que sean variables y otros campos de bits. La siguiente estructura tiene esta característica:

```
struct reg-estudiante{
 char nombre [33];
 char apell1 [33];
 char apell2 [33];
 unsigned masculino: 1;
 unsigned femenino: 1;
 unsigned curso: 3;
};
```

Los campos de bits se utilizan para rutinas de encriptación de datos y fundamentalmente para ciertos interfaces de dispositivos externos. Presentan ciertas restricciones. Así, no se puede tomar la dirección de una variable campo de bits; no puede haber arrays de campos de bits; no se puede solapar fronteras de `int`. Depende del procesador que los campos de bits se alineen de izquierda a derecha o de derecha a izquierda (conviene hacer una comprobación para cada procesador, utilizando para ello un `union` con variable entera y campos de bits).

### Ejemplo 9.5

Se tiene la función `peticion_acceso()` capaz de direccionar una posición de memoria de 8 bits si recibe como argumento una variable llamada `ochobits`. Con esta variable controla a través de cada bit las peticiones de acceso a cada uno de los ocho periféricos distintos con que trabaja; eventos externos son los que se encargan de cargar la variable `ochobits`.

Se quiere escribir una función que determine cuantos accesos se producen por cada periférico en un bucle de 1000 llamadas a la función `peticion_acceso()`. Se supone que cada llamada sólo activa un periférico.

#### Análisis

El tipo de la variable `ochobits` va a ser una estructura de campos de bits, cada campo con longitud 1; por lo que puede tener dos estados, 0 o 1, para indicar no acceso o sí acceso. Un array de 8 elementos, tantos como periféricos se utiliza para contar los accesos a cada periférico.

```
/* Tipo estructura de campos de bits */

struct perifericos{
 unsigned perf1: 1;
 unsigned perf2: 1;
 unsigned perf3: 1;
 unsigned perf4: 1;
 unsigned perf5: 1;
 unsigned perf6: 1;
 unsigned perf7: 1;
 unsigned perf8: 1;
};
/* Prototipo de función peticion_acceso() */
void peticion-acceso (const struct perifericos ochobits);

/* Función que contabiliza los accesos a cada periférico. */

void accesos_perf(int acceper[])
{
 int i;
 const struct perifericos ochobits;
 const neventos=1000;
 for (i=0; i<8;)
 acceper[i++] = 0;

 /* Bucle principal de 1000 llamadas */
 for (i=0; i<neventos; i++)
 {
```

```

peticion_acceso(ochobits);
if (ochobits.perf1)
 ++acceper[0];
elseif (ocho.bits.perf2);
 ++acceper[1];
elseif (ocho.bits.perf3);
 ++acceper[2];
elseif (ocho.bits.perf4);
 ++acceper[3];
elseif (ocho.bits.perf5);
 ++acceper[4];
elseif (ocho.bits.perf6);
 ++acceper[5];
elseif (ocho.bits.perf7);
 ++acceper[6];
elseif (ocho.bits.perf8);
 ++acceper[7];

```

---

### Ejemplo 9.6

*Haciendo uso de una estructura de campo de bits y de una union, en este ejercicio se escribe un programa para visualizar la decodificación en bits de cualquier carácter leído por teclado.*

#### Análisis

Se declara un tipo estructura campo de bits, con tantos campos como bits tiene un byte, que a su vez es el almacenamiento de un carácter. La decodificación es inmediata declarando una union entre una variable carácter y una variable campo de bits del tipo indicado.

```

#include <stdio.h>
#include <conio.h>

#define mayus (ch) ((ch>='a' && ch<='z') ? (ch+'A'-'a') : ch)

struct byte {
 unsigned int a: 1;
 unsigned int b: 1;
 unsigned int c: 1;
 unsigned int d: 1;
 unsigned int e: 1;
 unsigned int f: 1;
 unsigned int g: 1;
 unsigned int h: 1;
};

union charbits{
 char ch;
 struct byte bits;
}caracter;

void decodifica (struct byte b);

```

```

void main()
{
 puts ('Teclea caracteres. Para salir carácter X');
 do {
 caracter.ch = getche();
 printf(" : ");
 decodifica(caracter.bits);
 }while mayusc(caracter.ch) !='X';
}
void decodifica(struct byte b)

/* Los campos de bits se alinean de derecha a izquierda, por esa razón se
escriben los campos en orden inverso */
printf("%2u%2u%2u%2u%2u%2u%2u%2u \n",
 b.h, b.g, b.f, b.e, b.d, b.c, b.b, b.a);
}

```

## 9.9. RESUMEN

Una estructura permite que los miembros dato de los mismos o diferentes tipos se encapsulen en una única implementación, al contrario que los arrays que son agregados de un tipo de dato simple. Los miembros dato de una estructura son accesibles con el operador punto(.), o con el operador flecha(->).

- Una estructura es un tipo de dato que contiene elementos de tipos diferentes.

```

struct empleado {
 char nombre[30];
 long salario;
 char num_telefono[10];
};

```

- Para crear una variable estructura se escribe

```

struct empleado pepe;

```

- Para determinar el tamaño en bytes de cualquier tipo de dato C utilizar el operador `sizeof`.
- El tipo de dato `union` es similar a `struct` de modo que es una colección de miembros dato de tipos diferentes o similares, pero al contrario que una definición `struct`, que asigna memoria suficiente para contener todos los miembros dato, una `union` puede contener sólo un miembro dato en cualquier momento dado y el tamaño de una unión es, por consiguiente, el tamaño de su miembro mayor.
- Utilizando un tipo `union` se puede hacer que diferentes variables coexistan en el mismo espacio de memoria. La unión permite reducir espacio de memoria en sus programas.
- Un `typedef` no es nuevo tipo de dato sino un **sinónimo de un tipo existente.**

```

typedef struct empleado
 regemplado;

```

## 9.10. EJERCICIOS

- 9.1. Encuentra los errores en la siguiente declaración de estructura y posterior definición de variable.

```
struct hormiga
{
 int patas;
 char especie(41);
 float tiempo;
} hormiga colonia[100];
```

- 9.2. Declara un tipo de datos para representar estaciones del año.
- 9.3. Escribe un función que devuelva la estación año que se ha leído del teclado. La función debe ser del tipo declarado en el Ejercicio 9.2.
- 9.4. Declara un tipo de dato enumerado para representar los meses del año, el mes enero debe estar asociado al dato entero 1, y así sucesivamente los demás meses.
- 9.5. Encuentra los errores del siguiente código

```
#include <stdio.h>
void escribe(struct fecha f)
int main()
{
 struct fecha
 (
 int dia;
 int mes;
 int anyo;
 char mes[];
```

```
 } ff;
 ff = {1,1,2000,"ENERO"};
 escribe(ff);
 return 1;
}
```

9.6. Con ayuda de la declaración anterior, escribe una función que devuelva la estación del año que se ha leído del teclado.

## 9.11. PROBLEMAS

9.1. Escribe un programa para calcular el número de días que hay entre dos fechas; declarar fecha como una estructura.

9.2. Escribe un programa de facturación de clientes. Los clientes tienen un nombre, el número de unidades solicitadas, el precio de cada unidad y el estado en que se encuentra: motoso, atrasado, pagado. El programa debe generar a los diversos clientes.

9.3. Modifique el programa de facturación de clientes de tal modo que se puedan obtener los siguientes listados.

- Clientes en estado moroso.
- Clientes en estado pagado con factura mayor de una determinada cantidad.

9.4. Escribe un programa que permita hacer las operaciones de suma, resta, multiplicación y división de números complejos. El tipo complejo ha de definirse como una estructura,

9.5. Un número racional se caracteriza por el numerador y denominador. Escribe un programa para operar con números racionales. Las operaciones a definir son la suma, resta, multiplicación y división; además de una función para simplificar cada número racional.

9.6. Se quiere informatizar los resultados obtenidos por los equipos de baloncesto y de fútbol de la localidad alcarreña Lupiana. La información de cada equipo:

- Nombre del equipo.
- Número de victorias.
- Número de derrotas.

Para los equipos de baloncesto añadir la información:

- Número de pérdidas de balón.
- Número de rebotes cogidos.
- Nombre del mejor anotador de triples.
- Número de triples del mejor triplista.

Para los equipos de fútbol añadir la información:

- Número de empates.
- Número de goles a favor.
- Número de goles en contra.
- Nombre del goleador del equipo.
- Número de goles del goleador.

Escribir un programa para introducir la información para todos los equipos integrantes en ambas ligas.

9.7. Modificar el programa 9.6 para obtener los siguientes informes o datos.

- Listado de los mejores triplistas de cada equipo.
- Máximo goleador de la liga de fútbol.
- Suponiendo que el partido ganado son tres puntos y el empate 1 punto: equipo ganador de la liga de fútbol.
- Equipo ganador de la liga de baloncesto.

9.8. Un punto en el plano se puede representar mediante una estructura con dos campos. Escribir un programa que realice las siguientes operaciones con puntos en el plano.

- Dados dos puntos calcular la distancia entre ellos.
- Dados dos puntos determinar la ecuación de la recta que pasa por ellos.
- Dados tres puntos, que representan los vértices de un triángulo calcular su área.

# CAPÍTULO 10

## PUNTEROS (APUNTADORES)

---

### CONTENIDO

- 10.1. Direcciones en memoria.
- 10.8. Concepto de puntero (apuntador).
- 10.3.** Punteros `null` y `void`.
- 10.4. Punteros a punteros.
- 10.6. Punteros a arrays.
- 10.6. Arrays de punteros.
- 10.7. Punteros a cadenas.
- 10.8. Aritmética de punteros.
- 10.9. Punteros constantes frente a punteros a constantes.
- 10.10. Punteros como argumento de funciones.
- 10.11.** Punteros a funciones.
- 10.12.** Punteros a estructuras.
- 10.13. *Resumen.*
- 10.14. *Ejercicios.*
- 10.15. *Problemas.*

---

## INTRODUCCIÓN

Los punteros en C tienen la fama, en el mundo de la programación, de dificultad, tanto en el aprendizaje como en su uso. En este capítulo se tratará de mostrar que los punteros no son más difíciles de aprender que cualquier otra herramienta de programación ya examinada o por examinar a lo largo de este libro. El *puntero*, no es más **que** una herramienta muy potente que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los punteros son, sin género de dudas, una de las razones fundamentales para que el lenguaje C sea tan potente y tan utilizado.

Una *variable puntero* (o *puntero*, como **se** llama normalmente) es una variable que contiene direcciones de otras variables. Todas las variables vistas **hasta** este momento contienen valores de datos, por el contrario las variables punteros contienen valores que **son** direcciones de memoria donde se almacenan **datos**. En resumen, un puntero es una variable que contiene una dirección de memoria, y utilizando punteros su programa puede realizar muchas tareas que no sería posible utilizando tipos de **datos** estándar.

En este capítulo se estudiarán los diferentes aspectos de los punteros:

- punteros;
- utilización de punteros;
- asignación dinámica de memoria;
- aritmética de punteros;
- arrays de punteros;
- punteros a punteros, funciones y estructuras.

---

## CONCEPTOS CLAVE

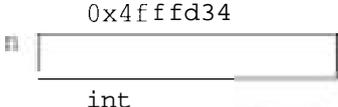
- Puntero (apuntador).
- Direcciones.
- Referencias.
- Palabra reservada **null**.
- Palabra reservada **void**.
- Arrays de punteros.
- Aritmética de punteros.
- Punteros **versus** arrays.
- Tipos de punteros.
- Palabra reservada **const**.

## 10.1. DIRECCIONES EN MEMORIA

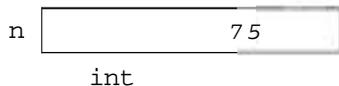
Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su *nombre*, su *tipo* y su *dirección* en memoria.

### Ejemplo

```
int n; /* asocia al nombre n, el tipo int y la dirección
 de alguna posición de memoria donde se almacena
 el valor de n
 */
```



Esta caja representa la posición de almacenamiento en memoria. El nombre de la variable está a la izquierda de la caja, la dirección de variable está encima de la caja y el tipo de variable está debajo en la caja. Si el valor de la variable se conoce, se representa en el interior de la caja



Al valor de una variable se accede por medio de su nombre. Por ejemplo, se puede imprimir el valor de *n* con la sentencia:

```
printf("%d", n);
```

A la dirección de la variable se accede por medio del *operador de dirección* `&`. Por ejemplo, se puede imprimir la dirección de *n* con la sentencia:

```
printf("%p", &n);
```

El operador de dirección `&` «opera» (se aplica) sobre el nombre de la variable para obtener sus direcciones. Tiene precedencia de nivel 15 con el mismo nivel que el operador lógico NOT (`!`) y el operador de preincremento `++`. (Véase Capítulo 4.)

### Ejemplo 10.1

*Obtener el valor y la dirección de una variable.*

```
#include <stdio.h>
void main()
{
 int n = 75;
 printf("n = %d\n", n); /* visualiza el valor de n */
 printf("&n = %p\n", &n); /* visualiza dirección de n */
}
```

## Ejecución

```
n = 45
&n = 0x4ffffd34
```

**Nota:** 0x4ffffd34 es una dirección en código hexadecimal.  
"0x" es el prefijo correspondiente al código hexadecimal.

## 10.2. CONCEPTO DE PUNTERO (APUNTADOR)<sup>1</sup>

Cada vez que se declara una variable C, el compilador establece un área de memoria para almacenar el contenido de la variable. Cuando se declara una variable `int` (entera), por ejemplo, el compilador asigna dos bytes de memoria. El espacio para esa variable se sitúa en una posición específica de la memoria, conocida como *dirección de memoria*. Cuando se referencia (se hace uso) al valor de la variable, el compilador de C accede automáticamente a la dirección de memoria donde se almacena el entero. Se puede ganar en eficacia en el acceso a esta dirección de memoria utilizando un *puntero*.

Cada variable que se declara en C tiene una dirección asociada con ella. Un *puntero* es una dirección de memoria. El concepto de punteros tiene correspondencia en la vida diaria. Cuando se envía una carta por correo, su información se entrega basada en un puntero que es la dirección de esa carta. Cuando se telefonea a una persona, se utiliza un puntero (el número de teléfono que se marca). Así pues, una dirección de correos y un número de teléfono tienen en común que ambos indican dónde encontrar algo. Son punteros a edificios y teléfonos, respectivamente. Un puntero en C también indica dónde encontrar algo, ¿dónde encontrar los datos que están asociados con una variable? *Un puntero C es la dirección de una Variable*. Los punteros se rigen por estas reglas básicas:

- un *puntero* es una *variable* como cualquier otra;
- una variable puntero contiene una *dirección* que apunta a otra posición en memoria;
- en esa posición se almacenan los datos a los que apunta el puntero;
- un puntero apunta a una variable de memoria.

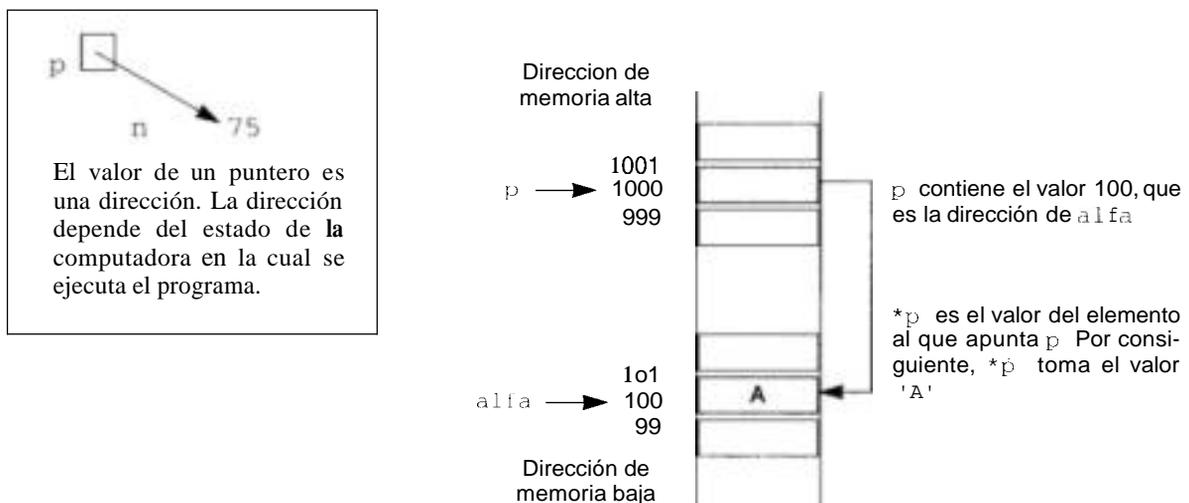


Figura 10.1. Relaciones entre `*p` y el valor de `p` (dirección de alfa).

<sup>1</sup> En Latinoamérica es usual emplear el término *apuntador*.

El tipo de variable que almacena una dirección se denomina *puntero*.

### Ejemplo 10.2

```
#include <stdio.h>
void main()
{
 int n = 75;
 int* p = &n; /* p variable puntero, tiene dirección de n*/
 printf("n = %d, &n = %p, p = %p\n",n,&n,p);
 printf("&p = %p\n",&p);
}
```

### Ejecución

```
n = 75, &n = 0x4fffd34, p = 0x4fffd34
&p = 0x4fffd30
```

|   |           |   |           |
|---|-----------|---|-----------|
|   | 0x4fffd30 |   | 0x4fffd34 |
| p | 0x4fffd34 | n | 75        |
|   | int*      |   | int       |

La variable *p* se denomina «**puntero**» debido a que su valor «*apunta*» a la posición de otro valor. Es un puntero *int* cuando el valor al que apunta es de tipo *int* como en el ejemplo anterior.

#### 10.2.1. Declaración de punteros

Al igual que cualquier variable, las variables punteros han de ser declaradas antes de utilizarlas. La declaración de una variable puntero debe indicar al compilador el tipo de dato al que apunta el puntero; para ello se hace preceder a su nombre con un asterisco (\*), mediante el siguiente formato:

```
<tipo de dato apuntado> *<identificador de puntero>
```

Algunos ejemplos de variables punteros:

```
int* ptr1; /* Puntero a un tipo de dato entero (int)*/
long* ptr2; /* Puntero a un tipo de dato entero largo (long int)*/
char* ptr3; /* Puntero a un tipo de dato char */
float *f; /* Puntero a un tipo de dato float */
```

Un operador \* en una declaración indica que la variable declarada almacenará una dirección de un tipo de dato especificado. La variable *ptr1* almacenará la dirección de un entero, la variable *ptr2* almacenará la dirección de un dato tipo *long*, etc.

**Siempre que aparezca un asterisco (\*) en una definición de una variable, ésta es una variable puntero.**

## 10.2.2. Inicialización<sup>2</sup> (iniciación) de punteros

Al igual que otras variables, C no inicializa los punteros cuando se declaran y es preciso inicializarlos antes de su uso. La inicialización de un puntero proporciona a ese puntero la dirección del dato correspondiente. Después de la inicialización, se puede utilizar el puntero para referenciar los datos direccionados. Para asignar una dirección de memoria a un puntero se utiliza el operador de referencia &. Así, por ejemplo,

```
&valor
```

significa «la dirección de valor». Por consiguiente, el método de inicialización (iniciación), también denominado *estático*, requiere:

- Asignar memoria (estáticamente) definiendo una variable y a continuación hacer que el puntero apunte al valor de la variable.

```
int i; /* define una variable i */
int *p; /* define un puntero a un entero p*/
p = &i; /* asigna la dirección de i a p */
```

- Asignar un valor a la dirección de memoria.

```
*p = 50;
```

Cuando ya se ha definido un puntero, el asterisco delante de la variable puntero indica «*el contenido de*» de la memoria apuntada por el puntero y será del tipo dado.

Este tipo de inicialización es **estática**, ya que la asignación de memoria utilizada para almacenar el valor es fijo y no puede desaparecer. Una vez que la variable se define, el compilador establece suficiente memoria para almacenar un valor del tipo de dato dado. La memoria permanece reservada para esta variable y no se puede utilizar para otra cosa durante la ejecución del programa. En otras palabras, no se puede liberar la memoria reservada para una variable. El puntero a esa variable se puede cambiar, pero permanecerá la cantidad de memoria reservada.

El operador & devuelve la dirección de la variable a la cual se aplica,

Otros ejemplos de inicialización estáticos:

```
1. int edad = 50; /*define una variable edad de valor 50 */
 int *p_edad = &edad; /*define un puntero de enteros inicializándolo
 con la dirección de edad */

2. char *p; /*Figura 10.1 */
 char alfa = 'A';
 p = &alfa;

3. char cd[] = "Compacto";
 char *c;
 .c = cd; /*c tiene la dirección de la cadena cd */
```

Es un error asignar un valor, a un contenido de una variable puntero si previamente no se ha inicializado con la dirección de una variable, o bien se le ha asignado dinámicamente memoria. Por ejemplo:

```
float* px; /* puntero a float */
px = 23.5; / error, px no contiene dirección */
```

<sup>2</sup> El diccionario de la Real Academia de la Lengua Española sólo acepta el término **iniciar** y el término **inicial**. El empleo de *inicializar* sólo se justifica por el extenso uso de dicho término en jerga informática.

Existe un segundo método para inicializar un puntero, es mediante *asignación dinámica de memoria*. Este método utiliza las funciones de asignación de memoria `malloc()`, `calloc()`, `realloc()` y `free()`, y se analizará más adelante en el capítulo siguiente.

### 10.2.3. Indirección de punteros

Después de definir una variable puntero, el siguiente paso es inicializar el puntero y utilizarlo para direccionar algún dato específico en memoria. El uso de un puntero para obtener el valor al que apunta, es decir, su dato apuntado se denomina *indireccionar el puntero* («desreferenciar el puntero»); para ello, se utiliza el operador de indirección `*`.

```
int edad;
int* p_edad;
p_edad= &edad;
*p_edad= 50;
```

Las dos sentencias anteriores se describen en la Figura 10.2. Si se desea imprimir el valor de edad, se puede utilizar la siguiente sentencia:

```
printf("%d", edad); /* imprime el valor de edad */
```

También se puede imprimir el valor de edad dereferenciando el puntero a edad:

```
printf("%d", *p_edad); /*indirecciona p_edad */
```

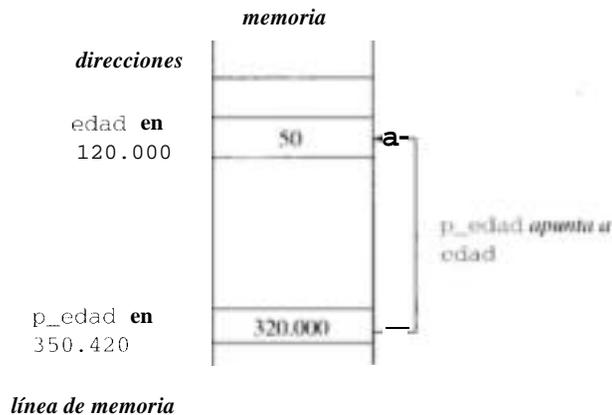


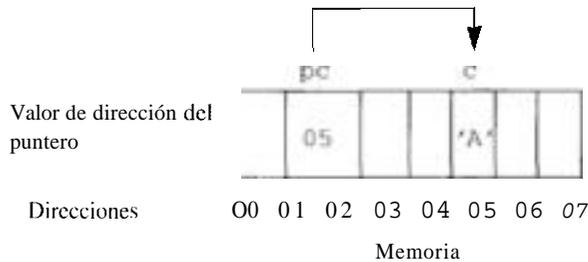
Figura 10.2. `p_edad` contiene la dirección de `edad`, `p_edad` apunta a la variable `edad`

El listado del siguiente programa muestra el concepto de creación, inicialización e indirección de una variable puntero.

```
#include <stdio.h>
char c; /* variable global de tipo carácter*/
int main0

char *pc; /* un puntero a una variable carácter*/
pc = &c;
for (c = 'A'; c <= 'Z'; c++);
printf("%c", *pc);
return 0;
}
```

La ejecución de este programa visualiza el alfabeto. La variable puntero `pc` es un puntero a una variable carácter. La línea `pc = &c` asigna a `pc` la dirección de la variable `c` (`&c`). El bucle `for` almacena en `c` las letras del alfabeto y la sentencia `printf("%c", *pc)`; visualiza el contenido de la variable apuntada por `pc`; `c` y `pc` se refieren a la *misma* posición en memoria. Si la variable `c`, que se almacena en cualquier parte de la memoria, y `pc`, que apunta a esa misma posición, se refiere a los mismos datos, de modo que el cambio de una variable debe afectar a la otra; `pc` y `c` se dice que son *alias*, debido a que `pc` actúa como otro nombre de `c`.



**Figura 10.3.** `pc` y `c` direccionan la misma posición de memoria.

La Tabla 10.1 resume los operadores de punteros.

**Tabla 10.1.** Operadores de punteros.

| Operador           | Propósito                                     |
|--------------------|-----------------------------------------------|
| <code>&amp;</code> | Obtiene la dirección de una variable.         |
| <code>*</code>     | Define una variable como puntero.             |
| <code>*</code>     | Obtiene el contenido de una variable puntero. |

### Nota

Son variables punteros aquellas que apuntan a la posición en donde otra/s variable/s de programa se almacenan.

#### 10.2.4. Punteros y verificación de tipos

Los punteros se enlazan a tipos de datos específicos, de modo que C verificará si se asigna la dirección de un tipo de dato al tipo correcto de puntero. Así, por ejemplo, si se define un puntero a `float`, no se le puede asignar la dirección de un carácter o un entero. Por ejemplo, este segmento de código no funcionará:

```
float *fp;
char c;
fp = &c; /* no es válido */
```

C no permite la asignación de la dirección de `c` a `fp`, ya que `fp` es una variable puntero que apunta a datos de tipo real, `float`.

C requiere que las variables puntero **direccionen** realmente variables del mismo **tipo** de dato que está ligado a los punteros en sus declaraciones.

### 10.3. PUNTEROS `NULL` Y `void`

Normalmente un puntero inicializado adecuadamente apunta a alguna posición específica de la memoria. Sin embargo, un puntero no inicializado, como cualquier variable, tiene un valor aleatorio hasta que se inicializa el puntero. En consecuencia, será preciso asegurarse que las variables puntero utilicen direcciones de memoria válida.

Existen dos tipos de punteros especiales muy utilizados en el tratamiento de **sus** programas: los punteros `void` y `NULL` (nulo).

Un *puntero nulo* no apunta a ninguna parte —dato válido— en particular, es decir, «un puntero nulo no direcciona ningún dato válido en memoria». Un puntero nulo se utiliza para proporcionar a un programa un medio de conocer cuando una variable puntero no direcciona a un dato válido. Para declarar un puntero nulo se utiliza la macro `NULL`, definida en los archivos de cabecera `STDEF.H`, `STDIO.H`, `STDLIB.H` y `STRING.H`. Se debe incluir uno o más de estos archivos de cabecera antes de que se pueda utilizar la macro `NULL`. Ahora bien, se puede definir `NULL` en la parte superior de su programa (o en un archivo de cabecera personal) con la línea:

```
#define NULL 0
```

Un sistema de inicializar una variable puntero a nulo es:

```
char *p = NULL;
```

Algunas funciones C también devuelven el valor `NULL` si se encuentra un error. Se pueden añadir test para el valor `NULL` comparando el puntero con `NULL`:

```
char *p;
p = malloc(121*sizeof(char));
if (p == NULL)

 puts("Error de asignación de memoria");
}
```

o bien

```
if (p != NULL) ...
/* este if es equivalente a : */
if (p) ...
```

Otra forma de declarar un puntero nulo es asignar un valor de 0. Por ejemplo,

```
int *ptr = (int *) 0; /* ptr es un puntero nulo */
```

El modelo (*casting*) anterior (`int *`), no es necesario, hay una conversión estándar de 0 a una variable puntero.

```
int *ptr = 0;
```

Nunca se utiliza un puntero nulo para referenciar un valor. Como antes se ha comentado, los punteros nulos se utilizan en un test condicional para determinar si un puntero se ha inicializado. En el ejemplo

```
if (ptr)
 printf("Valor de la variable apuntada por ptr es: %d\n", *ptr);
```

se imprime un valor si el puntero es válido y no es un puntero nulo.

Los punteros nulos se utilizan con frecuencia en programas con arrays de punteros. Cada posición del array se inicializa a `NULL`; después se reserva memoria dinámicamente y se asigna a la posición correspondiente del array, la dirección de la memoria.

En C se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, no se asigna a un tipo de dato específico. El método es declarar el puntero como un puntero `void *`, denominado puntero genérico.

```
void *ptr; /* declara un puntero void, puntero genérico */
```

El puntero `ptr` puede direccionar cualquier posición en memoria, pero el puntero no está unido a un tipo de dato específico. De modo similar, los punteros `void` pueden direccionar una variable `float`, una `char`, o una posición arbitraria o una cadena.

### Nota

No confundir punteros `void` y `NULL`. Un puntero nulo no direcciona **ningún** dato válido. Un puntero `void` direcciona datos de un tipo no especificado. Un puntero `void` se puede igualar a nulo si no se direcciona ningún dato válido. Nulo es un valor; `void` es un tipo de **dato**.

## 10.4. PUNTEROS A PUNTEROS

Un puntero puede apuntar a otra variable puntero. Este concepto se utiliza con mucha frecuencia en programas complejos de C. Para declarar un puntero a un puntero se hace preceder a la variable con dos asteriscos (`**`).

En el ejemplo siguiente `ptr5` es un puntero a un puntero.

```
int valor-e = 100;
int *ptr1 = &valor-e;
int **ptr5 = &ptr1;
```

`ptr1` y `ptr5` son punteros. `ptr1` apunta a la variable `valor-e` de tipo `int`. `ptr5` contiene la dirección de `ptr1`. En la Figura 10.4 se muestran las declaraciones anteriores.

Se puede asignar valores a `valor-e` con cualquiera de las sentencias siguientes:

```
valor-e = 95;
ptr1 = 105; / Asigna 105 a valor-e */
**ptr5 = 99; /* Asigna 99 a valor-e */
```

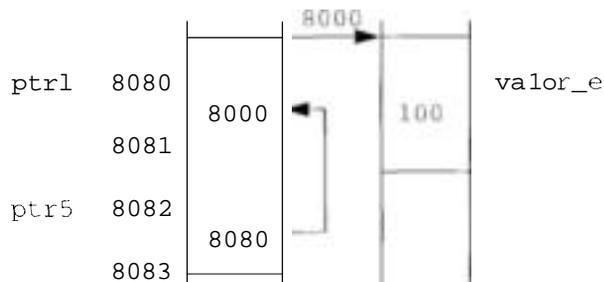
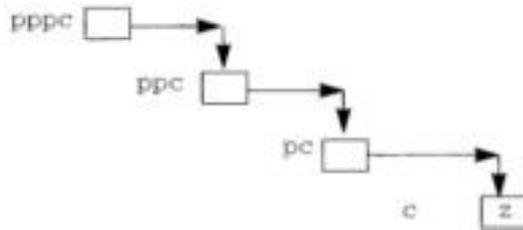


Figura 10.4. Un puntero a un puntero.

### Ejemplo

```
char c = 'z';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'm'; /* cambia el valor de c a 'm' */
```



## 10.5. PUNTEROS Y ARRAYS

Los arrays y punteros están fuertemente relacionados en el lenguaje C. Se pueden direccionar arrays como si fueran punteros y punteros como si fueran arrays. La posibilidad de almacenar y acceder a punteros y arrays, implica que se pueden almacenar cadenas de datos en elementos de arrays. Sin punteros eso no es posible, ya que no existe el tipo de dato cadena (*string* en C. No existen variables de cadena. Únicamente constantes de cadena.

### 10.5.1. Nombres de arrays como punteros

Un nombre de un array es simplemente un puntero. Supongamos que se tiene la siguiente declaración de un array:

```
int lista[5] = {10, 20, 30, 40, 50};
```

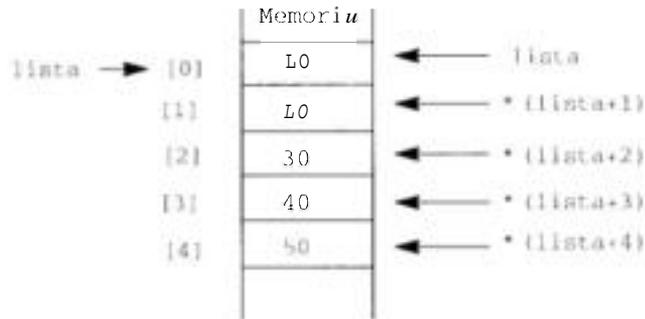


Figura 10.5. Un array almacenado en memoria.

Si se manda visualizar `lista[0]` se verá 10 . Pero, ¿qué sucederá si se manda visualizar `*lista`? Como un nombre de un array es un puntero, también se verá 10 . Esto significa que

|                        |                 |                       |
|------------------------|-----------------|-----------------------|
| <code>lista + 0</code> | <i>apunta a</i> | <code>lista[0]</code> |
| <code>lista + 1</code> | <i>apunta a</i> | <code>lista[1]</code> |
| <code>lista + 2</code> | <i>apunta a</i> | <code>lista[2]</code> |
| <code>lista + 3</code> | <i>apunta a</i> | <code>lista[3]</code> |
| <code>lista + 4</code> | <i>apunta a</i> | <code>lista[4]</code> |

Por consiguiente, para imprimir (visualizar), almacenar o calcular un elemento de un array, se puede utilizar notación de subíndices o notación de punteros. Dado que un nombre de un array contiene la dirección del primer elemento del array, se debe indireccionar el puntero para obtener el valor del elemento.

El nombre de un array es un puntero, contiene la dirección en memoria de comienzo de la secuencia de elementos que forma el array. Es un puntero constante ya que no se puede modificar, sólo se puede acceder para indexar a los elementos del array. En el ejemplo se pone de manifiesto operaciones correctas y erróneas con nombres de array.

```
float v[10];
float *p;
float x = 100.5;
int j;

 /* se indexa a partir de v */
for (j= 0; j<10; j++)
 *(v+j) = j*10.0;

p = v+4; /* se asigna la dirección del quinto elemento */
v = &x; /* error: intento de modificar un puntero constante */
```

## 10.5.2. Ventajas de los punteros

Un nombre de un array es una *constante puntero*, no una variable puntero. No se puede cambiar el valor de un nombre de array, como no se pueden cambiar constantes. Esto explica por qué no se pueden asignar valores nuevos a un array durante una ejecución de un programa. Por ejemplo, si *cnombre* es un array de caracteres, la siguiente sentencia no es válida en C:

```
cnombre = "Hermanos Daltón";
```

Se pueden asignar valores al nombre de un array sólo en el momento de la declaración, o bien utilizando funciones, tales como (ya se ha hecho anteriormente) `strcpy()`.

Se pueden cambiar punteros para hacerlos apuntar a valores diferentes en memoria. El siguiente programa muestra como cambiar punteros. El programa define dos valores de coma flotante. Un puntero de coma flotante apunta a la primera variable *v1* y se utiliza en `printf()`. El puntero se cambia entonces, de modo que apunta a la segunda variable de coma flotante *v2*.

```
#include <stdio.h>

int main()
{
 float v1 = 756.423;
 float v2 = 900.545;
 float *p_v;

 p_v = &v1;
 printf("El primer valor es %f \n", *p_v); /*se imprime 756.423 */

 p_v = &v2;
 printf("El segundo valor es %f \n", *p_v); /*se imprime 900.545 */
 return 0;
}
```

Por esta facilidad para cambiar punteros, la mayoría de los programadores de C utilizan punteros en lugar de arrays. Como los arrays son fáciles de declarar, los programadores declaran arrays y a continuación utilizan punteros para referencia a los elementos de dichos arrays.

## 10.6. ARRAYS DE PUNTEROS

Si se necesita reservar muchos punteros a muchos valores diferentes, se puede declarar un *array de punteros*. Un array de punteros es un array que contiene como elementos punteros, cada uno de los cuales apunta a un tipo de dato específico. La línea siguiente reserva un array de diez variables puntero a enteros:

```
int *ptr[10]; /* reserva un array de 10 punteros a enteros */
```

La Figura 10.6 muestra cómo C organiza este array. Cada elemento contiene una dirección que *apunta* a otros valores de la memoria. Cada valor apuntado debe ser un entero. Se puede asignar a un elemento de `ptr` una dirección, tal como para variables puntero no arrays. Así, por ejemplo,

```
ptr[5] = &edad; /* ptr[5] apunta a la dirección de edad */
ptr[4] = NULL; /* ptr[4] no contiene dirección alguna */
```

Otro ejemplo de arrays de punteros, en este caso de caracteres es:

```
char *puntos[25]; /* array de 25 punteros a carácter */
```

De igual forma, se podría declarar un puntero a un array de punteros a enteros.

```
int *(*ptr10)[];
```

y las operaciones paso a paso son:

```
(*ptr10) es un puntero, ptr10 es un nombre de variable.
(*ptr10)[1] es un puntero a un array
*(*ptr10)[] es un puntero a un array de punteros
int *(*ptr10)[] es un puntero a un array de punteros de variables int
```

Una matriz de número enteros, o reales, puede verse como un array de punteros; de tantos elementos como filas tenga la matriz, apuntando cada elemento del array a un array de enteros o reales, de tantos elementos como columnas.

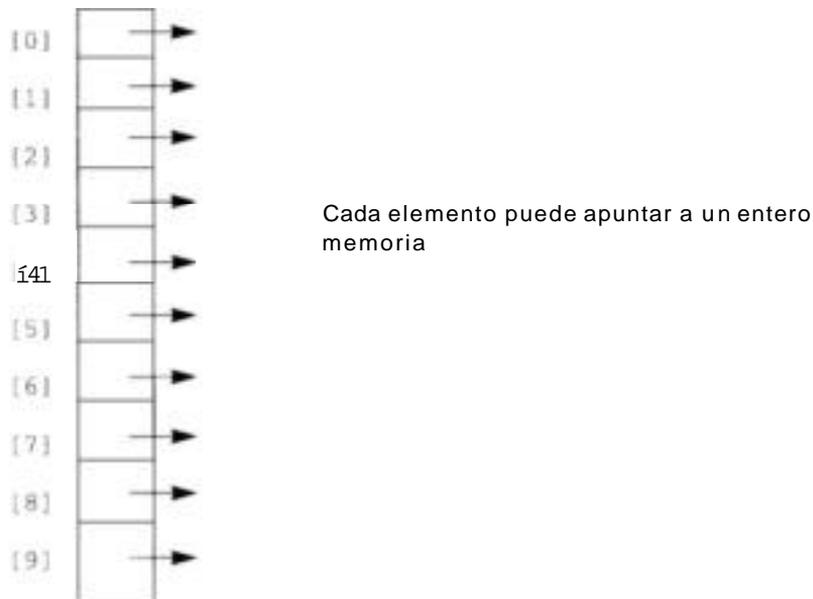


Figura 10.6. Un array de 10 punteros a enteros.

### 10.6.1. Inicialización de un array de punteros a cadenas

La inicialización de un array de punteros a cadenas se puede realizar con una declaración similar a ésta:

```
char *nombres_meses [12] = { "Enero", "Febrero", "Marzo",
 "Abril", "Mayo", "Junio",
 "Julio", "Agosto", "Septiembre",
 "Octubre", "Noviembre",
 "Diciembre" };
```

## 10.7. PUNTEROS DE CADENAS

Los punteros se pueden utilizar en lugar de índices de arrays. Considérese la siguiente declaración de un array de caracteres que contiene las veintiséis letras del alfabeto internacional (no se considera la ñ).

```
char alfabeto[27] = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

Declaremos ahora `p` un puntero a `char`

```
char *p;
```

Se establece que `p` apunta al primer carácter de `alfabeto` escribiendo

```
p = &alfabeto[0]; /* o también p = alfabeto */
```

de modo que si escribe la sentencia

```
printf("%c \n", *p);
```

se visualiza la letra A, ya que `p` apunta al primer elemento de la cadena. Se puede hacer también

```
p = &alfabeto[15];
```

de modo que `p` apuntará al carácter 16º (la letra Q). Sin embargo, no se puede hacer

```
p = &alfabeto;
```

ya que `alfabeto` es un array cuyos elementos son de tipo `char`, y se produciría un error al compilar (tipo de asignación es incompatible).

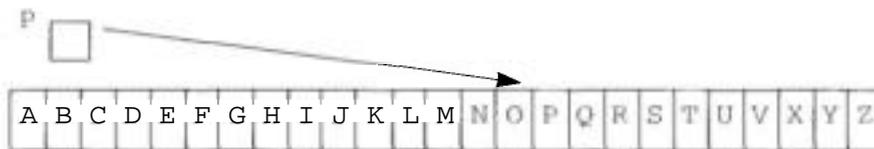


Figura 10.7. Un puntero a `alfabeto[15]`.

Es posible, entonces, considerar dos tipos de definiciones de cadena:

```
char cadenal[]="Hola viejo mundo"; /* array contiene una cadena */
char *cptr = "C a su alcance"; /* puntero a cadena, el sistema
 reserva memoria para la cadena*/
```

### 10.7.1. Punteros *versus* arrays

El siguiente programa implementa una función para contar el número de caracteres de una cadena. En el primer programa, la cadena se describe utilizando un array, y en el segundo, se describe utilizando un puntero.

```

/* Implementación con un array */
#include <stdio.h>
int longitud(const char cad[]);
void main()
{
 static char cad[] = "Universidad Pontificia";
 printf("La longitud de %s es %d caracteres\n",
 cad, longitud(cad));
}
int longitud(const char cad[])
{
 int posicion = 0;
 while (cad[posicion] != '\0')
 {
 posicion++;
 }
 return posicion;
}

```

El segundo programa utiliza un puntero para la función que cuenta los caracteres de la cadena. Además, utiliza la aritmética de punteros para indexar los caracteres. El bucle termina cuando llega al último carácter, que es el delimitador de una cadena: \0.

```

/* Implementación con un puntero */

#include <stdio.h>
int longitud(const char*);
void main()
{
 static char cad[] = "Universidad Pontificia";
 printf("La longitud de %s es %d caracteres\n",
 cad, longitud(cad));
}
int longitud(const char* cad)
{
 int cuenta = 0;
 while (*cad++) cuenta++;
 return cuenta;
}

```

En ambos casos se imprimirá:

La longitud de Universidad Pontificia es 22 caracteres

### Comparaciones entre punteros y arrays de punteros

```

int *ptr1[1]; /* Array de punteros a int */
int (*ptr2)[1]; /* Puntero a un array de elementos int */
int *(*ptr3)[]; /* Puntero a un array de punteros a int */

```

## 10.8. ARITMÉTICA DE PUNTEROS

Al contrario que un nombre de array, que es un puntero constante y no se puede modificar, un puntero es una variable que se puede modificar. Como consecuencia, se pueden realizar ciertas operaciones aritméticas sobre punteros.

A un puntero se le puede sumar o restar un entero  $n$ ; esto hace que apunte  $n$  posiciones adelante, o atrás de la actual. Una variable puntero puede modificarse para que contenga una dirección de memoria  $n$  posiciones adelante o atrás. Observe el siguiente fragmento:

```
int v[10];
int *p;
p = v;
(v+4); /* apunta al 5º elemento */
p = p+6; /* contiene la dirección del 7º elemento */
```

A una variable puntero se le puede aplicar el operador  $++$ , o el operador  $--$ . Esto hace que contenga la dirección del siguiente, o anterior elemento. Por ejemplo:

```
float m[20];
float *r;
r = m;
r++; /* contiene la dirección del elemento siguiente */
```

Recuérdese que un puntero es una dirección, por consiguiente, sólo aquellas operaciones de «sentido común» son legales. No tiene sentido, por ejemplo, sumar o restar una constante de coma flotante.

### Operaciones no válidas con punteros

- No se pueden sumar dos punteros.
- No se pueden multiplicar dos punteros.
- No se pueden dividir dos punteros.

---

### Ejemplo 10.3

Si  $p$  apunta a la letra A en `alfabeto`, si se escribe

```
p = p+1;
```

entonces  $p$  apunta a la letra B.

Se puede utilizar esta técnica para explorar cada elemento de `alfabeto` sin utilizar una variable de índice. Un ejemplo puede ser

```
p = &alfabeto[0];
for (i = 0; i < strlen(alfabeto); i++)
{
 printf("%c", *p);
 p = p+1;
};
```

Las sentencias del interior del bucle se pueden sustituir por

```
printf("%c", *p++);
```

---

El ejemplo anterior con el bucle `for` puede ser abreviado, haciendo uso de la característica de *terminador* nulo al final de la cadena. Utilizando la sentencia `while` para realizar el bucle y poniendo la condición de terminación de nulo o byte cero al final de la cadena. Esto elimina la necesidad del bucle `for` y su variable de control. El bucle `for` se puede sustituir por

```
while (*p) printf("%c", *p++);
```

mientras que  $*p$  toma un valor de carácter distinto de cero, el bucle `while` se ejecuta, el carácter se imprime y  $p$  se incrementa para apuntar al siguiente carácter. Al alcanzar el byte cero al final de la cadena,  $*p$  toma el valor de `'\0'` o cero. El valor cero hace que el bucle termine.

### 10.8.1. Una aplicación de punteros: conversión de caracteres

El siguiente programa muestra un puntero que recorre una cadena de caracteres y convierte cualquier carácter en minúsculas a caracteres mayúsculas.

```

/* Utiliza un puntero como índice de un array de caracteres
 y convierte caracteres minúsculas a mayúsculas
*/
#include <stdio.h>
#include <conio.h>
void main()
{
 char *p;
 char CadenaTexto[81];

 puts("Introduzca cadena a convertir:");
 gets(CadenaTexto);

 /* p apunta al primer carácter de la cadena */
 p = &CadenaTexto[0]; /* equivale a p = CadenaTexto */

 /* Repetir mientras *p no sea cero */
 while (*p)
 {
 /* restar 32, constante de código ASCII */
 if ((*p >= 'a') && (*p <= 'z'))
 *p++ = *p-32;
 else
 p++;
 }
 puts("La cadena convertida es:");
 puts(CadenaTexto);

 puts("\nPulse Intro (Enter) para continuar");
 getch();
}

```

Obsérvese que si el carácter leído está en el rango entre 'a' y 'z'; es decir, es una letra minúscula, la asignación

```
*p++ = *p-32;
```

se ejecutará, y restar 32 del código ASCII de una letra minúscula convierte a esta letra en mayúscula).

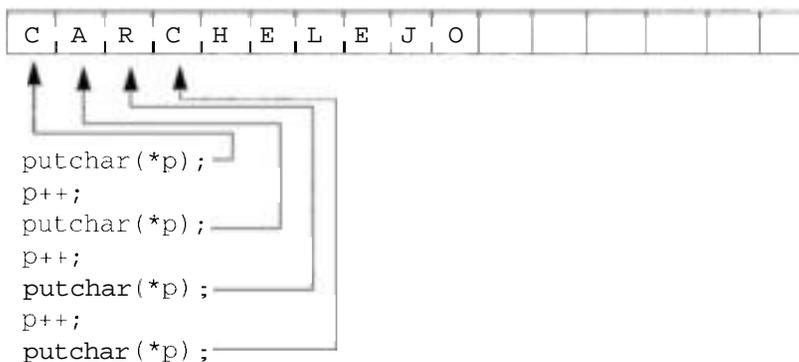


Figura 10.8. \*p++ se utiliza para acceder de modo incremental en la cadena

## 10.9. PUNTEROS CONSTANTES FRENTE A PUNTEROS A CONSTANTES

Ya está familiarizado con punteros constantes, como es el caso de un nombre de un array. Un puntero constante es un puntero que no se puede cambiar, pero que los datos apuntados por el puntero pueden ser cambiados. Por otra parte, un puntero a una constante se puede modificar para apuntar a una constante diferente, pero los datos apuntados por el puntero no se pueden cambiar.

### 10.9.1. Punteros constantes

Para crear un puntero constante diferente de un nombre de un array, se debe utilizar el siguiente formato:

```
<tipo de dato > *const <nombre puntero> = <dirección de variable >;
```

Como ejemplo de una definición de punteros de constantes, considérense las siguientes sentencias:

```
int x;
int y;
int *const p1 = &x;
```

`p1` es un puntero de constantes que apunta a `x`, por lo que `p1` es una constante, pero `*p1` es una variable. Por consiguiente, se puede cambiar el valor de `*p1`, pero no `p1`.

Por ejemplo, la siguiente asignación es legal, dado que se cambia el contenido de memoria a donde apunta `p1`, pero no el puntero en sí.

```
*p1 = y;
```

Por otra parte, la siguiente asignación no es legal, ya que se intenta cambiar el valor del puntero

```
p1 = &y;
```

El sistema para crear un puntero de constante a una cadena es:

```
char *const nombre = "Luis";
```

`nombre` no se puede modificar para apuntar a una cadena diferente en memoria. Por consiguiente,

```
*nombre = 'C';
```

es legal, ya que se modifica el dato apuntado por `nombre` (cambia el primer carácter). Sin embargo, *no es legal*:

```
nombre = &Otra-Cadena;
```

dado que se intenta modificar el propio puntero

### 10.9.2. Punteros a constantes

El formato para definir un puntero a una constante es:

```
const <tipo de dato elemento> *<nombre puntero> =
 <dirección de constante >;
```

Algunos ejemplos:

```
const int x = 25;
const int y = 50;
const int *p1 = &x;
```

en los que `p1` se define como un puntero a la constante `x`. Los datos son constantes y no el puntero; en consecuencia, se puede hacer que `p1` apunte a otra constante.

```
p1 = &y;
```

Sin embargo, cualquier intento de cambiar el contenido almacenado en la posición de memoria a donde apunta `p1` creará un error de compilación. Así, la siguiente sentencia no se compilará correctamente:

```
*p1 = 15;
```

## Nota

Una definición de un puntero constante tiene la palabra reservada **const** delante del nombre del puntero, mientras que el puntero a una definición constante requiere que la palabra reservada **const** se sitúe antes del tipo de dato. Así, la definición en el primer caso se puede leer como «puntero constante o de constante», mientras que en el segundo caso la definición se lee «puntero a tipo constante de dato».

La creación de un puntero a una constante cadena se puede hacer del modo siguiente:

```
const char *apellido = "Remirez";
```

En el prototipo de la siguiente función se declara el argumento como puntero a una constante:

```
float cargo(const float *v);
```

### 10.9.3. Punteros constantes a constantes

El último caso a considerar es crear punteros constantes a constantes utilizando el formato siguiente:

```
const <tipo de dato elemento> *const <nombre puntero> =
 <dirección de constante >;
```

Esta definición se puede leer como «un tipo constante de dato y un puntero constante». Un ejemplo puede ser:

```
const int x = 25;
const int *const p1 = &x;
```

que indica: «`p1` es un puntero constante que apunta a la constante entera `x`». Cualquier intento de modificar `p1` o bien `*p1` producirá un error de compilación.

## Regla

- Si sabe que un puntero siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defínalo como un puntero constante.
- Si sabe que el dato apuntado por el puntero nunca necesitará cambiar, defina el puntero como un puntero a una constante.

**Ejemplo 10.4**

Un puntero a una constante es diferente de un puntero constante. El siguiente ejemplo muestra las diferencias.

```

/*
 Este trozo de código define cuatro variables:
 un puntero p; un puntero constante cp; un puntero pc a una
 constante y un puntero constdnte cpc a una constante
*/
int *p; /* puntero a un int */
++(*p); /* incremento del entero *p */
++p; /* incrementa un puntero p */
int *const cp; /* puntero constante a un int */
++(*cp); /* incrementa el entero *cp */
++cp; /* no válido: puntero cp es constante */
const int * pc; /* puntero a una constante int */
++(*pc); /* no válido: int * pc es constante */
++pc; /* incrementa puntero pc */
const int * const cpc; /* puntero constante a constante int */
++(*cpc); /* no válido: int *cpc es constante */
++cpc; /* no válido: puntero cpc es constante */

```

**Regla**

El espacio en blanco no es significativo en la declaración de punteros. Las declaraciones siguientes son equivalentes:

```

int* p;
int * p;
int *p;

```

**10.10. PUNTEROS COMO ARGUMENTOS DE FUNCIONES**

Con frecuencia se desea que una función calcule y devuelva más de un valor, o bien se desea que una función modifique las variables que se pasan como argumentos. Cuando se pasa una variable a una función (*paso por valor*) no se puede cambiar el valor de esa variable. Sin embargo, si se pasa un puntero a una variable a una función (*paso por dirección*) se puede cambiar el valor de la variable.

Cuando una variable es local a una función, se puede hacer la variable visible a otra función pasándola como argumento. Se puede pasar un puntero a una variable local como argumento y cambiar la variable en la otra función.

Considere la siguiente definición de la función `Incrementar5()`, que incrementa un entero en 5:

```

void Incrementar5(int *i)
{
 *i += 5;
}

```

La llamada a esta función se realiza pasando una dirección que utilice esa función. Por ejemplo, para llamar a la función `Incrementar5()` utilice:

```

int i;
i = 10;
Incrementar5(&i);

```

Es posible mezclar paso por referencia y por valor. Por ejemplo, la función `func1` definida como

```
void func1(int *s, int t)
{
 *s = 6;
 t = 25;
}
```

y la invocación a la función podría ser:

```
int i, j;
i = 5;
j = 7;
func1(&i, j); /*llamada a func1 */
```

Cuando se retorna de la función `func1` tras su ejecución, `i` será igual a 6 y `j` seguirá siendo 7, ya que se pasó por valor. El paso de un nombre de array a una función es lo mismo que pasar un puntero al array. Se pueden cambiar cualquiera de los elementos del array. Cuando se pasa un elemento a una función, sin embargo, el elemento se pasa por valor. En el ejemplo

```
int lista[] = {1, 2, 3};
func(lista[1], lista[2]);
```

ambos elementos se pasan por valor.

**En C, por defecto, el paso de parámetros se hace por valor. C no tiene parámetros por referencia, hay que emularlo mediante el paso de la dirección de una variable, utilizando punteros en los argumentos de la función.**

En el siguiente ejemplo, se crea una estructura para apuntar las temperaturas más alta y más baja de un día determinado.

```
struct temperatura {
 float alta;
 float baja;
};
```

Un caso típico podría ser almacenar las lecturas de un termómetro conectado de algún modo posible a una computadora. Una función clave del programa lee la temperatura actual y modifica el miembro adecuado, `alta` o `baja`, en una estructura `temperatura` de la que se pasa la dirección del argumento a un parámetro puntero.

```
void registrottemp(struct temperatura *t)
{
 float actual;

 leertempactual(actual);
 if (actual > t -> alta)
 t -> alta = actual;
 else if (actual < t -> baja)
 t -> baja = actual;
}
```

La llamada a la función se puede hacer con estas sentencias:

```
struct temperatura tmp;
registrottemp(&tmp);
```

## 10.11. PUNTEROS A FUNCIONES

Hasta este momento se han analizado punteros a datos. Es posible declarar punteros a cualquier tipo de variables, estructura o array. De igual modo, las funciones pueden declarar parámetros punteros para permitir que sentencias pasen las direcciones de los argumentos a esas funciones.

Es posible también crear punteros que apunten a funciones. En lugar de direccionar datos, los punteros de funciones apuntan a código ejecutable. Al igual que los datos, las funciones se almacenan en memoria y tienen direcciones iniciales. En *C* se pueden asignar las direcciones iniciales de funciones a punteros. Tales funciones se pueden llamar en un modo indirecto, es decir, mediante un puntero cuyo valor es igual a la dirección inicial de la función en cuestión.

La sintaxis general para la declaración de un puntero a una función es:

```
Tipo-de-retorno (*PunteroFuncion) (<lista de parámetros>);
```

Este formato indica al compilador que *PunteroFuncion* es un puntero a una función que devuelve el tipo *Tipo-de-retorno* y tiene una lista de parámetros.

Un puntero a una función es simplemente un puntero cuyo valor es la dirección del nombre de la función. Dado que el nombre es, en sí mismo, un puntero; un puntero a una función es un puntero a un puntero constante.

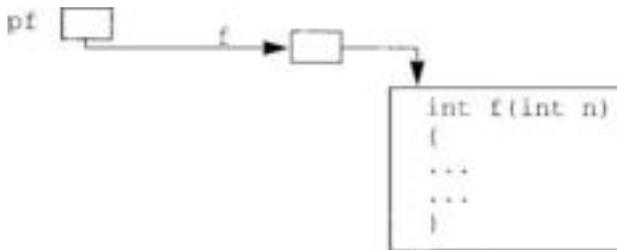


Figura 10.9. Puntero a función.

Por ejemplo:

```
int f(int); /* declara la función f */
int (*pf)(int); /* define puntero pf a función int con argumento
 int */
pf = f; /* asigno la dirección de f a pf */
```

### Ejemplo 10.5

```
double (*fp) (int n);
float (*p) (int i, int j);
void (*sort)(int* ArrayEnt, unsigned n);
unsigned (*search)(int BuscarClave, int* ArrayEnt, unsigned n);
```

El primer identificador, *fp*, apunta a una función que devuelve un tipo *double* y tiene un único parámetro de tipo *int*. El segundo puntero, *p*, apunta a una función que devuelve un tipo *float* y acepta dos parámetros de tipo *int*. El tercer puntero, *sort*, es un puntero a una función que devuelve un tipo *void* y toma dos parámetros: un puntero a *int* y un tipo *unsigned*. Por último, *search* es un puntero a una función que devuelve un tipo *unsigned* y tiene tres parámetros: un *int*, un puntero a *int* y un *unsigned*.

#### 10.11.1. Inicialización de un puntero a una función

La sintaxis general para inicializar un puntero a una función es:

```
PunteroFuncion = unaFuncion
```

La función asignada debe tener el mismo tipo de retorno y lista de parámetros que el puntero a función; en caso contrario, se producirá un error de compilación. Así, por ejemplo, un puntero `qf` a una función `double`:

```
double calculo (int* v; unsigned n); /* prototipo de función */

double (*qf) (int*, unsigned); /* puntero a función */
int r[11] = {3,5,6,7,1,7,3,34,5,11,44};
double x;
qf = calculo; /* asigna dirección de la función */
x = qf(r,11); /* llamada a la función con el puntero a función */
```

Algunas de las funciones de la biblioteca, tales como `qsort()`, requiere pasar un argumento que consta de un puntero a una función. Se debe pasar a `qsort` un puntero de función que apunta a una función.

### Ejemplo 10.6

*Se desea ordenar un array de números reales, la ordenación se va a realizar con la función `qsort()`.*

Esta función tiene un parámetro que es un puntero a función del tipo `int (*) (const void*, const void*)`. Se necesita un función de comparación, que devuelva negativo si primer argumento es menor que el segundo, 0 si son iguales y positivo si es mayor. A continuación se escribe el programa:

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

int compara_float(const void* a, const void* b); /* prototipo de función
de comparación */

float v[] = {34.5, -12.3, 4.5, 9.1, -2.5, 18.0, 10., 5.5};

int main()
{
 int j, n;
 int (*pf)(const void*, const void*); /* puntero a función */

 n = sizeof(v)/sizeof(v[0]); /* numero de elementos */
 printf ("\n Numero de elementos: %d\n", n);

 pf = compara_float;
 qsort((void*)v, n, sizeof(v[0]), pf); /* Llamada a función de
biblioteca. */

 for (j = 0; j < n; j++)
 printf("%.2f ", v[j]);
 puts ("\n Pulsa cualquier tecla para continuar. ...");
 j = getchar();
 return 0;
}

int compara_float(const void *a, const void *b)
{
 float *x, *y;
 x = (float*)a; y = (float*)b;
 return(*x - *y);
}
```

**Ejemplo 10.7**

Supongamos un puntero *p* a una función tal como

```
float (*p) (int i, int j);
```

a continuación se puede asignar la dirección de la función ejemplo:

```
float ejemplo(int i, int j)
{
 return 3.14159 * i * i + j;
}
```

al puntero *p* escribiendo

```
p = ejemplo;
```

Después de esta asignación se puede escribir la siguiente llamada a la función:

```
(*p) (12, 45)
```

Su efecto es el mismo que

```
ejemplo(12, 45)
```

También se puede omitir el asterisco (así como los paréntesis) en la llamada `(*p) (12, 45)`: convirtiéndose en esta otra llamada.

```
p(12, 45)
```

La *utilidad de las funciones a punteros* se ve más claramente si se imagina un programa grande, al principio del cual se desea elegir una entre varias funciones, de modo que la función elegida se llama, entonces, muchas veces. Mediante un puntero, la elección sólo se hace una vez: después de asignar (la dirección de) la función seleccionada a un puntero y a continuación se puede llamar a través de ese puntero.

*Los punteros a funciones también permiten pasar una función como un argumento a otra función.* Para pasar el nombre de una función como un argumento función, se especifica el nombre de la función como argumento. Supongamos que se desea pasar la función `mifunc()` a la función `sufunc()`. El código siguiente realiza las tareas anteriores:

```
void sufunc(int (*f)()); /* prototipo de sufunc */
int mifunc(int i); /* prototipo de mifunc */
void main()
{
 sufunc(mifunc);
}

int mifunc(int i)
{
 return 5*i;
}
```

En la función llamada se declara la función pasada como un puntero función.

```
void sufunc(int (*f)())
{
 ...
 j = f(5);
 ...
}
```

Como ejemplo práctico veamos cómo escribir una función general que calcule la suma de algunos valores, es decir,

$$f(1) + f(2) + \dots + f(n)$$

para cualquier función  $f$  que devuelva el tipo `double` y con un argumento `int`. Diseñaremos una función `funcsuma` que tiene dos argumentos:  $n$ , el número de términos de la suma, y  $f$ , la función utilizada. Así pues, la función `funcsuma` se va a llamar dos veces, y va a calcular la suma de

$$\begin{aligned} \text{inversos}(k) &= 1.0/k && \{\text{para } k = 1, 2, 3, 4, 5\} \\ \text{cuadrados}(k) &= k && \{\text{para } k = 1, 2, 3\} \end{aligned}$$

El programa siguiente muestra la función `funcsuma`, que utiliza la función  $f$  en un caso para inversos y en otro para cuadrados.

```
#include <stdio.h>
/* prototipos de funciones */
double inversos(int k);
double cuadrados(int k);
double funcsuma(int n, double (*f)(int k));

int main()

 printf("Suma de cinco inversos: %.3lf \n", funcsuma(5, inversos));
 printf("Suma de tres cuadrados: %.3lf \n", funcsuma(3, cuadrados));
 return 0;
}

double funcsuma(int n, double (*f)(int k))

 double s = 0;
 int i;
 for (i = 1; i <= n; i++)
 s += f(i);
 return s;
}

double inversos(int k)
{
 return 1.0/k;
}

double cuadrados(int k)

 return (double)k * k;
}
```

El programa anterior calcula las sumas de

$$a) \quad 1 + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.0}{4} + \frac{1.0}{5}$$

$$b) \quad 1.0 + 4.0 + 9.0$$

y su salida será:

```
Suma de cinco inversos: 2.283
Suma de tres cuadrados: 14.000
```

## 10.11.2. Aplicación de punteros a función para ordenación

Algunas de las funciones de la biblioteca, tal como `qsort()` o `bsearch()`, requieren pasar un argumento que consta de un puntero a una función. Se debe pasar a ambas, `qsort()` y `bsearch()`, un puntero de función que apunta hacia una función que se debe definir. `qsort()` utiliza el algoritmo de ordenación rápida (*quicksort*) para ordenar un array de cualquier tipo de dato. `bsearch()` utiliza la búsqueda binaria para determinar si un elemento está en un array. La función que debe proporcionarse es para realizar comparaciones de elementos de array. En el programa siguiente, la función `comparar()` se pasa a `qsort()` y a `bsearch()`. La función `comparar()` compara entradas del array `tabla` y devuelve (retorna) un número negativo si `arg1` es menor que `arg2`, devuelve cero si son iguales, o un número positivo si `arg1` es mayor que `arg2`.

El programa siguiente ordena un array de números enteros y busca si existe un valor clave.

```
#include <stdio.h>
#include <search.h>
#include <stdlib.h>
#include <time.h>

int comparar(const void *arg1, const void *arg2);

void main()

 int i, x;
 int tabla[15];
 int *b;

 randomize();
 /* genera tabla de elementos aleatorios de 1 a 100 */
 for (i = 0; i < 15; i++)
 tabla[i] = random(100)+1;
 printf("\n\nLista original: ");
 for (i = 0; i < 15; i++)
 printf("%d ", tabla[i]);

 /* Ordena tabla utilizando el algoritmo quicksort */
 qsort((void *)tabla, (size_t)15, sizeof(int), comparar);

 printf("\n\nLista ordenada: ");
 for (i = 0; i < 15; i++)
 printf("%d ", tabla[i]);

 printf("\n\nClave a buscar: ");
 scanf("%d", &x);
 /* Realiza una búsqueda binaria en el vector ordenado */
 b = bsearch(&x, (void *)tabla, (size_t)15, sizeof(int), comparar);

 if (b)
 /* clave encontrada */
 printf("\nEl elemento %d está en la tabla", x);
 else
 printf("\nEl elemento %d no está en la tabla", x);

 printf("\n\nPulsa cualquier tecla para continuar ");
 i = getch();
}
```

```

/* Comparar dos elementos de la lista */
int comparar(const void *arg1, const void *arg2)
{
 return *(int *) arg1 - *(int *) arg2;
}

```

### Recuerde

Los parámetros de la función `qsort()` y `bsearch()` son:

- `&x` Dirección de la clave a buscar.
- `(void *) tabla` Array que contiene valores a ordenar.
- `(size_t) 15` Número de elementos del array.
- `sizeof(int)` Tamaño en bytes de cada elemento del array.
- `comparar()` Nombre de la función que compara dos elementos del array.

### 10.11.3. Arrays de punteros de funciones

Ciertas aplicaciones requieren disponer de numerosas funciones, basadas en el cumplimiento de ciertas condiciones. Un método para implementar tal aplicación es utilizar una sentencia `switch` con muchos selectores `case`. Otra solución es utilizar un array de punteros de función. Se puede seleccionar una función de la lista y llamarla.

La sintaxis general de un array de punteros de función es:

```
tipoRetorno (*PunteroFunc[LongArray]) (<Lista de parámetros>);
```

---

#### Ejemplo 10.8

```

double (*fp[3]) (int n);
void (*ordenar[MAX-ORD]) (int* ArrayEnt, unsigned n);

```

---

`fp` apunta a un array de funciones; cada miembro devuelve un valor `double` y tiene un único parámetro de tipo `int`. `ordenar` es un puntero a un array de funciones; cada miembro devuelve un tipo `void` y toma dos parámetros: un puntero a `int` y un `unsigned`.

### Recuerde

- `func`, nombre de un elemento.
- `func[]` es un array.
- `(*func[])` es un array de punteros.
- `(*func[])()` es un array de punteros a funciones.
- `int (*func[])()` es un array de punteros a funciones que devuelven valores `int`.

Se puede asignar la dirección de las funciones al array, proporcionando las funciones que ya han sido declaradas. Un ejemplo es

```

int func1(int i, int j);
int func2(int i, int j);
int (*func[])(int, int) = {func1, func2};

```

### 10.11.4. Una aplicación de punteros de funciones

El listado siguiente, `CALCULA.C`, es un programa que simula calculador que puede sumar, restar, multiplicar o dividir números. Se escribe una expresión simple por teclado y el programa visualiza la respuesta.

El programa define cuatros funciones: `sumar()`, `restar()`, `mult()` y `div()`, y un array de punteros a función que se inicializa a cada una de las funciones. Se pide la operación a realizar, se busca el índice del puntero a función que le corresponde (dependiendo del operador) y se realiza la llamada a la función con su puntero.

```
#include <stdio.h>
/* prototipos de funciones */
float sumar(float x, float y);
float restar(float x, float y);
float mult(float x, float y);
float div(float x, float y);

void main()
{
 char signo, operadores[] = {'+', '-', '*', '/'};
 float(*func[])(float, float) = {sumar, restar, mult, div};
 int i;
 unsigned char t;
 float x, y;

 puts("\nCalculador de expresiones");
 do {
 printf("\nExpresión: ");
 scanf("%f %c %f",&x,&signo,&y);
 /* búsqueda del operador */
 for (i = 0; i < 4; i++)
 {
 if (signo == operadores[i])
 {
 printf("\n%.1f %c %.1f = %.2f", x, signo, y, func[i](x,y));
 I
 }
 printf("\nOtra expresion?: ");
 scanf("%*c%c",&t);t=tolower(t);
 }while (t=='s');
 }

 float sumar(float x, float y)
 {
 return x + y;
 }

 float restar(float x, float y)
 {
 return x - y;
 }

 float mult(float x, float y)
 {
 return x * y;
 }
}
```

```
float div(float x, float y)
{
 return x / y;
}
```

## 10.12. PUNTEROS A ESTRUCTURAS

Un puntero también puede apuntar a una estructura. Se puede declarar un puntero a una estructura tal como se declara un puntero a cualquier otro objeto y se declara un puntero estructura tal como se declara cualquier otra variable estructura: especificando un puntero en lugar del nombre de la variable estructura.

```
struct persona
{
 char nombre[30];
 int edad;
 int altura;
 int peso;
};
struct persona empleado = {"Amigo, Pepe", 47, 182, 85};
struct persona *p; /* se crea un puntero de estructura */
p = &empleado;
```

Cuando se referencia un miembro de la estructura utilizando el nombre de la estructura, se especifica la estructura y el nombre del miembro separado por un punto (.). Para referenciar el nombre de una persona, utilice empleado.nombre. Se referencia una estructura utilizando el puntero estructura. Se utiliza el operador -> para acceder a un miembro de ella.

---

### Ejemplo 10.9

*En este ejemplo se declara el tipo estructura t\_persona, que se asocia con el tipo persona para facilidad de escritura. Un array de esta estructura se inicializa con campos al azar y se muestran por pantalla.*

```
#include <stdio.h>

struct t_persona
{
 char nombre[30];
 int edad;
 int altura;
 int peso;
};
typedef struct t_persona persona;

void mostrar_persona(persona *ptr)
void main()
{
 int i;
 persona empleados[] = { {"Mortí er, Pepe", 47, 182, 85},
 {"García, Luis", 39, 170, 75},
 {"Jiménez, Tomás", 18, 175, 80} };
 persona *p; /* puntero a estructura */
 p = empleados;
```

```

for (i = 0; i < 3; i++, p++)
 mostrar_persona(p);
}

void mostrar_persona(persona *ptr)
{
 printf("\nNombre: %s", ptr -> nombre);
 printf("\tEdad: %d ", ptr -> edad);
 printf("\tAltura: %d ", ptr -> altura);
 printf("\tPeso: %d\n", ptr -> peso);
}

```

Al ejecutar este programa se visualiza la salida siguiente:

```

Nombre: Mortimer, Pepe Edad: 47 Altura: 180 Peso: 85
Nombre: García, Luis Edad: 39 Altura: 170 Peso: 75
Nombre: Jiménez, Tomás Edad: 18 Altura: 175 Peso: 80

```

## 10.13. RESUMEN

Los punteros son una de las herramientas más eficientes para realizar aplicaciones en C. Aunque su práctica puede resultar difícil y tediosa es, sin lugar a dudas, una necesidad vital su aprendizaje si desea obtener el máximo rendimiento de sus programas.

En este capítulo habrá aprendido los siguientes conceptos:

- Un puntero es una variable que contiene la dirección de una posición en memoria.
- Para declarar un puntero se sitúa un asterisco entre el tipo de dato y el nombre de la variable, como en `int *p`.
- Para obtener el valor almacenado en la dirección utilizada por el puntero, se utiliza el operador de indirección (\*). El valor de `p` es una dirección de memoria y el valor de `*p` es el dato entero almacenado en esa dirección de memoria.
- Para obtener la dirección de una variable existente, se utiliza el operador de dirección (&).
- Se debe declarar un puntero antes de su uso.
- Un puntero `void` es un puntero que no se asigna a un tipo de dato específico y puede, por consiguiente, utilizarse para apuntar a tipos de datos diferentes en diversos lugares de su programa.
- Para inicializar un puntero que no apunta a nada, se utiliza la constante `NULL`.
- Estableciendo un puntero a la dirección del primer elemento de un array, se puede utilizar el puntero para acceder a cada elemento del array de modo secuencial.

Así mismo, se han estudiado los conceptos de aritmética de punteros, punteros a funciones, punteros a estructuras y arrays de punteros.

## 10.14. EJERCICIOS

- 10.1. Encuentra los errores en la siguiente declaración de punteros:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= 'C';
float x;
void* r = &x;
```

- 10.2. Dada la siguiente declaración, escribir una función que tenga como argumento un puntero al tipo de dato y muestre por pantalla los campos.

```
struct boton
{
 char* rotulo;
 int codigo;
};
```

- 10.3. ¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?

- 10.4. Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

- 10.5. En el siguiente código se accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de punteros.

```
#define N 4
#define M 5
int f,c;
double mt[N][M];
. . .
for (f = 0; f<N; f++)
{
 for (c = 0; c<M; c++)
 printf("%lf ", mt[f][c]);
 printf("\n");
}
```

- 10.6. Escribe una función con un argumento de tipo puntero a double y otro argumento de tipo int. El primer argumento se debe de corresponder con un array y el segundo con el número de elementos del array. La función ha de ser de tipo puntero a double para devolver la dirección del elemento menor.

- 10.7. Dada la siguiente función:

```
double* gorta(double* v, int
m, double k)
{
 int j;
 for (j = 0; j<m ; j++)
 if (*v == k)
 return v;
 return 0;
}
```

- ¿Hay errores en la codificación? ¿De qué tipo?

Dadas las siguientes definiciones:

```
double w[15], x, z;
void *r;
```

- ¿Es correcta la siguiente llamada a la función?:

```
r = gorta(w,10,12.3);
```

- ¿Y estas otras llamadas?:

```
printf("%lf", *gorta(w,15,10.5));
z = gorta(w,15,12.3);
```

- 10.8. ¿Qué diferencias se pueden encontrar entre estas dos declaraciones?:

```
float mt[5][5];
float *m[5];
```

¿Se podría hacer estas asignaciones?:

```
m = mt;
m[1] = mt[1];
m[2] = &mt[2][0];
```

- 10.9. Dadas las siguientes declaraciones de estructuras, escribe cómo acceder al campo x de la variable estructura t.

```
struct fecha
{
 int d, m, a;
 float x;
};
struct dato
{
 char* mes;
 struct fecha* r;
} t;
```

## 10.15. PROBLEMAS

¿Qué problemas habría en la siguiente sentencia?

```
gets(c,mes);
cia;
El prototipo de una función es: void es-
cribe_mat(int** t, int mf, int
nc);
```

La función tiene como propósito mostrar por pantalla la matriz. El primer argumento se corresponde con una matriz entera, el segundo y tercero es el número de filas y columnas de la matriz. Escribe la implementación de la función aplicando la aritmética de punteros.

10.1. Escribe un programa en el que se lean 20 líneas de texto, cada línea con un máximo de 80 caracteres. Muestra por pantalla el número de vocales que tiene cada línea.

10.2. Escribe un programa que encuentre una matriz de números reales simétrica. Para ello, una función entera con entrada la matriz determinará si ésta es simétrica. En otra función se generará la matriz con números aleatorios de 1 a 19.

Utilizar la aritmética de punteros en la primera función, en la segunda indexación.

10.3. En una competición de natación se presentan 16 nadadores. Cada nadador se caracteriza por su nombre, edad, prueba en la que participa y tiempo (minutos, segundos) de la prueba. Escribe un programa que realice la entrada de los datos y calcule la desviación estándar respecto al tiempo. Para la entrada de datos, definir una función que lea del dispositivo estándar el nombre, edad, prueba y tiempo.

10.4. Se quiere evaluar las funciones  $f(x)$ ,  $g(x)$  y  $z(x)$  para todos los valores de  $x$  en el intervalo  $0 \leq x < 3.5$  con incremento de 0.2. Escribe un programa que evalúe dichas funciones, utilizar un array de punteros a función. Las funciones son las siguientes:

$$f(x) = 3 \cdot e^x - 2x$$

$$g(x) = -x \cdot \sin(x) + 1.5$$

$$z(x) = x^2 - 2x + 1$$

10.5. Se quiere sumar enteros largos, con un número de dígitos que supera el máximo entero largo. Los enteros tienen un máximo de 40 dígitos. Para solventar el problema se utilizan

10.6. Escribe una función que tenga como entrada una cadena y devuelva un número real. La cadena contiene los caracteres de un número real en formato decimal (por ejemplo, la cadena "25.56" se ha de convertir en el correspondiente valor real).

10.7. Escribe un programa para generar una matriz de  $4 \times 5$  elementos reales, multiplicar la primera columna por cualquier otra y mostrar la suma de los productos. El programa debe de descomponerse en subproblemas y utilizar punteros para acceder a los elementos de la matriz.

10.8. Desarrolla un programa en C que use una estructura para la siguiente información sobre un paciente de un hospital: nombre, dirección, fecha de nacimiento, sexo, día de visita y problema médico. El programa debe tener una función para entrada de los datos de un paciente, guardar los diversos pacientes en un array y mostrar los pacientes cuyo día de visita sea uno determinado.

10.9. Escribe un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. Utilizar un array de punteros de funciones, siendo las funciones las que permitan calcular el área.

10.10. Se tiene la ecuación  $3 \cdot e^x - 7x = 0$ , para encontrar una raíz (una solución) escribir tres funciones que implementen, respectivamente, el método de Newton, Regula-Falsi y Bisección. Mediante puntero de función aplicar uno de estos métodos para encontrar una raíz de dicha ecuación.

# CAPÍTULO 11

## ASIGNACIÓN DINÁMICA DE MEMORIA

---

### CONTENIDO

- 11.1. Gestión dinámica de la memoria.
- 11.2. Funcion de asignación de memoria `malloc()`.
- 11.3. La función `free()`.
- 11.4. Funciones de asignacion `calloc()` y `realloc()`.
- 11.6. Asignación dinámica para arrays.
- 11.6. Arrays dinámicos.
- 11.7. Regias de funcionamiento de funciones de asignación dinámica.
- 11.8. *Resumen.*
- 11.8. *Ejercicios.*
- 11.10. *Problemas.*

---

## INTRODUCCIÓN

Los programas pueden crear variables globales o locales. Las variables declaradas globales en sus programas se almacenan en posiciones fijas de memoria, en la zona conocida como *segmento de datos* del programa, y todas las funciones pueden utilizar estas variables. **Las** variables locales se almacenan en la **pila** (*stack*) y existen *sólo* mientras están activas las funciones que están declaradas. Es posible, también, crear variables static (similares a las globales) que se almacenan en posiciones fijas de memoria, pero sólo están disponibles en el módulo (es decir, el archivo de texto) o función en que se declaran; su espacio de almacenamiento es el segmento de **datos**.

Todas estas clases de variables comparten una característica común: se definen cuando se compila el programa. Esto significa que el compilador reserva (define) espacio para almacenar valores de los tipos de datos declarados. Es decir, en el caso de las variables globales y locales se ha de indicar al compilador exactamente cuántas y de qué tipo son las variables a asignar. O sea, el espacio de almacenamiento se reserva en el momento de la compilación.

**Sin** embargo, no siempre es posible conocer con antelación a la ejecución cuanta memoria se debe reservar al programa. En C, se asigna memoria en el momento de la ejecución en el *montículo* o *montón* (**heap**), mediante las funciones **malloc()**, **realloc()**, **calloc()** y **free()**, que asignan y liberan la memoria de una zona denominada **almacén libre**.

---

## CONCEPTOS CLAVE

- o Array dinámico.
- o Array estático.
- o Desbordamiento de memoria.
- o Función **free**.
- o Función **malloc**.
- o Gestión dinámica.
- o Puntero genérico.
- o Variable apuntada.

## 11.1. GESTIÓN DINÁMICA DE LA MEMORIA

Consideremos un programa que evalúe las calificaciones de los estudiantes de una asignatura. El programa almacena cada una de las calificaciones en los elementos de una lista o tabla (*array*) y el tamaño del array debe ser lo suficientemente grande para contener el total de alumnos matriculados en la asignatura. Por ejemplo, la sentencia

```
int asignatura [40];
```

reserva 40 enteros, un número fijo de elementos. Los arrays son un método muy eficaz cuando se conoce su longitud o tamaño en el momento de escribir el programa. Sin embargo, presentan un grave inconveniente si el tamaño del array *sólo* se conoce en el momento de la ejecución. Las sentencias siguientes producirían un error durante la compilación:

```
scanf("%d",&num_estudiantes);
int asignatura[num_estudiantes];
```

ya que el compilador requiere que el tamaño del array sea constante. Sin embargo, en numerosas ocasiones no se conoce la memoria necesaria hasta el momento de la ejecución. Por ejemplo, si se desea almacenar una cadena de caracteres tecleada por el usuario, no se puede prever, *a priori*, el tamaño del array necesario, a menos que se reserve un array de gran dimensión y se malgaste memoria cuando no se utilice. En el ejemplo anterior, si el número de alumnos de la clase aumenta, se debe variar la longitud del array y volver a compilar el programa. El método para resolver este inconveniente es recurrir a punteros y a técnicas de *asignación dinámica de memoria*.

El espacio de la variable asignada dinámicamente se crea durante la ejecución del programa, al contrario que en el caso de una variable local cuyo espacio se asigna en tiempo de compilación. La asignación dinámica de memoria proporciona control directo sobre los requisitos de memoria de su programa. El programa puede crear o destruir la asignación dinámica en cualquier momento durante la ejecución. Se puede determinar la cantidad de memoria necesaria en el momento en que se haga la asignación. Dependiendo del modelo de memoria en uso, se pueden crear variables mayores de 64 K.

El código del programa compilado se sitúa en segmentos de memoria denominados *segmentos de código*. Los datos del programa, tales como variables globales, se sitúan en un área denominada *segmento de datos*. Las variables locales y la información de control del programa se sitúan en un área denominada *pila*. La memoria que queda se denomina *memoria del montículo o almacén libre*. Cuando el programa solicita memoria para una variable dinámica, se asigna el espacio de memoria deseado desde el montículo.

### **Error típico de programación en C**

La declaración de un array exige especificar su longitud como una expresión constante, así *str* declara un array de 100 elementos:

```
char str[100];
```

Si se utiliza una variable en la expresión que determina la longitud de un array, se producirá un error.

```
int n;
...
scanf("%d",&n);
char str[n]; /* error */
```

### 11.1.1. Almacén libre (*freestore*)

El mapa de memoria del modelo de un programa grande es muy similar al mostrado en la Figura 11.1. El diseño exacto dependerá del modelo de programa que se utilice. Para grandes modelos de datos, el almacén libre (*heap*) se refiere al área de memoria que existe dentro de la pila del programa. Y el almacén libre es, esencialmente, toda la memoria que queda libre después de que se carga el programa.

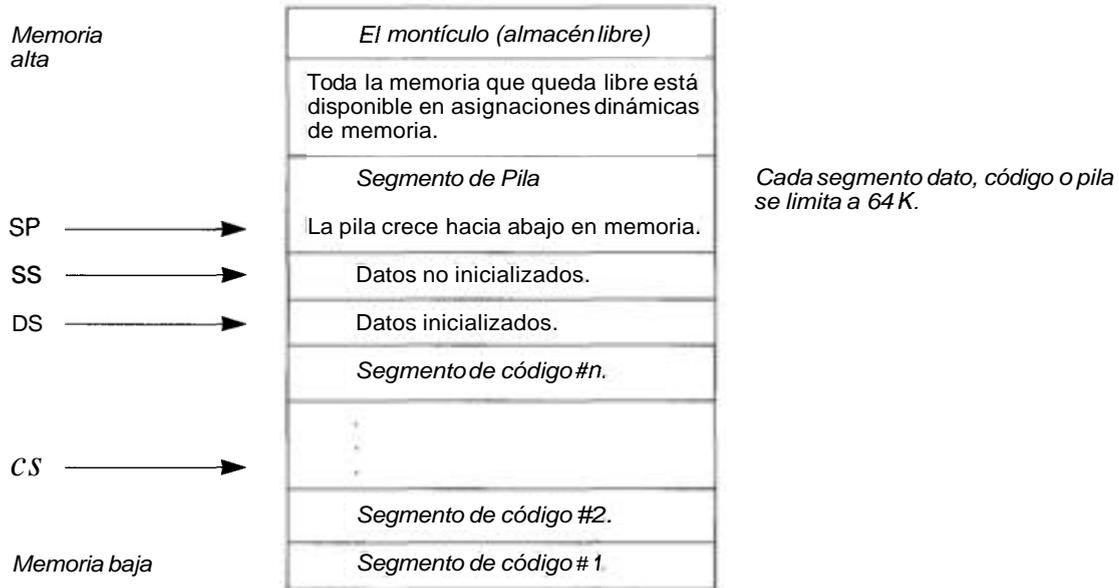


Figura 11.1. Mapa de memoria de un programa.

En C las funciones `malloc()`, `realloc()`, `calloc()` y `free()` asignan y liberan memoria de un bloque de memoria denominado el *montículo del sistema*. Las funciones `malloc()`, `calloc()` y `realloc()` asignan memoria utilizando *asignación dinámica* debido a que puede gestionar la memoria durante la ejecución de un programa; estas funciones requieren, generalmente, moldeado (conversión de tipos).

## 11.2. FUNCIÓN `malloc()`

La forma más habitual de C para obtener bloques de memoria es mediante la llamada a la función `malloc()`. La función asigna un bloque de memoria que es el número de bytes pasados como argumento. `malloc()` devuelve un puntero, que es la dirección del bloque asignado de memoria. El puntero se utiliza para referenciar el bloque de memoria y devuelve un puntero del tipo `void*`. La forma de llamar a la función `malloc()` es:

```
puntero = malloc(tamaño en bytes);
```

Generalmente se hará una conversión al tipo del puntero:

```
tipo *puntero;
puntero =(tipo *)malloc(tamaño en bytes);
```

Por ejemplo:

```
long* p;
p = (long*)malloc(32);
```

El operador unario `sizeof` se utiliza con mucha frecuencia en las funciones de asignación de memoria. El operador se aplica a un tipo de dato (o una variable), el valor resultante es el número de bytes que ocupa. Así, si se quiere reservar memoria para un buffer de 10 enteros:

```
int *r;
r = (int*) malloc(10*sizeof(int));
```

Al llamar a la función `malloc()` puede ocurrir que no haya memoria disponible, en ese caso `malloc()` devuelve `NULL`.

### Sintaxis de llamada a `malloc()`

```
tipo *puntero;
puntero = (tipo*)malloc(tamaño);
```

La función devuelve la dirección de la variable asignada dinámicamente, el tipo que devuelve es `void*`.

### Prototipo que incluye `malloc()`

```
void* malloc(size_t n);
```

Figura 10.2. Sintaxis (formato) de la función `malloc()`.

En la sintaxis de llamada, ***puntero*** es el nombre de la variable puntero a la que se asigna la dirección del objeto dato, o se le asigna la dirección de memoria de un bloque lo suficientemente grande para contener un array de  $n$  elementos, o `NULL`, si falla la operación de asignación de memoria. El siguiente código utiliza `malloc()` para asignar espacio para un valor entero:

```
int *pEnt;
...
pEnt = (int*) malloc(sizeof(int));
```

La llamada a `malloc()` asigna espacio para un `int` (entero) y almacena la dirección de la asignación en `pEnt`. `pEnt` apunta ahora a la posición en el almacén libre (montículo) donde se establece la memoria. La Figura 10.3 muestra como `pEnt` apunta a la asignación del almacén libre. Así, por ejemplo, para reservar memoria para un array de 100 números reales:

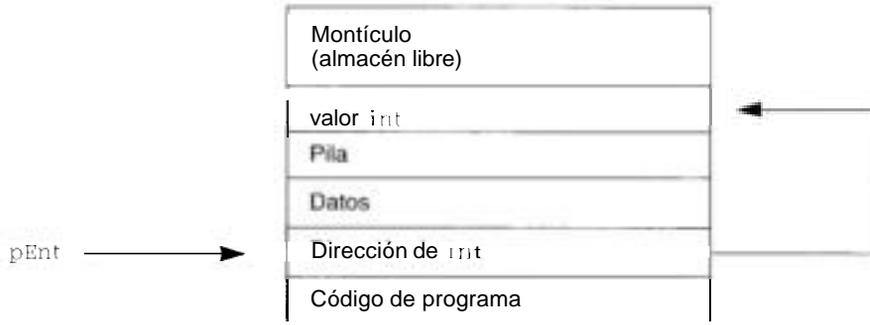
```
float *BloqueMem;
BloqueMem = (float*)malloc(100*sizeof(float));
```

En el ejemplo se declara un puntero denominado `BloqueMem` y lo inicializan a la dirección devuelta por `malloc()`. Si un bloque del tamaño solicitado está disponible, `malloc()` devuelve un puntero al principio de un bloque de memoria del tamaño especificado. Si no hay bastante espacio de almacenamiento dinámico para cumplir la petición, `malloc()` devuelve cero o `NULL`. La reserva de  $n$  caracteres se puede declarar así:

```
int n;
char *s;

scanf("%d",&n);
s = (char*) malloc(n*sizeof(char));
```

La función `malloc()` está declarada en el archivo de cabecera `stdlib.h`.



**Figura 11.3.** Después de `malloc()`, con el tamaño de un entero, `pEnt` apunta a la posición del montículo donde se ha asignado espacio para el entero.

### Ejemplo 11.1

En el siguiente ejemplo se lee una línea de caracteres, se reserva memoria para un buffer de tantos caracteres como los leídos y se copia en el buffer la cadena.

```
#include <stdio.h>
#include <string.h> /* por el uso de strcpy() */

void main()
{
 char cad[121], *ptr;
 int lon;

 puts("\nIntroduce una línea de texto\n");
 gets(cad);

 lon = strlen(cad);
 ptr = (char*) malloc((lon+1)*sizeof(char));

 strcpy(ptr, cad); /* copia cad a nueva área de memoria
 apuntada por ptr */
 printf("ptr = %s", ptr); /* cad está ahora en ptr */
 free(ptr); /* libera memoria de ptr */
}
```

#### La expresión

```
ptr = (char*) malloc((lon+1)*sizeof(char));
```

devuelve un puntero que apunta a una sección de memoria capaz de contener la cadena de longitud `strlen()` más un byte extra por el carácter `'\0'` al final de la cadena.

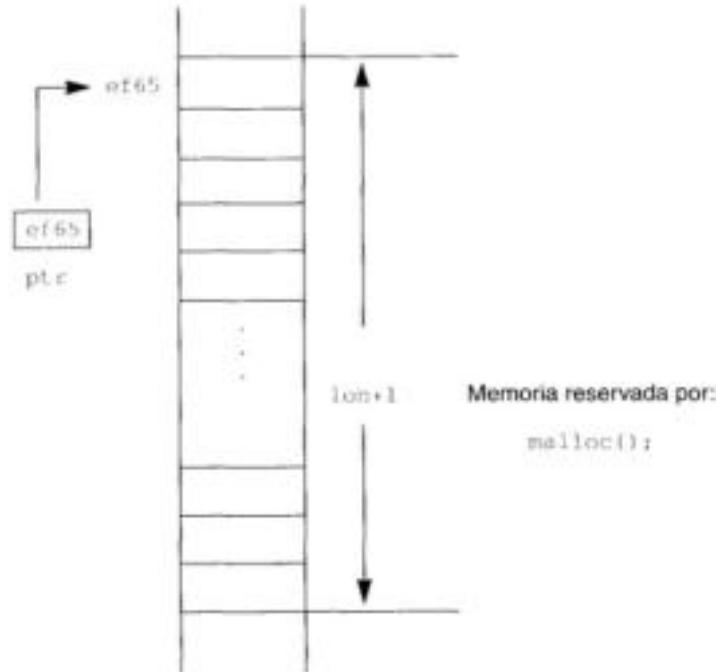


Figura 11.4. Memoria obtenida por función malloc().

### Precaución

El almacenamiento libre no es una fuente inagotable de memoria. Si la función `malloc()` se ejecuta con falta de memoria, se devuelve un puntero `NULL`. Es responsabilidad del programador comprobar *siempre* el puntero para asegurar que es válido, antes de que se asigne un valor al puntero. Supongamos, por ejemplo, que se desea asignar un array de 1.000 números reales en doble precisión:

```
#define TOPE 1999
double *ptr_lista;
int i;
ptr_lista = (double*)malloc(1000*sizeof(double));
if (ptr_lista == NULL)
{
 puts("Error en la asignación de memoria");
 return -1; /* intentar recuperar memoria */
}

for (i = 0; i < 1000; i++)
 ptr_lista[i] = (double)*random(TOPE);
```

Si no existe espacio de almacenamiento suficiente, la función `malloc()` devuelve `NULL`. La escritura de un programa totalmente seguro, exige comprobar el valor devuelto por `malloc()` para asegurar que no es `NULL`. `NULL` es una constante predefinida en C. Se debe incluir los archivos de cabecera `<stdlib.h>` para obtener la definición de `NULL`.

## Ejemplo 11.2

El programa *TESTMEM* comprueba la cantidad de memoria que se puede asignar dinámicamente (está disponible). Para ello se llama a `malloc()`, solicitando en cada llamada 1.000 bytes de memoria.

```

/*
TESTMEM: programa para determinar memoria libre.
*/
#include <stdio.h>

int main()
{
 void *p;
 int i;
 long m = 0;
 for (i = 1; ; i++)
 {
 p = malloc(1000);
 if (p == NULL) break;
 m += 1000;
 }
 printf("\nTotal de memoria asignada %d\n",m) ;
 return 0;
}

```

Se asigna repetidamente 1 **kB (Kilobytes)** hasta que falla la asignación de memoria y el bucle se termina.

### 11.2.1. Asignación de memoria de un tamaño desconocido

Se puede invocar a la función `malloc()` para obtener memoria para un array, incluso si no se conoce con antelación cuanta memoria requieren los elementos del array. Todo lo que se ha de hacer es invocar a `malloc()` en tiempo de ejecución, pasando como argumento el número de elementos del array multiplicado por el tamaño del tipo del array. El número de elementos se puede solicitar al usuario y leerse en tiempo de ejecución. Por ejemplo, este segmento de código asigna memoria para un array de  $n$  elementos de tipo `double`, el valor de  $n$  se conoce en tiempo de ejecución:

```

double *ad;
int n;
printf("Número de elementos del array: ");
scanf("%d",&n);
ad = (double*)malloc(n*sizeof(double));

```

En este otro ejemplo se declara un tipo de dato complejo, se solicita cuántos números complejos se van a utilizar, se reserva memoria para ellos y se comprueba que existe memoria suficiente. Al final, se leen los  $n$  números complejos.

```

struct complejo
{
 float x, y;
};
int n, j;
struct complejo *p;

printf("Cuántos números complejos: ");
scanf("%d",&n);

```

```

p = (struct complejo*) malloc(n*sizeof(struct complejo));
if (p == NULL)
{
 puts("Fin de ejecución. Error de asignación de memoria.");
 exit(-1);
}
for (j = 0; j < n; j++, p++)
{
 printf("Parte real e imaginaria del complejo %d : ", j);
 scanf ("%f %f", &p->x, &p->y);
}

```

### 11.2.2. Uso de malloc() para arrays multidimensionales

Un array bidimensional es, en realidad, un array cuyos elementos son arrays. Al ser el nombre de un array unidimensional un puntero constante, un array bidimensional será un puntero a puntero constante (tipo \*\*). Para asignar memoria a un array multidimensional, se indica cada dimensión del array de igual forma que se declara un array unidimensional. En el Ejemplo 11.3 se reserva memoria en tiempo de ejecución para una matriz de  $n$  filas y para cada fila  $m$  elementos.

---

#### Ejemplo 11.3

```

/* matriz de n filas y cada fila de un número variable de elementos */
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int **p ;
 int n,m,i;

 do {
 printf("\n Numero de filas: "); scanf("%d",&n);
 } while (n<=0);
 p = (int**) malloc(n*sizeof(int*));
 for (i = 0; i < n; i++)
 {
 int j;
 printf("Número de elementos de fila %d ", i+1);
 scanf("%d",&m);
 p[i] = (int*)malloc(m*sizeof(int));
 for (j = 0; j < m; j++)
 scanf("%d",&p[i][j]);
 }

 return 1;
}

```

---

En el ejemplo, la sentencia `p = (int**) malloc(n*sizeof(int*))`; reserva memoria para un array de  $n$  elementos, cada elemento es un puntero a entero (`int*`). Cada iteración del bucle `for` externo requiere por teclado, el número de elementos de la fila ( $m$ ); reserva memoria para esos  $m$  elementos con la sentencia `p[i] = (int*)malloc(m*sizeof(int))`; a continuación lee los datos de la fila.

### 11.3. LIBERACIÓN DE MEMORIA, FUNCIÓN `free()`

Cuando se ha terminado de utilizar un bloque de memoria previamente asignado por `malloc()`, u otras funciones de asignación, se puede liberar el espacio de memoria y dejarlo disponible para otros usos, mediante una llamada a la función `free()`. El bloque de memoria suprimido se devuelve al espacio de almacenamiento libre, de modo que habrá más memoria disponible para asignar otros bloques de memoria. El formato de la llamada es

```
free (puntero)
```

Así, por ejemplo, para las declaraciones

```
1. int *ad;
 ad = (int*)malloc(sizeof(int));

2. char *adc;
 adc = (char*) malloc(100*sizeof(char));
```

el espacio asignado se puede liberar con las sentencias

```
free(ad);
```

Y

```
free(adc);
```

#### Sintaxis de llamada a `free()`

```
tipo *puntero;
...
free (puntero);
```

**La variable puntero puede apuntar a una dirección de memoria de cualquier tipo.**

#### Prototipo que incluye `free()`

```
void free(void *);
```

Figura 11.5. Sintaxis (formato) de la función `Free()`.

---

#### Ejemplo 11.4

*En este ejemplo se reserva memoria para un array de 10 estructuras; después se libera la memoria reservada.*

```
struct gato *pgato; /* declara puntero a la estructura gato */

pgato = (struct gato*)malloc(10*sizeof(struct gato));
if (pgato == NULL)
 puts ("Memoria agotada");
else
{
 ...
 free (pgato); /* Liberar memoria asignada a pgato */
}
```

---

## 11.4. FUNCIONES DE ASIGNACIÓN DE MEMORIA `calloc()` y `realloc()`

Además de la función `malloc()` para obtener bloques de memoria, hay otras dos funciones que permiten obtener memoria libre en tiempo de ejecución, éstas son `calloc()` y `realloc()`. Con ambas se puede asignar memoria, como con `malloc()`, cambia la forma de transmitir el número de bytes de memoria requeridos. Ambas devuelven un puntero al bloque asignado de memoria. El puntero se utiliza para referenciar el bloque de memoria. El puntero que devuelven es del tipo `void*`.

### 11.4.1. Función `calloc()`

La forma de llamar a la función `calloc()` es:

```
puntero = calloc(número elementos,tamaño de cada elemento);
```

Generalmente se hará una conversión al tipo del puntero:

```
tipo *puntero;
puntero =(tipo*)calloc(numero elementos,tamaño de cada elemento);
```

El tamaño de cada elemento se expresa en bytes, se utiliza para obtenerlo el operador `sizeof`. Por ejemplo, se quiere reservar memoria para 5 datos de tipo `double`:

```
#define N 5
double* pd;

pd = (double*) calloc(N,sizeof(double));
```

En este otro ejemplo se reserva memoria para una cadena variable:

```
char *c, B[121];
puts("Introduce una línea de caracteres.");
gets(B);
/* Se reserva memoria para el número de caracteres + 1 para el carácter
 fin de cadena.
*/
c = (char*) calloc(strlen(B)+1,sizeof(char));
strcpy(c,B);
```

Al llamar a la función `calloc()` puede ocurrir que no haya memoria disponible, en ese caso `calloc()` devuelve `NULL`.

#### Sintaxis de llamada a `calloc()`

```
tipo *puntero;
int numelementos;
...
puntero =(tipo*)calloc (numelementos,tamaño de tipo);
```

La función devuelve la dirección de la variable asignada dinámicamente, el tipo que devuelve es `void*`.

#### Prototipo que tiene `calloc()`

```
void* calloc(size_t n,size_t t);
```

Figura 11.6. Sintaxis (formato) de la función `calloc()`

En la sintaxis de llamada, *puntero* es el nombre de la variable puntero al que se asigna la dirección de memoria de un bloque de *numelementos*, o `NULL` si falla la operación de asignación de memoria.

La función `calloc()` está declarada en el archivo de cabecera `stdlib.h`, por lo que será necesario incluir ese archivo de cabecera en todo programa que llame a la función. Se puede reservar memoria dinámicamente para cualquier tipo de dato, incluyendo `char`, `int`, `float`, arrays, estructuras e identificadores de `typedef`.

En el siguiente programa se considera una secuencia de números reales, con una variable puntero a `float` se procesa un array de longitud variable, de modo que se puede ajustar la cantidad de memoria necesaria para el número de valores durante la ejecución del programa.

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

int main (void)
{
 float *pf = NULL;
 int num, i;

 do {
 printf("Número de elementos del vector: ");
 scanf("%d",&num);
 }while (num < 1);

 /* Asigna memoria: num*tamaño bytes */
 pf = (float *) calloc(num, sizeof(float));
 if (pf == NULL)
 {
 puts("Error en la asignación de memoria.");
 return 1;
 }
 printf ("\nIntroduce%d valores ",num);
 for (i=0; i<num; i++)
 scanf("%f",&pf[i]);

 /* proceso del vector */
 /* liberación de la memoria ocupada */
 free(pf);
 return 0;
}
```

### 11.4.2. Función `realloc()`

Esta función también es para asignar un bloque de memoria libre. Tiene una variación respecto a `malloc()` y `calloc()`, permite ampliar un bloque de memoria reservado anteriormente. La forma de llamar a la función `realloc()` es:

```
puntero = realloc(puntero a bloque, tamaño total de nuevo bloque);
```

Generalmente se hará una conversión al tipo del puntero:

```
tipo *puntero;
```

```
puntero =(tipo*)realloc(puntero a bloque, tamaño total nuevo bloque);
```

El tamaño del bloque se expresa en bytes. El puntero a *bloque* referencia a un bloque de memoria reservado previamente con `malloc()`, `calloc()` o la propia `realloc()`.

**Ejemplo 11.5**

*Reservar memoria para una cadena y a continuación, ampliar para otra cadena más larga.*

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

int main()
{
 char *cadena;
 int tam;
 tam = (strlen("Primavera")+1)*sizeof(char);
 cadena = (char*)malloc(tam);

 strcpy(cadena, "Primavera");
 puts(cadena);

 /* Amplia el bloque de memoria */

 tam += (strlen(" en Lupiana\n")+1)*sizeof(char);
 cadena = (char *) realloc(cadena, tam);
 strcat(cadena, " en Lupiana\n");
 puts(cadena);

 /* liberación de memoria */
 free(cadena);
 return 0;
}
```

El segundo argumento de `realloc()`, es el tamaño total que va a tener el bloque de memoria libre. Si se pasa cero (0) como tamaño se libera el bloque de memoria al que está apuntando el puntero primer argumento, y la función devuelve `NULL`. En el siguiente ejemplo se reserva memoria con `calloc()` y después se libera con `realloc()`.

```
#define N 10
long* pl;

pl = (long*) calloc(N, sizeof(long));
...
pl = realloc(pl, 0);
```

El puntero del primer argumento de `realloc()` puede tener el valor de `NULL`, en este caso la función `realloc()` reserva tanta memoria como la indicada por el segundo argumento, en definitiva, actúa como `malloc()`.

**Ejemplo 11.6**

*En este ejemplo se leen dos cadenas de caracteres; si la segunda cadena comienza por `COPIA` se añade a la primera. La memoria se reserva con `realloc()`.*

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

int main()
{
 char *C1=NULL, *C2=NULL, B[121];
 char *clave ="COPIA";
```

```

int tam;
puts("\n\t Primera cadena ");
gets(B);
tam = (strlen(B)+1)*sizeof(char);
C1 = (char*)realloc(C1,tam);
strcpy(C1,B);

puts("\n\t Segunda cadena ");
gets(B);
tam = (strlen(B)+1)*sizeof(char);
C2 = (char*)realloc(C2,tam);
strcpy(C2,B);

/* Compara los primeros caracteres de C2 con clave.
 La comparación se realiza con la función strcmp() */
if (strlen(clave) <= strlen(C2))
{
 int j;
 char *R = NULL;
 R = realloc (R,(strlen(clave)+1)*sizeof (char)) ;
 /* copia los strlen(clave) primeros caracteres */
 for (j=0; j<strlen(clave);j++)
 *(R+j) = *(C2+j);
 *(R+j) = '\0';
 /* compara con clave */
 if (strcmp(clave,R)==0)
 {
 /* amplia el bloque de memoria */
 tam = (strlen(C1)+strlen(C2)+1)*sizeof(char);
 C1 = realloc(C1,tam);
 strcat(C1,C2);
 }
}

printf("\nCadena primera: %s",C1);
printf("\nCadena segunda: %s",C2);
return 1;
}

```

Al llamar a la función `realloc()` para ampliar el bloque de memoria puede ocurrir que no haya memoria disponible; en ese caso `realloc()` devuelve `NULL`.

### Sintaxis de llamada a `realloc()`

tipo \*puntero;

puntero = (tipo\*)realloc (puntero,tamaño del bloque de memoria);

La función devuelve la dirección de la variable asignada dinámicamente, el tipo que devuelve es `void*`.

### Prototipo que tiene `realloc()`

`void* realloc(void* puntero,size_t t);`

Figura 11.7. Sintaxis (formato) de la función `realloc()`.

Hay que tener en cuenta que la expansión de memoria que realiza `realloc()` puede hacerla en otra dirección de memoria de la que contiene la variable puntero transmitida como primer argumento. En cualquier caso, `realloc()` copia los datos referenciados por puntero en la memoria expandida.

La función `realloc()`, al igual que las demás funciones de asignación de memoria, está declarada en el archivo de cabecera `stdlib.h`.

## 11.5. ASIGNACIÓN DE MEMORIA PARA ARRAYS

La gestión de listas y tablas mediante arrays es una de las operaciones más usuales en cualquier programa. La asignación de memoria para arrays es, en consecuencia, una de las tareas que es preciso conocer en profundidad.

El listado de `ASIGCADS.c` muestra cómo se puede utilizar la función `malloc()` para asignar memoria a un array de cadenas de longitud variable.

---

### Ejemplo 11.7

*El programa `ASIGCADS.c` lee  $n$  líneas de texto, reserva memoria según la longitud de la línea leída, cuenta las vocales de cada línea e imprime cada línea y el número de vocales que tiene.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define N 10

void salida(char*[], int*);
void entrada(char*[]);
int vocales(char*);

int main()
{
 char *cad[N];
 int j, voc[N];

 entrada(cad);
 /* Cuenta de vocales por cada linea */
 for (j = 0; j<N; j++)
 voc[j] = vocales(cad[j]);

 salida(cad, voc);
 return 0;
}

void entrada(char* cd[])
{
 char B[121];
 int j, tam;

 printf("\n\tEscribe %d lineas de texto\n",N);
 for (j = 0; j<N; j++)
 {
 gets(B);
 tam = (strlen(B)+1)*sizeof(char);
 cd[j] = (char*)malloc(tam);
 strcpy(cd[j],B);
 }
}
```

```

}

int vocales(char* c)
{
 int k, j;
 /* Cuenta vocales de la cadena c */
 for (j=k = 0; j<strlen(c); j++)
 switch (tolower(*(c+j)))
 {
 case 'a':;
 case 'e':;
 case 'i':;
 case 'o':;
 case 'u': k++;
 }
 return k;
}

void salida(char* cd[], int* v)
{
 int j;

 puts("\n\tSalida de las líneas junto al número de vocales");
 for (j = 0; j<N; j++)
 {
 printf("%s : %2d\n",cd[j],v[j]);
 }
}

```

El programa declara `char *cad[N]` como array de punteros a `char`, de tal forma que en la función `entrada()` se reserva memoria, con `malloc()`, para cada línea de texto.

### 11.5.1. Asignación de memoria interactivamente

El programa `ASIGMEM.C` muestra cómo se puede invocar a `calloc()` para asignar memoria para un array. Cuando se ejecuta el programa, se pide al usuario teclear el tamaño de un array, y si se contesta adecuadamente el programa genera un array de números enteros aleatorios. A su vez, genera otro array con los mismos valores pero sin duplicidades; este segundo array se crea dinámicamente con la función `realloc()`. La estrategia para reservar memoria es llamar a `realloc()` para expandir el array cada 10 valores; es decir, primero se asigna memoria para 10 valores y cuando se ha completado se asignan otros 10 y así sucesivamente.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define S 10
#define NUM 99

struct array
{
 int *v; /* puntero al array */
 int n; /* número de elementos del array */
};

typedef struct array vector; /* declaración del nuevo tipo: vector */
void gen_array(vector* inic); /* genera array con n valores */
void nuevo_array(vector inic, vector* nd); /* genera nuevo
vector sin duplicados */

```

```

void escribe_array(vector w);

int main()
{
 vector prim, dest;

 do {
 printf("\nNumero de elementos del array: ");
 scanf("%d",&prim.n);
 }while (prim.n<1);
 randomize();

 gen_array(&prim);
 escribe_array(prim);

 nuevo_array(prim,&dest);
 escribe_array(dest);

 return 0;

void gen_array(vector* inic)
{
 int k;
 inic->v = (int*)calloc(inic->n,sizeof(int)); /*reserva memoria */
 for (k = 0; k< inic->n; k++)
 inic->v[k] = random(NUM)+1; /* genera valores enteros de 1 a NUM */
}

void escribe_array(vector w)
{
 int k;
 printf("\n\t Valores que contiene el vector\n");
 for (k = 0; k< w.n; k++)
 printf("%d%c",w.v[k], (k+1)%19==0 ? '\n': ' '); /*cada 19 enteros salta
 de linea*/
}

void nuevo_array(vector inic, vector* nd)
{
 int k,tam;

 /* Reserva inicial de memoria para 10 valores */

 nd->v = NULL;
 tam = sizeof(int)*S;
 nd->v = (int*)realloc(nd->v,tam);
 /* copia el primer elemento */
 nd->v[0] = inic.v[0];
 nd->n = 1;

/* copia los demas elementos si no estan ya en el array.
 Cuenta los elementos copiados para reservar memoria */
 for (k = 1; k< inic.n; k++)

 int j,dup;
 j=dup= 0;
 while ((j<nd->n) && !dup)
 {
 dup = inic.v[k]==nd->v[j++];
 }
}

```

```

if (!dup)
{
 if (nd->n%S == 0) /* amplia memoria */
 {
 tam += sizeof(int)*S;
 nd->v =(int*)realloc(nd->v,tam);

 /*
 asigna el elemento. Los indices en C estan en el rango de 0 a n-1, por
 esa razon se asigna y despues se incrementa.
 */
 nd->v[nd->n++] = inic.v[k];
 }
}

```

### 11.5.2. Asignación de memoria para un array de estructuras

El programa ASIGNAES.C define varios modelos de estructuras para representar un curso de perfeccionamiento, al que asisten varios alumnos de diversos departamentos de una empresa. Se declara una estructura `persona`, una estructura `alumno`, otra `profesor` y la estructura `curso`. Un alumno es una `persona` y los campos `departamento` y `nivel`. El profesor es una `persona` y el campo `expe` años de experiencia. El curso consta de `N` alumnos y un profesor, además del número de días de duración y la descripción del curso. El programa utiliza funciones de asignación de memoria dinámica para asignar memoria que contenga las cadenas de caracteres y un array de `N` estructuras `alumno`; define una función que recibe una cadena y reserva memoria para contener la cadena; la función de biblioteca `strcpy()` se utiliza para copiar una constante de cadena en la memoria reservada. El programa da entrada a los datos referidos anteriormente y visualiza el contenido del curso.

```

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

typedef struct persona
{
 char* nom;
 int edad;
 char* dir;
} PERSONA;

typedef struct alumno
{
 PERSONA p;
 char* depar;
 short nivel;
} ALUMNO;

typedef struct profesor
{
 PERSONA p;
 short expe;
} PROFESOR;

struct curso
{

```

```

ALUMNO* ptral;
PROFESOR* pf;
char* descrip;
short dias;
short n; /* Numero de alumnos del curso */
I;

char* asigcad(void);
PERSONA* asigper(void);
PROFESOR* asigprof(void);
ALUMNO* asigalms(short n);

int main()
{
 struct curso dom;
 int j;

 printf("\n\tCurso de perfeccionamiento.\nDescripcion del curso: ");
 dom.descripcion = asigcad();
 printf("Dias lectivos del curso: ");
 scanf("%d%c",&dom.dias);

 printf("\t Datos del profesor del curso.\n");
 dom.pf = asigprof();

 printf("\t Numero de alumnos del curso: ");
 scanf("%d%c",&dom.n);

 dom.ptral = asigalms(dom.n);

 /* Mostra de los datos del curso */

 printf("\n\n\t\t Curso: %s\n",dom.descripcion);
 puts("\t\t --- ");
 printf("\t\tProfesor: %s\n",dom.pf->p.nom);
 printf("\t\tRelacion de asistentes al curso\n");
 for (j = 0; j<dom.n; j++)
 {
 printf("\t\t%s\n", (dom.ptral+j)->p.nom);
 }
 return 0;
}

char* asigcad()
{
 char b[121], *cd;
 gets(b);
 cd = (char*)malloc((strlen(b)+1)*sizeof(char));
 if (cd == NULL)
 {
 puts("\n\t!! Error de asignacion de memoria, fin de ejecucion!!");
 exit(-1);
 }
 strcpy(cd,b);
 return cd;
}

PERSONA* asigper()
{
 PERSONA* p;

 p = (PERSONA*)malloc(sizeof(PERSONA));

```

```

printf("\nNombre: "); p->nom = asigcad();
printf("Edad: "); scanf("%d%c",&p->edad);
printf("Direccion: "); p->dir = asigcad();
return p;
}

PROFESOR* asigprof()
{
 PROFESOR* t;
 t =(PROFESOR*)malloc(sizeof(PERSONA));
 t ->p = *asigper();
 printf("\nAños de experiencia: ");
 scanf("%d%c",&t->expe);
 return t;
}

ALUMNO* asigalms(short n)
{
 int j;
 ALUMNO* a;
 a = (ALUMNO*)calloc(n,sizeof(ALUMNO));
 if (a == NULL)
 {
 puts("\n\t!!Error de asignacion de memoria, fin de ejecucion!!");
 exit(-1);
 }
 /* Entrada de datos de cada alumno */
 for (j=0; j<n; j++)
 {
 (a+j)->p = *asigper();
 printf("Departamento al que pertenece: ");
 (a+j)->depar = asigcad();
 printf("Nivel en que se encuentra: ");
 scanf("%d%c",&(a+j)->nivel);
 }
 return a;
}

```

## 11.6. ARRAYS DINÁMICOS

Un nombre de un array es realmente un puntero constante que se asigna en tiempo de compilación:

```
float m[30]; /* m es un puntero constante a un bloque de 30 float*/
float" const p =(float*)malloc(30*sizeof(float));
```

*m* y *p* son punteros constantes a bloques de 30 números reales (*float*). La declaración de *m* se denomina *ligadura estática* debido a que se asigna en tiempo de compilación; el símbolo se enlaza a la memoria asignada aunque el array no se utiliza nunca durante la ejecución del programa.

Por el contrario, se puede utilizar un puntero no constante para posponer la asignación de memoria hasta que el programa se esté ejecutando. Este tipo de enlace o ligadura se denomina *ligadura dinámica* o *ligadura en tiempo de ejecución*

```
float" p = (float*)malloc(30*sizeof(float));
```

Un array que se declara de este modo se denomina *array dinámico*.

**Comparar los dos métodos de definición de un array**

- `float m[30];` /\* array estático \*/
- `float* p=(float*)malloc(30*sizeof(float));` /\* array dinámico\*/

El *array estático* `m` se crea en tiempo de compilación; su memoria permanece asignada durante toda la ejecución del programa. El *array dinámico* se crea en tiempo de ejecución; su memoria se asigna sólo cuando se ejecuta su declaración. No obstante, la memoria asignada al array `p` se libera tan pronto como se invoca a la función `free()`, de este modo

```
free(p);
```

**11.7. REGLAS DE FUNCIONAMIENTO DE LA ASIGNACIÓN DE MEMORIA**

Como ya se ha comentado se puede asignar espacio para cualquier objeto dato de C. Las reglas para utilizar las funciones `malloc()`, `calloc()`, `realloc()` y `free()` como medio para obtener/liberar espacio libre de memoria son las siguientes:

1. El prototipo de las funciones esta en `stdlib.h`.

```
#include <stdlib.h>
int* datos;
...
datos = (int*)malloc(sizeof(int));
```

2. Las funciones `malloc()`, `calloc()`, `realloc()` devuelven el tipo `void*`, lo cual exige hacer una conversión al tipo del puntero.

```
#include <stdlib.h>

void main()
{
 double* vec;
 int n;

 vec = (double*)calloc(n, sizeof(double));
}
```

3. Las funciones de asignación tienen como argumento el número de bytes a reservar.
4. El operador `sizeof` permite calcular el tamaño de un tipo de objeto para el que está asignando memoria.

```
struct punto
{
 float x,y,z;
};

struct punto*p = (struct punto*)malloc(sizeof(struct punto));
```

5. La función `realloc()` permite expandir memoria reservada.

```
#include <stdlib.h>

int *v=NULL;;
int n;
scanf("%d",&n);
v = (int*)realloc(v,n);
v = (int*)realloc(v,2*n);
```

6. Las funciones de asignación de memoria devuelven NULL si no han podido reservar la memoria requerida.

```
double" v;
v = malloc(1000*sizeof(double));
if (v == NULL)
{
 puts("Error de asignación de memoria.");
 exit(-1);
}
```

7. Se puede utilizar cualquier función de asignación de memoria para reservar espacio de objetos más complejos, tales como estructuras, arrays, en el almacenamiento libre.

```
#include <stdlib.h>
struct complejo

 float x,y;
};

void main()
{
 struct complejo* pz; /* Puntero a estructura complejo */
 int n;
 scanf("%d", &n);
 /* Asigna memoria para un array de tipo complejo */
 pz = (struct complejo *)calloc(n,sizeof(struct complejo));
 if (pz == NULL)
 {
 puts("Error de asignación de memoria.");
 exit(-1);
 }
}
```

8. Se pueden crear arrays multidimensionales de objetos con las funciones de asignación de memoria. Para un array bidimensional  $n \times m$ , se asigna en primer lugar memoria para un array de punteros (de  $n$  elementos), y después se asigna memoria para cada fila ( $m$  elementos) con un bucle desde 0 a  $n-1$ .

```
#include <stdlib.h>

double **mat;
int n,m,i;

mat = (double**)malloc(n*sizeof(double*)); /* array de punteros */
for (i=0; i<n; i++)
{
 mat[i]=(double*)malloc(m*sizeof(double)); /*fila de m elementos */
}
}
```

9. Toda memoria reservada con alguna de las funciones de asignación de memoria se puede liberar con la función `free()`. Para liberar la memoria de la matriz dinámica `mat`:

```
double **mat;

for (i=0; i<n; i++)
{
 free(mat[i]);
}
}
```

## 11.8. RESUMEN

La asignación dinámica de memoria permite utilizar tanta memoria como se necesite. Se puede asignar espacio a una variable en el almacenamiento libre cuando se necesite y se libera la memoria cuando se desee.

En C se utilizan las funciones `malloc()`, `calloc()`, `realloc()` y `free()` para asignar y liberar memoria. Las funciones `malloc()`, `calloc()`, `realloc()` permiten asignar memoria para cualquier tipo de dato especificado (un `int`, un `float`, una estructura, un array o cualquier otro tipo de dato).

Cuando se termina de utilizar un bloque de memoria, se puede liberar con la función `free()`. La memoria libre se devuelve al almacenamiento libre, de modo que quedará más memoria disponible para asignar otros bloques de memoria.

El siguiente ejemplo asigna un array y llama a la función `free()` que libera el espacio ocupado en memoria:

```
typedef struct animal
{
 . . .
}ANIMAL;
ANIMAL* pperro;
pperro =
(ANIMAL*)malloc(5*sizeof (ANIMAL));
if (pperro == NULL)
 puts("¡Falta memoria!");
else
{

 /* uso de pperro */

 free(pperro); /* libera espacio
 de pperro */
}
```

## 11.9. EJERCICIOS

- 11.1. Encuentre los errores en las siguientes declaraciones y sentencias.

```
int n, *p;
char** dob= "Cadena de dos
 punteros";
p = n*malloc(sizeof(int));
```

- 11.2. Dada la siguiente declaración, definir un puntero `b` a la estructura, reservar memoria dinámicamente para una estructura asignando su dirección a `b`.

```
struct boton
{
 char* rotulo;
 int codigo;
};
```

- 11.3. Una vez asignada memoria al puntero `b` del Ejercicio 11.2 escribir sentencias para leer los campos `rotulo` y `codigo`.

- 11.4. Un array unidimensional puede considerarse una constante puntero. ¿Cómo puede considerarse un array bidimensional?, ¿y un array de tres dimensiones?

- 11.5. Declara una estructura para representar un punto en el espacio tridimensional. Declara un puntero a la estructura para que tenga la dirección de un array dinámico de  $n$  estructuras punto. Utiliza la función `calloc()` para asignar memoria al array y comprueba que se ha podido asignar la memoria requerida.

- 11.6. ¿Qué diferencias existen entre las funciones `malloc()`, `calloc()` y `realloc()`?

- 11.7. Dada la declaración de la estructura `punto` (Ejercicio 11.5) escribe una función que devuelva la dirección de un array dinámico de  $n$  puntos en el espacio tridimensional. Los valores de los datos se leen del dispositivo de entrada (teclado).

11.8. Dada la declaración del array de punteros:

```
#define N 4
char * [N];
```

Escriba las sentencias de código para leer N líneas de caracteres y asignar cada línea a un elemento del array.

11.9. Escriba una función que reciba el array dinámico creado en el Ejercicio 11.7 y amplíe el array en otros *m* puntos del espacio.

11.10. Escriba una función que reciba las N líneas leídas en el Ejercicio 11.8 y libere las líneas de longitud menor de 20 caracteres. Las líneas restantes han de quedar en orden consecutivo, desde la posición cero.

11.11. ¿Qué diferencias existe entre las siguientes declaraciones?:

```
char *c[15];
char **c,
char c[15][12];
```

## 11.10. PROBLEMAS

*En todos los problemas, utilice siempre que sea posible punteros para acceder a los elementos de los arrays, tanto numéricos como cadenas de caracteres.*

11.1. Escriba un programa para leer *n* cadenas de caracteres. Cada cadena tiene una longitud variable y está formada por cualquier carácter. La memoria que ocupa cada cadena se ha de ajustar al tamaño que tiene. Una vez leídas las cadenas se debe de realizar un proceso que consiste en eliminar todos los blancos, siempre manteniendo el espacio ocupado ajustado al número de caracteres. El programa debe mostrar las cadenas leídas y las cadenas transformadas.

11.2. Se desea escribir un programa para leer números grandes (de tantos dígitos que no entran en variables `long`) y obtener la suma de ellos. El almacenamiento de un número grande se ha de hacer en una estructura que tenga un array dinámico y otro campo con el número de dígitos. La suma de dos números grandes dará como resultado otro número grande representado en su correspondiente estructura.

11.3. En una competición de ciclismo se presentan *n* ciclistas. Cada participante se representa por el nombre, club, los puntos obtenidos y prueba en que participará en la competición. La competi-

ción es por eliminación. Hay prueba de dos tipos, persecución y velocidad. En la de persecución participan tres ciclistas, el primero recibe 3 puntos y el tercero se elimina. En la de velocidad participan 4 ciclistas, el más rápido obtiene 4 puntos el segundo 1 y el cuarto se elimina. Las pruebas se van alternando, empezando por velocidad. Los ciclistas participantes en una prueba se eligen al azar entre los que en menos pruebas han participado. El juego termina cuando no quedan ciclistas para alguna de las dos pruebas. Se ha de mantener arrays dinámicos con los ciclistas participantes y los eliminados. El ciclista ganador será el que más puntos tenga.

11.4. Se tiene una matriz de 20x20 elementos enteros. En la matriz hay un elemento repetido muchas veces. Se quiere generar otra matriz de 20 filas y que en cada fila estén sólo los elementos no repetidos. Escribir un programa que tenga como entrada la matriz de 20x20, genere la matriz dinámica pedida y se muestre en pantalla.

11.5. Escriba un programa para generar una matriz simétrica con números aleatorios de 1 a 9. El usuario introduce el tamaño de cada dimensión de la matriz y el programa reserva memoria libre para el tamaño requerido.

# CAPÍTULO 12

## CADENAS

---

### CONTENIDO

- 12.1. Concepto de cadena.
- 12.2. Lectura de cadenas.
- 12.3. La biblioteca `string.h`.
- 12.4. Arrays y cadenas como parámetros de funciones.
- 12.5. Asignación de cadenas.
- 12.6. Longitud y concatenación de cadenas.
- 12.7. Comparación de cadenas.
- 12.8. Inversión de cadenas.
- 12.9. Conversión de cadenas.
- 12.10. Conversión de cadenas a números.
- 12.11. Búsqueda de caracteres y cadenas.
- 12.12. *Resumen.*
- 12.13. *Ejercicios.*
- 12.14. *Problemas.*

---

## INTRODUCCIÓN

El lenguaje C no tiene datos predefinidos tipo cadena (**string**). En su lugar C, manipula cadenas mediante arrays de caracteres que terminan con el carácter nulo ASCII (`'\0'`). Una *cadena* se considera como un array unidimensional de tipo `char` o `unsigned char`. En este capítulo se estudiarán temas tales como:

- cadenas en C;
- lectura y salida de cadenas;
- uso de funciones de cadena de la biblioteca estándar;
- asignación de cadenas;
- operaciones diversas de cadena (longitud, concatenación, comparación y conversión);
- localización de caracteres y subcadenas;
- inversión de los caracteres de una cadena.

---

## CONCEPTOS CLAVE

- Asignación.
- Biblioteca *string.h*.
- Cadena.
- Cadenavacía.
- Carácter nulo (`NULL`, `'\0'`).
- Comparación.
- Conversión.
- Funciones de cadena.
- Inversión.
- **String**.

## 12.1. CONCEPTO DE CADENA

Una *cadena* (también llamada *constante de cadena* o *literal de cadena*) es un tipo de dato compuesto, un array de caracteres (`char`), terminado por un carácter *nulo* (`'\0'`), NULL (Fig. 12.1). Un ejemplo es "ABC".

Cuando la cadena aparece dentro de un programa se verá como si se almacenarán cuatro elementos: 'A', 'B', 'C' y '\0'. En consecuencia, se considerará que la cadena "ABC" es un array de cuatro elementos de tipo `char`. El valor real de esta cadena es la dirección de su primer carácter y su tipo es un puntero a `char`. Aplicando el operador `*` a un puntero a `char` se obtiene el carácter que forma su contenido; es posible también utilizar aritmética de direcciones con cadenas:

|                           |                   |                   |
|---------------------------|-------------------|-------------------|
| <code>*"ABC"</code>       | <i>es igual a</i> | <code>'A'</code>  |
| <code>*("ABC" + 1)</code> | <i>es igual a</i> | <code>'B'</code>  |
| <code>*("ABC" + 2)</code> | <i>es igual a</i> | <code>'C'</code>  |
| <code>*("ABC" + 3)</code> | <i>es igual a</i> | <code>'\0'</code> |

De igual forma, utilizando el subíndice del array se puede escribir:

|                       |                   |                   |
|-----------------------|-------------------|-------------------|
| <code>"ABC"[0]</code> | <i>es igual a</i> | <code>'A'</code>  |
| <code>"ABC"[1]</code> | <i>es igual a</i> | <code>'B'</code>  |
| <code>"ABC"[2]</code> | <i>es igual a</i> | <code>'C'</code>  |
| <code>"ABC"[3]</code> | <i>es igual a</i> | <code>'\0'</code> |

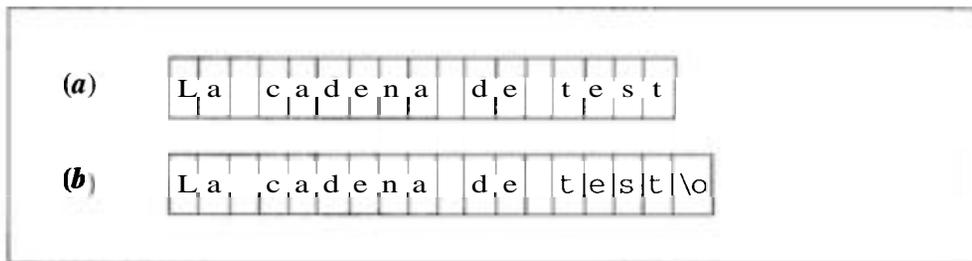


Figura 12.1. (a) array de caracteres; (b) cadena de caracteres.

**El número total de caracteres de una cadena en C es siempre igual a la longitud de la cadena más 1.**

### Ejemplos

- `char cad[] = "Lupiana";`  
cad tiene ocho caracteres; 'L', 'u', 'p', 'i', 'a', 'n', 'a' y '\0'
- `printf("%s", cad);`  
el sistema copiará caracteres de cad a stdout (pantalla) hasta que el carácter NULL, '\0', se encuentre.
- `scanf("%s", cad);`  
el sistema copiará caracteres desde stdin (teclado) a cad hasta que se encuentre un carácter espacio en blanco o fin de línea. El usuario ha de asegurarse que el buffer cad esté definido como una cadena de caracteres lo suficiente grande para contener la entrada.

**Las funciones declaradas en el archivo de cabecera <string.h> se utilizan para manipular cadenas.**

### 12.1.1. Declaración de variables de cadena

Las cadenas se declaran como los restantes tipos de arrays. El operador postfijo [] contiene el tamaño máximo del objeto. El tipo base, naturalmente, es char, o bien unsigned char:

```
char texto[81]; /* una línea de caracteres de texto */
char orden[40]; /* cadena utilizada para recibir una orden del
 teclado */
unsigned char datos; /* puede contener cualquier carácter ASCII */
```

El tipo unsigned char puede ser de interés en aquellos casos en que los caracteres especiales presentes puedan tener el bit de orden alto activado. Si el carácter se considera con signo, el bit de mayor peso (orden alto) se interpreta como *hit de signo* y se puede propagar a la posición de mayor orden (peso) del nuevo tipo.

Observe que el tamaño de la cadena ha de incluir el carácter '\0'. En consecuencia, para definir un array de caracteres que contenga la cadena "ABCDEF", escriba

```
char UnaCadena[7];
```

**A veces se puede encontrar una declaración como ésta:**

```
char *s;
```

**¿Es s realmente una cadena? No, no lo es. Es un puntero a un carácter (el primer carácter de una cadena), que todavía no tiene memoria asignada,**

### 12.1.2. Inicialización de variables de cadena

Todos los tipos de arrays requieren una *inicialización* (iniciación) que consiste en una lista de valores separados por comas y encerrados entre llaves.

```
char texto[81] = "Esto es una cadend.";
char textodemo[255] = "Esta es una cadena muy larga";
char cadenatest[] = "¿Cuál es la longitud de esta cadena?";
```

Las cadenas texto y textodemo pueden contener 80 y 254 caracteres respectivamente más el carácter nulo. La tercera cadena, cadenatest, se declara con una especificación de tipo incompleta y se completa sólo con el inicializador. Dado que en el literal hay 36 caracteres y el compilador añade el carácter '\0', un total de 37 caracteres se asignarán a cadenatest.

Ahora bien, una cadena no se puede inicializar fuera de la declaración. Por ejemplo, si trata de hacer

```
UnaCadena = "ABC";
```

C le dará un error al compilar. La razón es que un identificador de cadena, como cualquier identificador de array se trata como un valor de dirección, como un puntero constante. *¿Cómo se puede inicializar una cadena fuera de la declaración?* Más adelante se verá, pero podemos indicar que será necesario utilizar una función de cadena denominada strcpy().

### Ejemplo 12.1

Las cadenas terminan con el carácter nulo. Así en el siguiente programa se muestra que el carácter `NULL` (`'\0'`) se añade a la cadena:

```
#include <stdio.h>
int main()
{
 char S[] = "ABCD";
 for (int i = 0; i < 5; i++)
 printf("S[%d] = %c\n",i,S[i]);
 return 0;
}
```

Ejecución

```
S[0] = A
S[1] = B
S[2] = C
S[3] = D
S[4] =
```

Comentario: Cuando el carácter `NULL` se mandó imprimir, no escribe nada.

## 12.2. LECTURA DE CADENAS

La lectura usual de datos se realiza con la función `scanf()`, cuando se aplica a datos cadena el código de formato es `%s`. La función da por terminada la cadena cuando encuentra un espacio (un blanco) o fin de línea. Esto puede producir anomalías al no poder captar cadenas con blancos entre caracteres. Así, por ejemplo, trate de ejecutar el siguiente programa:

```
/* Este programa muestra cómo scanf() lee datos cadena */

#include <stdio.h>

void main()
{
 char nombre[30]; /* Define array de caracteres */
 scanf("%s", nombre); /* Leer la cadena */
 printf("%s \n", nombre); /* Escribir la cadena nombre */
}
```

El programa define `nombre` como un array de caracteres de **30** elementos. Suponga que introduce la entrada `Pepe Margolles`, cuando ejecuta el programa se visualizará en pantalla `Pepe`. Es decir, la palabra `Margolles` no se ha asignado a la variable cadena `nombre`. La razón es que la función `scanf()` termina la operación de lectura siempre que se encuentra un espacio en blanco o fin de línea.

Así pues, ¿cuál será la mejor forma para lectura de cadenas, cuando estas cadenas contienen más de una palabra (caso muy usual)? El método recomendado será utilizar una función denominada `gets()`. La función `gets()` permitirá leer la cadena completa, incluyendo cualquier espacio en blanco, termina al leer el carácter de fin de línea.

El prototipo de la función está en el archivo `stdio.h`. La función asigna la cadena al argumento transmitido a la función, que será un array de caracteres o un puntero (`char*`) a memoria libre, con un número de elementos suficiente para guardar la cadena leída. Si ha habido un error en la lectura de la cadena, devuelve `NULL`.

```
/* Lectura de caracteres hasta fin de línea */
```

```
char b[81];
gets(b);
```

## Ejemplo 12.2

*Entrada y salida de cadenas. Lectura de palabras de 79 caracteres de máxima longitud en una memoria intermedia (buffer) de 80 caracteres.*

```
#include <stdio.h>
void main()
{
 char palabra[80];
 do {
 scanf("%s", palabra);
 if (!feof(stdin))
 printf("\t\"%s\" \n", palabra);
 } while (!feof(stdin));
}
```

Al ejecutar este programa el número de veces que se repite el bucle `while` dependerá del número de palabras introducidas, incluido el carácter de control que termina el bucle **control-Z**.

## Ejecución

Hoy es 1 de Enero del 2000.

```
"Hoy"
"es"
"1"
"de"
"Enero"
"del"
"2000."
```

Mañana es Domingo.

```
"Mañana"
"es"
"Domingo."
```

El bucle anterior se ejecuta `ll` veces, una vez por cada palabra introducida (incluyendo **Control-Z** que detiene el bucle). Cada palabra de la entrada (`stdin`) hace eco en la salida (`stdout`). El flujo de salida no «se limpia» hasta que el flujo de entrada encuentra el final de la línea.

Cada cadena se imprime encerrada entre comillas. No será fin de archivo (`feof()` distinto de cero) mientras que no se pulse **Control-Z** (en Windows/DOS), que envía el carácter final de archivo del flujo estándar de entrada `stdin`.

## Advertencia

Los signos de puntuación, apóstrofes, comas, puntos, etc., se incluyen en las cadenas, pero no así los caracteres espacios en blanco (blancos, tabulaciones, nuevas líneas, etc.).

**Ejemplo 12.3**

El siguiente programa solicita introducir un nombre, comprueba la operación y lo escribe en pantalla.

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

 char nombre[80];
 printf("\nIntroduzca su nombre: ");
 if (gets(nombre)!= NULL)
 printf("Hola %s ¿cómo está usted?",nombre);
 return 0;
}
```

Si al ejecutarlo se introduce la cadena `Mara Martina`, el array `nombre` almacenará los caracteres siguientes:

```
nombre
M a r a M a r t i n a '\0'
[0][1][2][3][4][5][6][7][8][9][10][11][12]
```

**Ejemplo 11.4**

El siguiente programa lee y escribe el nombre, dirección y teléfono de un usuario.

```
#include <stdio.h>
void main()
{
 char Nombre[32];
 char Calle[32];
 char Ciudad[27];
 char Provincia[27];
 charCodigoPostal[5];
 char Telefono[10];

 printf "\nNombre: "; gets (Nombre);
 printf "\nCalle: "; gets (Calle);
 printf "\nCiudad: "; gets (Ciudad);
 printf "\nProvincia: "; gets (Provincia);
 printf "\nCodigo Postal: "; gets(CodigoPostal);
 printf "\nTelefono: "; gets(Telefono);

 /* vis alizar cadenas */

 printf("\n\n%s \t %s\n",Nombre,Calle);
 printf("%s \t %s\n",Ciudad,Provincia);
 printf("%s \t %s\n",CodigoPostal,Telefono);
}
```

**Regla**

- La llamada `gets(cad)` lee todos los caracteres hasta encontrar el carácter fin de línea, `'\n'`, que en la cadena `cad` se sustituye por `'\0'`.

### 12.2.1. Función `getchar()`

La función `getchar()` se utiliza para leer carácter a carácter. La llamada a `getchar()` devuelve el carácter siguiente del flujo de entrada `stdin`. En caso de error, o de encontrar el fin de archivo, devuelve EOF (macro definida en `stdio.h`).

---

#### Ejemplo 12.5

*El siguiente programa cuenta las ocurrencias de la letra 't' del flujo de entrada. Se diseña un bucle while que continúa ejecutándose mientras que la función `getchar()` lee caracteres y se asignan a `car`.*

```
#include <stdio.h>
int main()
{
 int car;
 int cuenta = 0;
 while ((car = getchar()) != EOF)
 if (car == 't') ++cuenta;
 printf("\n %d letras t \n", cuenta);
 return 0;
}
```

#### Nota

La salida del bucle es con Control-Z.

### 12.2.2. Función `putchar()`

La función opuesta de `getchar()` es `putchar()`. La función `putchar()` se utiliza para escribir en la salida (`stdout`) carácter a carácter. El carácter que se escribe es el transmitido como argumento. Esta función (realmente es una macro definida en `stdio.h`) tiene como prototipo:

```
int putchar(int ch);
```

---

#### Ejercicio 12.1

*El siguiente programa hace «eco» del flujo de entrada y convierte las palabras en palabras iguales que comienzan con letra mayúscula. Es decir, si la entrada es "poblado de peñas rubias" se ha de convertir en "Poblado De Peñas Rubias". Para realizar esa operación se recurre a la función `toupper(car)` que devuelve el equivalente mayúscula de `car` si `car` es una letra minúscula. El archivo de cabecera necesario para poder utilizar la función `toupper(car)` es `<ctype.h>`.*

```
#include <stdio.h>
#include <ctype.h>
int main()
{
 char car, pre = '\n';
 while ((car=getchar()) != EOF)
 {
 if (pre == ' ' || pre == '\n')
 putchar(toupper(car));
 else
```

```

 putchar (car);
 pre = car;
}
return 0;
}

```

## Ejecución

poblado de peñas rubias con capital en Lupiana  
Poblado De Peñas Rubias Con Capital En Lupiana

### Análisis

La variable `pre` contiene el carácter leído anteriormente. El algoritmo se basa en el hecho de que si `pre` es un blanco o el carácter nueva línea, entonces el carácter siguiente `car` será el primer carácter de la siguiente palabra. En consecuencia, `car`, se reemplaza por su carácter mayúscula equivalente: `car + 'A' - 'a'`.

### 12.2.3. Función `puts()`

La función `puts()` escribe en la salida una cadena de caracteres, incluyendo el carácter fin de línea por lo que sitúa el puntero de salida en la siguiente línea. Es la función recíproca de `gets()`; si `gets()` capta una cadena hasta fin de línea, `puts()` escribe una cadena y el fin de línea. El prototipo de la función se encuentra en `stdio`:

```
int puts(const char *s);
```

---

### Ejercicio 12.2

*El programa siguiente lee una frase y escribe en pantalla tantas líneas como palabras tiene la frase; cada línea que escribe, a partir de la primera, sin la última palabra de la línea anterior:*

### Análisis

La función `sgtepal()` explora los caracteres pasados en `p` hasta que encuentra el primer blanco (separador de palabras). La exploración se realiza de derecha a izquierda, en la posición del blanco asigna `'\0'` para indicar fin de cadena.

```

#include <stdio.h>
#include <string.h>
void sgtepal(char* p);

void main()
{
 char linea[81];
 printf("\n\tIntroduce una línea de caracteres.\n");
 gets(linea);
 while (*linea)
 {
 puts(linea);
 sgtepal(linea);
 }
}

void sgtepal(char* p)
{

```

```

int j;
j = strlen(p)-1;
while(j>0 && p[j]!=' ')
 j--;
p[j] = "\\0";
}

```

## Ejecución

```

Introduce una línea de caracteres.
Erase una vez la Mancha
Erase una vez la Mancha
Erase una vez la
Erase una vez
Erase una
Erase

```

### 12.2.4. Funciones `getch()` y `getche()`

Estas dos funciones no pertenecen a ANSI C, sin embargo, se incorporan por estar en casi todos los compiladores de C. Ambas funciones leen un carácter tecleado sin esperar el retorno de carro. La diferencia entre ellas reside en que con `getch()` el carácter tecleado no se visualiza en pantalla (no hace eco en la pantalla), y con `getche()` si hay eco en la pantalla. La llamada a cada una de ellas:

```

car = getch();
car = getche();

```

El prototipo de ambas funciones se encuentra en el archivo `conio.h`

```

int getch(void);
int getche(void);

```

---

### Ejemplo 12.6

*La siguiente función devuelve el carácter S o N :*

```

#include <conio.h>
#include <ctype.h>

int respuesta()
{
 char car;
 do
 car = toupper(getche());
 while (car != 'S' && car != 'N');
 return car;
}

```

---

## 12.3. LA BIBLIOTECA `STRING.H`

La biblioteca estándar de C contiene la biblioteca de cadena `STRING.H`, que incorpora las funciones de manipulación de cadenas utilizadas más frecuentemente. El archivo de cabecera `STDIO.H` también

soporta E/S de cadenas. Algunos fabricantes de C también incorporan otras bibliotecas para manipular cadenas, pero como *no son estándar* no se considerarán en esta sección. Las funciones de cadena tienen argumentos declarados de forma similar a:

```
char *s1; o bien, const char *s1;
```

Esto significa que la función espera una cadena que puede o no modificarse. Cuando se utiliza la función, se puede usar un puntero a char o se puede especificar el nombre de una variable array char. Cuando se pasa un array a una función, C pasa automáticamente la dirección del array char. La Tabla 12.1 resume algunas de las funciones de cadena más usuales.

Tabla 12.1. Funciones de <string.h>.

| Función           | Cabecera de la función y prototipo                                                                                                                                                                                                                                                                                                                               |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>memcpy()</b>   | void* memcpy(void* s1, const void* s2, size_t n);<br>Reemplaza los primeros <i>n</i> bytes de *s1 con los primeros <i>n</i> bytes de *s2. Devuelve s1.                                                                                                                                                                                                           |
| <b>strcat</b>     | char *strcat(char *destino, const char *fuente);<br>Añade la cadena fuente al final de destino. <i>concatena</i> . Devuelve la cadena destino.                                                                                                                                                                                                                   |
| <b>strchr()</b>   | char* strchr(char* s1, int ch);<br>Devuelve un puntero a la primera ocurrencia de ch en s1. Devuelve NULL si ch no está en s1.                                                                                                                                                                                                                                   |
| <b>strcmp()</b>   | int strcmp(const char *s1, const char *s2);<br>Compara alfabéticamente la cadena s1 a s2 y devuelve:<br>0 si s1 = s2<br><0 si s1 < s2<br>>0 si s1 > s2                                                                                                                                                                                                           |
| <b>strncmpi()</b> | int strncmpi(const char *s1, const char *s2);<br>Igual que strcmp(), pero sin distinguir entre mayúsculas y minúsculas.                                                                                                                                                                                                                                          |
| <b>strcpy()</b>   | char *strcpy(char *destino, const char *fuente);<br>Copia la cadena fuente a la cadena destino. Devuelve la cadena destino.                                                                                                                                                                                                                                      |
| <b>strcspn()</b>  | size_t strcspn(const char* s1, const char* s2);<br>Devuelve la longitud de la subcadena más larga de s1 que comienza con el carácter s1[0] y no contiene ninguno de los caracteres de la cadena s2.                                                                                                                                                              |
| <b>strlen()</b>   | size_t strlen(const char *s)<br>Devuelve la longitud de la cadena s.                                                                                                                                                                                                                                                                                             |
| <b>strncat()</b>  | char* strncat(char* s1, const char*s2, size_t n);<br>Añade los primeros n caracteres de s2 a s1. Devuelve s1. Si n >= strlen(s2), entonces strncat(s1, s2, n) tiene el mismo efecto que strcat(s1, s2).                                                                                                                                                          |
| <b>strncmp()</b>  | int strncmp(const char* s1, const char* s2, size_t n);<br>Compara s1 con la subcadena formada por los primeros n caracteres de s2. Devuelve un entero negativo, cero o un entero positivo, según que s1 lexicográficamente sea menor, igual o mayor que la subcadena s2. Si n ≥ strlen(s2), entonces strncmp(s1, s2, n) y strcmp(s1, s2) tienen el mismo efecto. |
| <b>strnset()</b>  | char *strnset(char *s, int ch, size_t n);<br>Copia n veces el carácter ch en la cadena s a partir de la posición inicial de s (s[0]). El máximo de caracteres que copia es la longitud de s.                                                                                                                                                                     |
| <b>strpbrk()</b>  | char* strpbrk(const char* s1, const char* s2);<br>Devuelve la dirección de la primera ocurrencia en s1 de cualquiera de los caracteres de s2. Devuelve NULL si ninguno de los caracteres de s2 aparece en s1.                                                                                                                                                    |

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strrchr()</b> | <code>char* strrchr(const char* s, int c);</code><br>Devuelve un puntero a la última ocurrencia de <code>c</code> en <code>s</code> . Devuelve <code>NULL</code> si <code>c</code> no está en <code>s</code> . La búsqueda la hace en sentido inverso, desde el final de la cadena al primer carácter, hasta que encuentra el carácter <code>c</code> .                                                                                                                                                                                                                                                                                      |
| <b>strspn()</b>  | <code>size_t strspn(const char* s1, const char* s2);</code><br>Devuelve la longitud de la subcadena izquierda ( <code>s1[0]...</code> ) más larga de <code>s1</code> que contiene únicamente caracteres de la cadena <code>s2</code> .                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>strstr()</b>  | <code>char* strstr(const char* s1, const char* s2);</code><br>Busca la cadena <code>s2</code> en <code>s1</code> y devuelve un puntero a los caracteres donde se encuentra <code>s2</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>strtok()</b>  | <code>char* strtok(char* s1, const char* s2);</code><br>Analiza la cadena <code>s1</code> en <i>tokens</i> (componentes léxicos), éstos delimitados por caracteres de la cadena <code>s2</code> . La llamada inicial a <code>strtok(s1, s2)</code> devuelve la dirección del primer <i>token</i> y sitúa <code>NULL</code> al final del <i>token</i> . Después de la llamada inicial, cada llamada sucesiva a <code>strtok(NULL, s2)</code> devuelve un puntero al siguiente <i>token</i> encontrado en <code>s1</code> . Estas llamadas cambian la cadena <code>s1</code> , reemplazando cada separador con el carácter <code>NULL</code> . |

---

### 12.3.1. La palabra reservada `const`

Las funciones de cadena declaradas en `<string.h>`, recogidas en la Tabla 12.1 y algunas otras, incluyen la palabra reservada `const`. La ventaja de esta palabra reservada es que se puede ver rápidamente la diferencia entre los parámetros de entrada y salida. Por ejemplo, el segundo parámetro *fuelle* de `strcpy` representa el área fuente; se utiliza sólo para copiar caracteres de ella, de modo que este área no se modificará. La palabra reservada `const` se utiliza para esta tarea. Se considera un parámetro de *entrada*, ya que la función *recibe* datos a través de ella. En contraste, el primer parámetro *destino* de `strcpy` es el área de destino, la cual se sobrescribirá y, por consiguiente, no se debe utilizar `const` para ello. En este caso, el parámetro correspondiente se denomina *parámetro de salida*, ya que los datos se escriben en el área de destino.

## 12.4. ARRAYS Y CADENAS COMO PARÁMETROS DE FUNCIONES

En los arrays y cadenas siempre se pasa la dirección del objeto, un puntero al primer elemento del array. En la función, las referencias a los elementos individuales se hacen por indirección de la dirección del objeto. Considérese el programa `PASARRAY.C`, que implementa una función `Longitud()` que calcula la longitud de una cadena terminada en nulo. El parámetro `cad` se declara como un array de caracteres de tamaño desconocido.

```

/* PASARRAY.C */
#include <stdio.h>
#include <conio.h>
int longitud(char cad[]);

void main(void)
{
 char* cd = "Cualquier momento es bueno para la felicidad";
 printf("\nLongitud de la cadena \"%s\": %d\n", cd, longitud(cd));
 puts("Pulse cualquier tecla para continuar ");
 getch();
}

int longitud(char cad[])
{
 int cuenta = 0;

```

```

 while (cad[cuenta++] != '\0');
 return cuenta;
}

```

En la función `main()` se reserva memoria para la constante cadena `cd`, a la función `longitud()` se transmite la dirección de la cadena. El cuerpo del bucle `while` dentro de la función `cuenta` los caracteres no nulos y termina cuando se encuentra el byte nulo al final de la cadena.

### Ejercicio 12.3

*El programa siguiente extrae n caracteres de una cadena introducida por el usuario.*

#### Análisis

La extracción de caracteres se realiza en una función que tiene como primer argumento la subcadena a extraer, como segundo argumento la cadena fuente y el tercero el número de caracteres a extraer. Se utilizan los punteros para pasar arrays a la función.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
int extraer(char *dest, const char *fuente, int num_cars);
void main (void)
{
 char s1[81];
 char* s2;
 int n;
 printf("\n\tCadena a analizar ?:");
 gets(s1);
 do {
 printf("Numero de caracteres a extraer: ");
 scanf("%d",&n);
 }while(n<1 || n>strlen(s1));
 s2 = malloc((n+1)*sizeof(char));
 extraer(s2,s1,n);
 printf("Cadena extraida \"%s\"",s2);
 puts("\nPulse intro para continuar");
 getch();
}
1
int extraer(char *dest, const char *fuente, int num_cars)
{
 int cuenta;
 for(cuenta = 1; cuenta <= num_cars; cuenta++)
 *dest++ = *fuente++;
 *dest = '\0';
 return cuenta; /* devuelve número de caracteres */
}

```

Observe que en las declaraciones de parámetros, ninguno está definido como array, sino como punteros de tipo `char`. En la línea

```
*dest++ = *fuente++;
```

los punteros se utilizan para acceder a las cadenas `fuente` y `destino`, respectivamente. En la llamada a la función `extraer()` se pasa la dirección de las cadenas `fuente` y `destino`.

## 12.5. ASIGNACIÓN DE CADENAS

C soporta dos métodos para asignar cadenas. Uno de ellos ya se ha visto anteriormente cuando se inicializaban las variables de cadena. La sintaxis utilizada:

```
char VarCadena [LongCadena] = ConstanteCadena;
```

---

### Ejemplo 12.7

*Inicializa dos arrays de caracteres con cadenas constantes.*

```
char Cadena[81] = "C maneja eficientemente las cadenas";
char nombre[] = "Luis Martin Cebo";
```

El segundo método para asignación de una cadena a otra es utilizar la función `strcpy()`. La función `strcpy()` copia los caracteres de la cadena fuente a la cadena destino. La función supone que la cadena destino tiene espacio suficiente para contener toda la cadena fuente. El prototipo de la función:

```
char* strcpy(char* destino, const char* fuente);
```

---

### Ejemplo 12.8

*Una vez definido un array de caracteres, se le asigna una cadena constante.*

```
char nombre[41];
strcpy(nombre, "Cadena a copiar");
```

La función `strcpy()` copia "Cadena a copiar" en la cadena nombre y añade un carácter nulo al final de la cadena resultante. El siguiente programa muestra una aplicación de `strcpy()`.

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

void main (void)
{
 char s[100] = "Buenos días Mr. Palacios", t[100];
 strcpy(t, s);
 strcpy(t+12, "Mr. C");
 printf("\n%s\n%s", s, t);
}
```

Al ejecutarse el programa produce la salida:

```
Buenos días Mr. Palacios
Buenos días Mr. C
```

La expresión `t+12` obtiene la dirección de la cadena `t` en `Mr. Palacios`. En esa dirección copia `Mr. C` y añade el carácter nulo (`'\0'`).

### 12.5.1. La función `strncpy()`

El prototipo de la función `strncpy` es

```
char* strncpy(char* destino, const char* fuente, size_t num);
```

y su propósito es copiar `num` caracteres de la cadena fuente a la cadena destino. La función realiza truncamiento o relleno de caracteres si es necesario.

**Ejemplo 12.9**

*Estas sentencias copia 4 caracteres de una cadena en otra.*

```
char cad1[] = "Pascal";
char cad2[] = "Hola mundo";
strncpy(cad1, cad2, 4);
```

La variable `cad1` contiene ahora la cadena "Hola".

**Consejo**

**Los punteros pueden manipular las partes posteriores de una cadena, asignando la dirección del primer carácter a manipular,**

```
char cad1[41] = "Hola mundo";
char cad2[41];
char* p = cad1;

p += 5; /* p apunta a la cadena "mundo" */
strcpy(cad2, p);
puts(cad2);
```

**La sentencia de salida visualiza la cadena "mundo".**

**12.6. LONGITUD Y CONCATENACIÓN DE CADENAS**

Muchas operaciones de cadena requieren conocer el número de caracteres de una cadena (*longitud*), así como la unión (*concatenación*) de cadenas.

**12.6.1. La función `strlen()`**

La función `strlen()` calcula el número de caracteres del parámetro `cadena`, excluyendo el carácter nulo de terminación de la cadena. El prototipo de la función es

```
size_t strlen(const char* cadena)
```

El tipo de resultado `size_t` representa un tipo entero general.

```
char cad[] = "1234567890";
unsigned i;
i = strlen(cad);
```

Estas sentencias asignan 10 a la variable `i`.

**Ejemplo**

*Este programa muestra por pantalla la longitud de varias cadenas.*

```
#include <string.h>
#include <stdio.h>
void main(void)
{
 char s[] = "IJKLMN";
 char bufer[81];
```

```

printf("strlen(%s) = %d\n",s,strlen(s));
printf("strlen(\"\") = %d\n",strlen(""));
printf("Introduzca una cadend: ");
gets(bufer);
printf("strlen(%s) = %d",bufer,strlen(bufer));
}

```

### Ejecución

```

strlen(IJKLMN) = 6
strlen("") = 0
Introduzca una cadena: Sierra de Horsche
strlen(Sierra de Horsche) = 16

```

## 12.6.2. Las funciones `strcat()` y `strncat()`

En muchas ocasiones se necesita construir una cadena, añadiendo una cadena a otra cadena, operación que se conoce como *concatenación*. Las funciones `strcat()` y `strncat()` realizan operaciones de concatenación. `strcat()` añade el contenido de la cadena fuente a la cadena destino, devolviendo un puntero a la cadena destino. Su prototipo es:

```
char* strcat(char" destino, const char* fuente);
```

---

### Ejemplo 12.10

*Copia una constante cadena y a continuación concatena con otra cadena.*

```

char cadena[81];
strcpy(cadena, "Borland");
strcat(cadena, "C");

```

La variable `cadena` contiene ahora "Borland C".

---

Es posible limitar el número de caracteres a concatenar utilizando la función `strncat()`. La función `strncat()` añade *num* caracteres de la cadena fuente a la cadena destino y devuelve el puntero a la cadena destino. Su prototipo es

```
char* strncat(char* destino, const char* fuente, size_t num)
```

y cuando se invoca con una llamada tal como

```
strncat(t, s, n);
```

*n* representa los primeros *n* caracteres de *s* que se van a unir a *t*, a menos que se encuentre un carácter nulo, en cuyo momento se termina el proceso.

---

### Ejemplo 12.11

*Concatenar 4 caracteres.*

```

char cad1[81] = "Hola soy yo ";
char cad2[41] = "Luis Merino";
strncat(cad1, cad2, 4);

```

La variable `cad1` contiene ahora "Hola soy yo Luis".

Ni la función `strcat()`, ni `strncat()` comprueba que la cadena destino tenga suficiente espacio para la cadena resultante. Por ejemplo:

```
char s1[] = "ABCDEFGH"; /* reserva espacio para 8+1 caracteres */
char s2[] = "XYZ"; /* reserva espacio para 3+1 caracteres */
strcat(s1,s2); /* produce resultado extraño por no haber espacio
 para la concatenación s1 con s2 */
```

---

### Ejercicio 12.4

El programa añade la cadena `s2` al final de la cadena `s1`. Reserva memoria dinámicamente, en tiempo de ejecución.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
void main (void)
{
 char* s1 = "ABCDEFGH";
 char s2[] = "XYZ";

 printf("\nAntes de strcat(s1,s2): \n");
 printf("\ts1 = [%s], longitud = %d\n",s1,strlen(s1));
 printf("\ts2 = [%s], longitud = %d\n",s2,strlen(s2));
 /* amplia memoria para la cadena resultante de la concatenación */
 s1 = realloc(s1,(strlen(s1)+strlen(s2)+1)*sizeof(char));
 printf("\ts1 = [%s], longitud = %d (amplia memoria)\n",
 s1,strlen(s1));
 strcat(s1,s2);
 puts ("Despues de strcat(s1,s2)");
 printf("\ts1 = [%s], longitud = %d\n",s1,strlen (s1));
 printf("\ts2 = [%s], longitud = %d\n",s2,strlen (s2));
}
```

### Ejecución

```
Antes de strcat(s1,s2):
 s1 = [ABCDEFGH], longitud = 8
 s2 = [XYZ], longitud = 3
 s1 = [ABCDEFGH], longitud = 8 (amplia memoria)
Despues de strcat(s1,s2)
 s1 = [ABCDEFGHXYZ], longitud = 11
 s2 = [XYZ], longitud = 3
```

## 12.7. COMPARACIÓN DE CADENAS

Dado que las cadenas son arrays de caracteres, la biblioteca `STRING.H` proporciona un conjunto de funciones que comparan cadenas. Estas funciones comparan los caracteres de dos cadenas utilizando el valor `ASCII` de cada carácter. Las funciones son `strcmp()`, `stricmp()`, `strncmp()` y `strnicmp()`:

### 12.7.1. La función strcmp()

Si se desea determinar si una cadena es igual a otra, mayor o menor que otra, se debe utilizar la función `strcmp()`. La comparación siempre es alfabética. `strcmp()` compara su primer parámetro con su segundo, y devuelve 0 si las dos cadenas son idénticas; un valor menor que cero si la cadena 1 es menor que la cadena 2; o un valor mayor que cero si la cadena 1 es mayor que la cadena 2 (los términos «*mayor que*» y «*menor que*» se refieren a la ordenación alfabética de las cadenas). Por ejemplo, Alicante es menor que Sevilla. Así, la letra A es menor que la letra a, la letra Z es menor que la letra a. El prototipo de la función `strcmp()` es

```
int strcmp(const char* cad1, const char* cad2);
```

La función compara las cadenas `cad1` y `cad2`. El resultado entero es:

|     |    |      |              |      |
|-----|----|------|--------------|------|
| < 0 | si | cad1 | es menor que | cad2 |
| = 0 | si | cad1 | es igual a   | cad2 |
| > 0 | si | cad1 | es mayor que | cad2 |

---

#### Ejemplo 12.12

Resultados de realizar comparaciones de cadenas.

```
char cad1[] = "Microsoft C";
char cad2[] = "Microsoft Visual C"
int i;

i = strcmp(cad1, cad2); /* i, toma un valor negativo */

strcmp("Waterloo", "Windows") < 0 {Devuelve un valor negativo}
strcmp("Mortimer", "Mortim") > 0 {Devuelve un valor positivo}
strcmp("Jertru", "Jertru") = 0 {Devuelve cero}
```

La comparación se realiza examinando los primeros caracteres de `cad1` y `cad2`; a continuación los siguientes caracteres y así sucesivamente. Este proceso termina cuando:

- se encuentran dos caracteres distintos del mismo orden: `cad1[i]` y `cad2[i]`;
- se encuentra el carácter nulo en `cad[i]` o `cad2[i]`

|          |              |                      |
|----------|--------------|----------------------|
| Waterloo | es menor que | Windows              |
| Mortimer | es mayor que | Mortim_carácter nulo |
| Jertru   | es igual que | Jertru               |

---

### 12.7.2. La función stricmp()

La función `stricmp()` compara las cadenas `cad1` y `cad2` sin hacer distinción entre mayúsculas y minúsculas. El prototipo es

```
int stricmp(const char* cad1, const char* cad2);
```

---

#### Ejemplo 12.13

Comparación de dos cadenas. con independencia de que sean letras mayúsculas o minúsculas.

```
char cad1[] = "Turbo C";
char cad2[] = "TURBO C";
int i;

i = stricmp(cad1, cad2);
```

Asigna 0 a la variable *i* ya que al no distinguir entre mayúsculas y minúsculas las dos cadenas son iguales.

---

### 12.7.3. La función `strncmp()`

La función `strncmp()` compara los `num` caracteres mas a la izquierda de las dos cadenas `cad1` y `cad2`. El prototipo es

```
int strncmp(const char* cad1, const char* cad2, size_t num);
```

y el resultado de la comparación será (considerando los `num` primeros caracteres):

|     |    |      |              |      |
|-----|----|------|--------------|------|
| < 0 | si | cad1 | es menor que | cad2 |
| = 0 | si | cad1 | es igual que | cad2 |
| > 0 | si | cad1 | es mayor que | cad2 |

---

#### Ejemplo 12.14

Comparar los 7 primeros caracteres de dos cadenas.

```
char cadena1[] = "Turbo C";
char cadena2[] = "Turbo Prolog"
int i;

i = strncmp(cadena1, cadena2, 7);
```

Esta sentencia asigna un número negativo a la variable *i*, ya que "Turbo C" es menor que "Turbo P". En el caso de comparar los 5 primeros caracteres:

```
i = strncmp(cadena1, cadena2, 5);
```

esta sentencia asigna un cero a la variable *i*, ya que "Turbo" es igual que "Turbo".

---

### 12.7.4. La función `strnicmp()`

La función `strnicmp()` compara los caracteres `num` a la izquierda en las dos cadenas, `cad1` y `cad2`, sin distinguir entre mayúsculas y minúsculas. El prototipo es

```
int strnicmp(const char* cad1, const char* cad2, size_t num);
```

El resultado será (considerando `num` primeros caracteres):

|     |    |      |              |      |
|-----|----|------|--------------|------|
| < 0 | si | cad1 | es menor que | cad2 |
| = 0 | si | cad1 | es igual que | cad2 |
| > 0 | si | cad1 | es mayor que | cad2 |

---

#### Ejemplo 12.15

Comparación de los 5 primeros caracteres. sin distinción entre mayúsculas y minúsculas

```
char cadena1[] = "Turbo C";
char cadena2[] = "TURBO C";
int i;
i = strnicmp(cadena1, cadena2, 5);
```

Esta sentencia asigna 0 a la variable *i*, ya que las cadenas "Turbo" y "TURBO" difieren sólo en que son mayúsculas o minúsculas.

---

## 12.8. INVERSIÓN DE CADENAS

La biblioteca `STRING.H` incluye la función `strrev()` que sirve para invertir los caracteres de una cadena. Su prototipo es:

```
char *strrev(char *s);
```

`strrev()` invierte el orden de los caracteres de la cadena especificada en el argumento `s`; devuelve un puntero a la cadena resultante.

---

### Ejemplo 12.16

*Muestra de inversión de cadenas.*

```
char cadena[] = "Hola";
strrev(cadena);
puts(cadena); /* visualiza "aloH" */
```

El programa siguiente invierte el orden de la cadena `Hola mundo`

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

int main(void)
{
 char *cadena = "Hola mundo";
 strrev(cadena);
 printf("\nCadena inversa: %s\n", cadena);
 return 0;
}
```

Estas dos sentencias

```
strrev(cadena);
printf("\nCadena inversa: %s\n", cadena);
```

se podrían haber sustituido por

```
printf("\nCadena inversa: %s\n", strrev(cadena));
```

## 12.9. CONVERSIÓN DE CADENAS

La biblioteca `STRING.H` de la mayoría de los compiladores C suele incluir funciones para convertir los caracteres de una cadena a letras mayúsculas y minúsculas respectivamente. Estas funciones se llaman `strlwr()` y `strupr()` en compiladores de AT&T y Borland, mientras que en Microsoft se denominan `_strlwr()` y `_strupr()`.

### 12.9.1. Función `strupr()`

La función `strupr()` convierte las letras minúsculas de una cadena a mayúsculas. Su prototipo es:

```
char *strupr(char *s);
```

---

**Ejemplo 12.17**

*Este programa convierte los caracteres en minúsculas de una cadena a mayúsculas; se escribe la cadena por pantalla.*

```

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

int main(void)
{
 char *cadena = "abcdefg";

 strupr(cadena);
 printf("La cadend convertida es: %s\n", cadena);
 return 0;
}

```

---

**12.9.2. Función striwr()**

*La función **striwr()** convierte las letras mayúsculas de una cadena a letras minúsculas. Su prototipo es:*

```
char *strlwr(char *s);
```

---

**Ejemplo 12.18**

*Una cadena formada por caracteres en mayúsculas se convierten en minúsculas.*

```

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

int main(void)
{
 char *cadena = "ABCDEFGG";

 strlwr(cadena);
 printf("La cadena convertida es: %s\n", cadena);
 return 0;
}

```

---



---

**Ejercicio 12.5**

*Se desea encontrar una cadena que sea palíndromo. El programa lee cadenas hasta encontrar un palíndromo.*

**Análisis**

La cadena se lee con `gets()`, se transforman todos los caracteres a mayúsculas con la función `strupr()`, se obtiene la cadena inversa con `strrev()` y se comparan con `strcmp()`.

No hubiera hecho falta convertir a mayúsculas si la comparación de cadenas se hubiera hecho con `stricmp()`.

```

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

int main void)
{

```

```

char ctr[81], ictr[81];
puts("\n\tIntroducir una frase hasta que sea capicua.");
do {
 gets(ctr);
 strupr(ctr); /* Todos los caracteres en mayúsculas */
 strcpy(ictr,ctr);
 strrev(ctr); /* Invierte la cadena */
} while (strcmp(ctr,ictr)); /*termina el bucle cuando son iguales */
printf("\nCadena %s es palíndromo",ictr);
return 0;
}

```

---

## 12.10. CONVERSIÓN DE CADENAS A NÚMEROS

Es muy frecuente tener que convertir números almacenados en cadenas de caracteres a tipos de datos numéricos. C proporciona las funciones `atoi()`, `atof()` y `atol()`, que realizan estas conversiones. Estas tres funciones se incluyen en la biblioteca `STDLIB.H`, por lo que ha de incluir en su programa la directiva

```
#include <stdlib.h>
```

### 12.10.1. Función `atoi()`

La función `atoi()` convierte una cadena a un valor entero. Su prototipo es:

```
int atoi(const char "cad");
```

`atoi()` convierte la cadena apuntada por `cad` a un valor entero. La cadena debe tener la representación de un valor entero y el formato siguiente:

```

[espacio en blanco] [signo] [ddd]
[espacio en blanco] = cadena opcional de tabulaciones y espacios
[signo] = un signo opcional para el valor
[ddd] = la cadena de dígitos

```

Una cadena que se puede convertir a un entero es:

```
"1232"
```

Sin embargo, la cadena siguiente no se puede convertir a un valor entero:

```
"-1234596.495"
```

La cadena anterior se puede convertir a un número de coma flotante con la función `atof()`.

Si la cadena no se puede convertir, `atoi()` devuelve cero.

---

### Ejemplo 12.19

*Convierte los dígitos de una cadena en un valor entero.*

```

char *cadena = "453423";
int valor;
valor = atoi(cadena);

```

---

### 12.10.2. Función `atof()`

La función `atof()` convierte una cadena a un valor de coma flotante. Su prototipo es:

```
double atof(const char *cad);
```

`atof()` convierte la cadena apuntada por `cad` a un valor `double` en coma flotante. La cadena de caracteres debe tener una representación de caracteres de un número de coma flotante. La conversión termina cuando se encuentre un carácter no reconocido. Su formato es:

```
[espacio en blanco][signo][ddd][.][ddd[e/E][signo][ddd]
```

#### Ejemplo 12.20

*Convierte los dígitos de una cadena a un número de tipo `double`.*

```
char *cadena = "545.7345";
double valor;
valor = atof(cadena);
```

### 12.10.3. Función `atol()`

La función `atol()` convierte una cadena a un valor largo (`long`). Su prototipo es:

```
long atol(const char *cad);
```

La cadena a convertir debe tener un formato de valor entero largo:

```
[espacio en blanco] [signo] [ddd]
```

#### Ejemplo 12.21

*Una cadena que tiene dígitos consecutivos se convierte en entero largo.*

```
char *cadena = "45743212";
long valor;
valor = atol(cadena);
```

### 12.10.4. Entrada de números y cadenas

Un programa puede necesitar una entrada que consista en un valor numérico y a continuación una cadena de caracteres. La entrada del valor numérico se puede hacer con `scanf()` y la cadena con `gets()`.

#### Ejemplo 12.22

*Lectura de un entero largo y a continuación una cadena.*

```
long int k;
char cad[81];
printf("Metros cuadrados: "); scanf("%ld",&k);
```

```
printf ("Nombre de la finca: "); gets (cad);
```

Al ejecutarse este fragmento de código, en pantalla sale

```
Metros cuadrados: 1980756
Nombre de la finca:
```



No se puede introducir el nombre de la finca, el programa le asigna la cadena vacía. ¿Por qué?: al teclear 1980756 y retorno de carro se asigna la cantidad a `k` y queda en el buffer interno el carácter fin de línea, que es el carácter en que termina la captación de una cadena por `gets()`, por lo que no se le asigna ningún carácter a `cad`. Para solucionar este problema tenemos dos alternativas, la primera:

```
printf ("Metros cuadrados: "); scanf ("%ld%c", &k);
printf ("Nombre de la finca: "); gets (cad);
```

Después de captar el número, `%c`, hace que se lea el siguiente carácter y no se asigne, así se queda el buffer de entrada vacío y `gets (cad)` puede captar la cadena que se teclee. La segunda alternativa es leer el valor numérico como una cadena de dígitos y después transformarlo con `atol (cad)` a entero largo.

```
printf ("Metros cuadrados: "); gets (cad);
k = atol (cad);
printf ("Nombre de la finca: "); gets (cad);
```

## 12.11. BÚSQUEDA DE CARACTERES Y CADENAS

La biblioteca `STRING.H` contiene un número de funciones que permiten localizar caracteres en cadenas y subcadenas en cadenas.

|                                            |                                             |                                              |                     |
|--------------------------------------------|---------------------------------------------|----------------------------------------------|---------------------|
| <i>Funciones de búsqueda de caracteres</i> | <code>strchr</code><br><code>strcspn</code> | <code>strrchr</code><br><code>strpbrk</code> | <code>strspn</code> |
| <i>Funciones de búsqueda de cadenas</i>    | <code>strstr</code>                         | <code>strtok</code>                          |                     |

### 12.11.1. La función `strchr()`

El prototipo de la función `strchr()` es

```
char *strchr(const char *s, int c);
```

`strchr()` permite buscar caracteres y patrones de caracteres en cadenas; localiza la primera ocurrencia de un carácter `c` en una cadena `s`. La búsqueda termina en la primera ocurrencia de un carácter coincidente.

### Ejemplo 12.23

*Búsqueda del carácter 'v' en una cadena.*

```
char cad[81] = "C lenguaje de medio nivel";
char *cadPtr;
cadPtr = strchr(cad, 'v');
```



## Ejecución

Longitud = 4

Este resultado se obtiene porque el primer carácter de `cadena1` es 3 y pertenece a `cadena2`, los tres caracteres siguientes a 12 pertenecen a `cadena2`.

### 12.11.4. La función `strcspn()`

La función `strcspn()` encuentra el índice del primer carácter de la primera cadena `s1` que está en el conjunto de caracteres especificado en la segunda cadena `s2`. El prototipo de `strcspn` es:

```
size_t strcspn(const char *s1, const char *s2);
```

---

#### Ejemplo 12.26

*Búsqueda de la primera posición del carácter 'd' o 'w' en una cadena.*

```
char cadena[] = "Los manolos de Carchelejo";
int i;
i = strcspn(cadena, "dw");
```

---

El ejemplo anterior asigna 12 (posición del carácter d en `cadena`) a la variable `i`.

### 12.11.5. La función `strpbrk()`

La función `strpbrk()` recorre una cadena buscando caracteres pertenecientes a un conjunto de caracteres especificado. El prototipo es

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

Esta función devuelve un puntero a la primera ocurrencia de cualquier carácter de `s2` en `s1`. Si las dos cadenas no tienen caracteres comunes se devuelve `NULL`.

---

#### Ejemplo 12.27

*Encuentra la dirección en `cad` del primer carácter encontrado que pertenezca a `subcad`.*

```
char *cad = "Hello Dolly, hey Julio";
char *subcad = "hy";
char *ptr;

ptr = strpbrk(cad, subcad);
printf("\n%s\n", ptr);
```

---

El segmento de programa visualiza "y, hey Julio", ya que encuentra "y" en la cadena antes que la "h".

### 12.11.6. La función `strstr()`

La biblioteca `STRING.H` contiene las funciones `strstr()` y `strtok()`, que permiten localizar una subcadena en una cadena o bien romper una cadena en subcadenas. La función `strstr()` busca una cadena dentro de otra cadena. El prototipo de la función es

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

La función devuelve un puntero al primer carácter de la cadena `s1` que coincide con la cadena `s2`. Si la subcadena `s2` no está en la cadena `s1`, la función devuelve `NULL`.

---

#### Ejemplo 12.28

*Búsqueda de la cadena "456" en cad1.*

```
char *cad1 = "123456789";
char *cad2 = "456";
char *resultado;

resultado = strstr(cad1, cad2);
printf("\n%s\n", resultado);
```

---

El segmento de programa anterior visualiza 456789

### 12.11.7. La función `strtok()`

La función `strtok()` permite romper una cadena en subcadenas, basada en un conjunto especificado de caracteres de separación. Su prototipo es

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

`strtok()` lee la cadena `s1` como una serie de cero o más símbolos y la cadena `s2` como el conjunto de caracteres que se utilizan como separadores de los símbolos de la cadena `s1`. Los símbolos en la cadena `s1` pueden encontrarse separados por un carácter o más del conjunto de caracteres separadores de la cadena `s2`. La segunda y posteriores llamadas a `strtok()` ha de hacerse con el primer argumento a `NULL` cuando devuelva `NULL`.

---

#### Ejercicio 11.6

*Este programa rompe una cadena en subcadenas y se imprime cada una de ellas.*

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

int main0
{
 char *cad = "Pepe Luis + Canovas * Marcos";
 char *separador = "+*";
 char *ptr = cad;

 printf("\n%s\n", cad);
 ptr = strtok(cad, separador);

 /* Anterior llamada, devuelve dirección a primer
 carácter y sitúa un NULL en el primer carácter
 coincidente con algún carácter de s2
```

```

*/
printf("\tSe rompe en tres subcadenas");
while (ptr)
{
 printf("\n%s",ptr) ;
 ptr = strtok(NULL, separador);
 /*Devuelve dirección primer carácter
 (a partir de subcadena anterior) y situa NULL en
 primer carácter coincidente con alguno de s2 */
}
return 0;

```

Al ejecutar este programa se visualiza:

```

Pepe Luis + Canovas * Marcos
 Se rompe en tres subcadenas
Pepe Luis
Canovas
Marcos

```

## 12.12. RESUMEN

En este capítulo se han examinado las funciones de manipulación de cadenas incluidas en el archivo de cabecera `STRING.H`. Los temas tratados han sido:

- Las cadenas en C son `mays` de caracteres que terminan con el carácter nulo (el carácter 0 de ASCII).
- La entrada de cadenas requiere el uso de la función `gets()`.
- \* La biblioteca `STRING.H` contiene numerosas funciones de manipulación de cadenas; entre ellas, se destacan las funciones que soportan asignación, concatenación, conversión, inversión y búsqueda.
- C soporta dos métodos de asignación de cadenas. El primer método, asigna una cadena a otra, cuando se declara esta última. El segundo método, utiliza la función `strcpy()`, que puede asignar una cadena a otra en cualquier etapa del programa.
- La función `strlen()` devuelve la longitud de una cadena.
- Las funciones `strcat()` y `strncat()` permiten concatenar dos cadenas. La función `strncat()` permite especificar el número de caracteres a concatenar.
- Las funciones `strcmp()`, `stricmp()`, `strncmp()` y `strnicmp()` permiten realizar diversos tipos de comparaciones. Las funciones `strcmp()` y `stricmp()` realizan una comparación de dos cadenas, sin tener en cuenta mayúsculas y minúsculas. La función `strncmp()` es una variante de la función `strcmp()`, que utiliza un número especificado de caracteres al comparar las cadenas. La función `strnicmp()` es una versión de la función `strncmp()` que realiza una versión con independencia del tamaño de las letras.
- Las funciones `strlwr()` y `strupr()` convierten los caracteres de una cadena en letras minúsculas y mayúsculas respectivamente.
- La función `strrev()` invierte el orden de caracteres en una cadena.
- Las funciones `strchr()`, `strspn()`, `strcspn()` y `strpbrk()` permiten buscar caracteres y patrones de caracteres en cadenas.
- La función `strstr()` busca una cadena en otra cadena. La función `strtok()` rompe (divide) una cadena en cadenas más pequeñas (subcadenas) que se separan por caracteres separadores especificados.

Asimismo, se han descrito las funciones de conversión de cadenas de tipo numérico a datos de tipo numérico. C proporciona las siguientes funciones de conversión: `atoi(s)`, `atol(s)` y `atof(s)`, que convierten el argumento `s` (cadena) a enteros, enteros largos y reales de coma flotante.

## 12.13. EJERCICIOS

- 12.1. Teniendo en cuenta el siguiente segmento de código, indicar los errores y la forma de corregirlos.

```
char *b = "Descanso activo";
char *p = b;
char c[] = "Para recuperar";
char* cd;
cd = c;
cd = "Asigna cadena";
```

- 12.2. Se quiere leer del dispositivo estándar de entrada las  $n$  códigos de asignaturas de la carrera de Sociología. Escribe un segmento de código para realizar este proceso.

- 12.3. Para entrada de cadenas de caracteres, qué diferencia existe entre `scanf("%s", cadena)` y `gets(cadena)` ¿En qué casos será mejor utilizar una u otra?

- 12.4. En el siguiente código C se lee un número real y una cadena de caracteres. ¿Qué problemas surgen y por qué? ¿Cómo resolverlo?

```
float x;
char nom[61];
printf('Distancia en Km: ');
scanf("%f", &x);
printf("Nombre del pueblo: ");
gets(nom);
```

- 12.5. Define un array de cadenas de caracteres para poder leer un texto compuesto por un máximo de 80 líneas. Escribe una función para leer el texto; la función debe de tener dos argumentos, uno el texto y el segundo el número de líneas.

- 12.6. Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.

- 12.7. ¿Qué diferencias y analogías existen entre las variables  $c1, c2, c3$ ? La declaración es:

```
char **c1;
char *c2[10];
char *c3[10][21];
```

- 12.8. Escribe una función que obtenga una cadena del dispositivo de entrada, de igual forma que `char* gets(char*)`. Utilizar para ello `getchar()`.

- 12.9. Escribir una función que obtenga una cadena del dispositivo estándar de entrada. La cadena termina con el carácter de fin de línea, o bien cuando se han leído  $n$  caracteres. La función devuelve un puntero a la cadena leída, o EOF si se alcanzó el fin de fichero. El prototipo de la función debe de ser:

```
char* lee_linea(char *c, int n);
```

- 12.10. La función `atoi()` transforma una cadena formada por dígitos decimales en el equivalente número entero. Escribir una función que transforme una cadena formada por dígitos hexadecimales en un entero largo.

- 12.11. Escribir una función para transformar un número entero en una cadena de caracteres formada por los dígitos del número entero.

- 12.12. Escribir una función para transformar un número real en una cadena de caracteres que sea la representación decimal del número real.

## 12.14. PROBLEMAS

- 12.1. Escribir un programa que lea un texto de como máximo 60 líneas, cada línea con un máximo de 80 caracteres. Una vez leído el texto intercambiar la línea de mayor longitud por la línea de menor longitud.
- 12.2. Escribir un programa que lea una línea de texto y escriba en pantalla las palabras de que consta la línea. Utilizar las funciones de `string-h`.
- 12.3. Se tiene un texto formado por un máximo de 30 líneas, del cual se quiere saber el número de apariciones de la palabra `CLAVE`. Escribir un programa que lea el texto y la palabra `CLAVE`, determine el número de apariciones de `CLAVE` en el texto.
- 12.4. Se tiene un texto de 40 líneas. Las líneas tienen un número de caracteres variable. Escribir un programa para almacenar el texto en una matriz de líneas, ajustada la longitud de cada línea al número de caracteres. El programa debe de leer el texto, almacenarlo en la estructura matricial y escribir por pantalla las líneas en orden creciente de su longitud.
- 12.5. Escribir un programa que lea líneas de texto, obtenga las palabras de cada línea y las escriba en pantalla en orden alfabético. Se puede considerar que el máximo número de palabras por línea es 28.
- 12.6. Se quiere leer un texto de como máximo 30 líneas. Se quiere que el texto se muestre de tal forma que aparezcan las líneas en orden alfabético.
- 12.7. Se sabe que en las líneas de que forma un texto hay valores numéricos enteros, representan los Kg de patatas recogidos en una finca. Los valores numéricos están separados de las palabras por un blanco, o el carácter fin de línea. Escribir un programa que lea el texto y obtenga la suma de los valores numéricos.
- 12.8. Escribir un programa que lea una cadena clave y un texto de como máximo 50 líneas. El programa debe de eliminar las líneas que contengan la clave.
- 12.9. Se quiere sumar números grandes, tan grandes que no pueden almacenarse en variables de tipo `long`. Por lo que se ha pensado en introducir cada número como una cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas. Hay que tener en cuenta que la cadena suma puede tener un carácter más que la máxima longitud de los sumandos.
- 12.10. Un texto está formado por líneas de longitud variable. La máxima longitud es de 80 caracteres. Se quiere que todas las líneas tengan la misma longitud, la de la cadena más larga. Para ello se debe de cargar con blancos por la derecha las líneas hasta completar la longitud requerida. Escribir un programa para leer un texto de líneas de longitud variable y formar el texto para que todas las líneas tengan la longitud de la máxima línea.
- 12.11. Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres en el mismo o en diferente orden. Hay que ignorar los blancos y considerar que las mayúsculas y las minúsculas son iguales.

PARTE III

**ESTRUCTURA DE DATOS**

## CAPÍTULO 13

# ENTRADAS Y SALIDAS POR ARCHIVOS

---

### CONTENIDO

- |                                              |                                                  |
|----------------------------------------------|--------------------------------------------------|
| <b>13.1.</b> Flujos.                         | <b>13.6.</b> Funciones para acceso aleatorio.    |
| <b>13.2.</b> Puntero <code>FILE</code> .     | <b>13.7.</b> Argumentos de <code>main()</code> . |
| <b>13.3.</b> Apertura de un archivo.         | <b>13.8.</b> <i>Resumen.</i>                     |
| <b>13.4.</b> Creación de archivo secuencial. | <b>13.9.</b> <i>Ejercicios.</i>                  |
| <b>13.5.</b> Archivos binarios en C.         | <b>13.10.</b> <i>Problemas.</i>                  |

---

## INTRODUCCIÓN

Hasta este momento se han realizado las operaciones básicas de entrada y salida. La operación de introducir (*leer*) datos en el sistema se denomina **lectura** y la generación de datos del sistema se denomina **escritura**. La lectura de datos se realiza desde su teclado e incluso desde su unidad de disco, y la escritura de datos se realiza en el monitor y en la impresora de su sistema.

**Las** funciones de entrada/salida no están definidas en el propio lenguaje C, **sino** que están incorporadas en cada compilador de C bajo la forma de *biblioteca de ejecución*. En C existe la biblioteca *stdio.h* estandarizada por ANSI; esta biblioteca proporciona tipos de datos, macros y funciones para acceder a los archivos. El manejo de archivos en C se hace mediante el concepto de **flujo** (*streams*) o canal, o también denominado secuencia. **Los** flujos pueden estar abiertos o cerrados, conducen los datos entre el programa y los dispositivos externos. Con las funciones proporcionadas por la biblioteca se pueden tratar archivos secuenciales, de acceso directo, archivos indexados, etc.

En este capítulo aprenderá a utilizar las características típicas de E/S para archivos en C, así como las funciones de acceso más utilizadas.

---

## CONCEPTOS CLAVE

- Acceso aleatorio.
- Acceso secuencial.
- Apertura y cierre de un archivo.
- Archivos binarios.
- Archivos de caracteres.
- Archivos indexados.
- Colisiones de claves.
- Flujos.
- Registro lógico.
- Transformación de claves.

## 13.1. FLUJOS

Un **flujo** (*stream*) es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal («*pipe*») por la que circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo, por el canal que comunica el archivo con el programa van a fluir las secuencias de datos. Hay tres flujos o canales abiertos automáticamente:

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Estas tres variables se inicializan al comenzar la ejecución del programa para admitir secuencias de caracteres, en modo texto. Su cometido es el siguiente:

```
stdin asocia la entrada estándar (teclado) con el programa.
stdout asocia la salida estándar (pantalla) con el programa.
stderr asocia la salida de mensajes de error (pantalla) con el programa.
```

Así cuando se ejecuta `printf ("Calle Mayor 2.");` se escribe en `stdout`, en pantalla; si se desea leer una variable entera con `scanf ("%d", &x);` se captan los dígitos de la secuencia de entrada `stdin`.

El acceso a los archivos se hace con un *buffer* intermedio. Se puede pensar en el *buffer* como un array donde se van almacenando los datos dirigidos al archivo, o desde el archivo; el *buffer* se vuelca cuando de una forma u otra se da la orden de vaciarlo. Por ejemplo, cuando se llama a una función para leer del archivo una cadena, la función lee tantos caracteres como quepan en el *buffer*. A continuación se obtiene la cadena del *buffer*; una posterior llamada a la función obtendrá la siguiente cadena del *buffer* y así sucesivamente hasta que se quede vacío y se llene con una llamada posterior a la función de lectura.

El lenguaje C trabaja con archivos con *buffer*, y está diseñado para acceder a una amplia gama de dispositivos, de tal forma que trata cada dispositivo como una secuencia, pudiendo haber secuencias de caracteres y secuencias binarias. Con las secuencias se simplifica el manejo de archivo en C.

## 13.2. Puntero FILE

Los archivos se ubican en dispositivos externos como cintas, cartuchos, discos, disco compactos, etc. y tienen un nombre y unas características. En el programa el archivo tiene un nombre interno que es un puntero a una estructura predefinida (*puntero a archivo*). Esta estructura contiene información sobre el archivo, tal como la dirección del buffer que utiliza, el modo de apertura del archivo, el último carácter leído del buffer y otros detalles que generalmente el usuario no necesita saber. El identificador del tipo de la estructura es `FILE` y esta declarada en el archivo de cabecera `stdio.h`:

```
typedef struct{
 short level;
 unsigned flags; /*estado del archivo: lectura, binario ... */
 char fd;
 unsigned char hold;
 short bsize;
 unsigned char *buffer, *curp;
 unsigned istemp;
 short token;
}FILE;
```

El detalle de los campos del tipo `FILE` puede cambiar de un compilador a otro. Al programador le interesa saber que existe el tipo `FILE` y que es necesario definir un puntero a `FILE` por cada archivo a

procesar. Muchas de las funciones para procesar archivos son del tipo `FILE *`, y tienen argumento(s) de ese tipo.

### Ejemplo 13.1

Se declara un puntero a `FILE`; se escribe el prototipo de una función de tipo puntero a `FILE` y con un argumento del mismo tipo.

```
FILE* pf;
FILE* mostrar(FILE*); /* Prototipo de una función definida por el
 programador* /
```

Cabe recordar que la entrada estándar al igual que la salida están asociadas a variables puntero a `FILE`:

```
FILE *stdin, *stdout;
```

## 13.3. Apertura de un archivo

Para procesar un archivo en C (y en todos los lenguajes de programación) la primera operación que hay que realizar es abrir el archivo. La apertura del archivo supone conectar el archivo externo con el programa, e indicar cómo va a ser tratado el archivo: binario, de caracteres, etc. El programa accede a los archivos a través de un puntero a la estructura `FILE`, la función de apertura devuelve dicho puntero. La función para abrir un archivo es `fopen()` y el formato de llamada es:

```
fopen(nombre_archivo, modo);
```

|        |          |                                                    |
|--------|----------|----------------------------------------------------|
| nombre | ≡ cadena | Contiene el identificador externo del archivo.     |
| modo   | ≡ cadena | Contiene el modo en que se va a tratar el archivo. |

La función devuelve un puntero a `FILE`, a través de dicho puntero el programa hace referencia al archivo. La llamada a `fopen()` se debe de hacer de tal forma que el valor que devuelve se asigne a una variable puntero a `FILE`, para así después referirse a dicha variable.

### Ejemplo 13.2

Declara una variable de tipo puntero a `FILE`. A continuación escribir una sentencia de apertura de un archivo.

```
FILE* pf;
pf = fopen(nombre_archivo, modo);
```

La función puede detectar un error al abrir el archivo, por ejemplo que el archivo no exista y se quiera leer, entonces devuelve `NULL`.

### Ejemplo 13.3

Se desea abrir un archivo de nombre `LICENCIA.EST` para obtener ciertos datos.

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

FILE *pf;
char nm[] = "C:\\LICENCIA.EST";
```

```

pf = fopen(nm, "r");
if (pf == NULL)
{
 puts("Error al abrir el archivo.");
 exit(1);
}

```

### Ejemplo 13.4

En este ejemplo se abre el archivo de texto JARDINES.DAT para escribir en él los datos de un programa.

En la misma línea en que se ejecuta `fopen()` se comprueba que la operación ha sido correcta, en caso contrario termina la ejecución.

```

#include <stdio.h>
#include <stdlib.h>

FILE *ff;
char* arch = "C:\\AMBIENTE\\JARDINES.DAT";

if ((ff = fopen(nm, "w")) == NULL)
{
 puts("Error al abrir el archivo para escribir.");
 exit(-1);
}

```

El prototipo de `fopen()` se encuentra en el archivo `stdio.h`, es el siguiente:

```
FILE* fopen (const char* nombre-archivo, const char* modo);
```

#### 13.3.1. Modos de apertura de un archivo

Al abrir el archivo `fopen()` se espera como segundo argumento el modo de tratar el archivo. Fundamentalmente se establece si el archivo es de lectura, escritura o añadido; y si es de texto o binario. Los modos básicos se expresan en esta tabla:

| Modo | Significado                                                                       |
|------|-----------------------------------------------------------------------------------|
| "r"  | Abre para lectura.                                                                |
| "w"  | Abre para crear nuevo archivo (si ya existe se pierden sus datos).                |
| "a"  | Abre para añadir al final.                                                        |
| "r+" | Abre archivo ya existente para modificar (leer/escibir).                          |
| "w+" | Crea un archivo para escribir/leer (si ya existe se pierden los datos).           |
| "a+" | Abre el archivo para modificar (escribir/leer) al final. Si no existe es como w+. |

En estos modos no se ha establecido el tipo del archivo, de texto o binario. Siempre hay una opción por defecto y aunque depende del compilador utilizado, suele ser modo texto. Para no depender del entorno es mejor indicar si es de texto o binario. Se utiliza la letra *t* para modo texto, la *b* para modo binario como último carácter de la cadena modo (también se puede escribir como carácter intermedio). Por consiguiente, los modos de abrir un archivo de texto:

```
"rt", "wt", "at", "r+t", "w+t", "a+t".
```

Y los modos de abrir un archivo binario:

"rb", "wb", "ab", "r+b", "w+b", "a+b".

---

### Ejemplo 13.5

Se dispone archivo de texto LICENCIA.EST, se quiere leerlo para realizar un cierto proceso y escribir datos resultantes en el archivo hinario RESUMEN.REC. Las operaciones de apertura son:

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

FILE *pf1, *pf2;
char org[] = "C:\\LICENCIA.EST";
char dst[] = "C:\\RESUMEN.REC";

pf1 = fopen(org, "rt");
pf2 = fopen(dst, "wb");
if (pf1 == NULL || pf2 == NULL)

 puts("Error al abrir los archivos.");
 exit(1);
}
```

---

### 13.3.2. NULL y EOF

Las funciones de biblioteca que devuelven un puntero (`strcpy()`, `fopen()` ...) especifican que si no pueden realizar la operación (generalmente si hay un error) devuelven `NULL`. Esta es una macro definida en varios archivos de cabecera, entre los que se encuentran `stdio.h` y `stdlib.h`.

Las funciones de librería de E/S de archivos, generalmente empiezan por `f` de `file`, tienen especificado que son de tipo entero de tal forma que si la operación falla devuelven `EOF`, también devuelven `EOF` para indicar que se ha leído el fin de archivo. Esta macro está definida en `stdio.h`.

---

### Ejemplo 13.6

El siguiente segmento de código lee del flujo estándar de entrada hasta fin de archivo:

```
int c;
while ((c=getchar()) != EOF)
{
}
}
```

---

### 13.3.3. Cierre de archivos

Los archivos en C trabajan con una memoria intermedia, son con *buffer*. La entrada y salida de datos se almacena en ese *buffer*, volcándose cuando está lleno. Al terminar la ejecución del programa podrá ocurrir que haya datos en el *buffer*, si no se volcasen en el archivo quedaría este sin las últimas actualizaciones. Siempre que se termina de procesar un archivo y siempre que se termine la ejecución del programa los archivos abiertos hay que cerrarlos para que entre otras acciones se vuelque el *buffer*.

La función `fclose(puntero_file)` cierra el archivo asociado al `puntero_file`, devuelve EOF si ha habido un error al cerrar. El prototipo de la función se encuentra en `stdio.h` y es:

```
int fclose(FILE* pf);
```

---

### Ejemplo 13.7

*Abrir dos archivos de texto, después se cierra cada uno de ellos.*

```
#include <stdio.h>
FILE *pf1, *pf2;

pf1 = fopen("C:\DATOS.DAT", "a+");
pf2 = fopen("C:\TEMPS.RET", "b+");

fclose(pf1);
fclose(pf2);
```

---

## 13.4. CREACIÓN DE UN ARCHIVO SECUENCIAL

Una vez abierto un archivo para escribir datos hay que grabar los datos en el archivo. La biblioteca *C* proporciona diversas funciones para escribir datos en el archivo a través del puntero a `FILE` asociado.

Las funciones de entrada y de salida de archivos tienen mucho parecido con las funciones utilizadas para entrada y salida para los flujos `stdin` (teclado) y `stdout` (pantalla): `printf()`, `scanf()`, `getchar()`, `putchar()`, `gets()` y `puts()`. Todas tienen una versión para archivos que empieza por la letra *f*, así se tiene `fprintf()`, `fscanf()`, `fputs()`, `fgets()`; la mayoría de las funciones específicas de archivos empiezan por *f*.

### 13.4.1. Funciones `putc()` y `fputc()`

Ambas funciones son idénticas, `putc()` está definida como macro. Escriben un carácter en el archivo asociado con el puntero a `FILE`. Devuelven el carácter escrito, o bien EOF si no puede ser escrito. El formato de llamada:

```
putc(c, puntero-archivo);
fputc(c, puntero-archivo);
```

siendo *c* el carácter a escribir.

---

### Ejercicio 13.1

*Se desea crear un archivo `SALIDA.PTA` con los caracteres introducidos por teclado.*

#### *Análisis*

Una vez abierto el archivo, un bucle mientras (`while`) no sea fin de archivo (macro EOF) lee carácter a carácter y se escribe en el archivo asociado al puntero `FILE`.

```
#include <stdio.h>
int main()
{
 int c;
 FILE* pf;
 char *salida = "SALIDA.TXT";
```

```

if ((pf = fopen(salida,"wt"))== NULL)
{
 puts("ERROR EN LA OPERACION DE APERTURA");
 return 1;
}

while ((c=getchar())!=EOF)
{
 putc(c,pf);
}

fclose(pf);
return 0;
}

```

En el Ejercicio 13.1 en vez de `putc(c, pf)` se puede utilizar `fputc(c, pf)`. El prototipo de ambas funciones se encuentra en `stdio.h`, es el siguiente:

```

int putc(int c, FILE* pf);
int fputc(int c, FILE* pf);

```

### 13.4.2. Funciones `getc()` y `fgetc()`

Estas dos funciones son iguales, igual formato e igual funcionalidad; pueden considerarse que son recíprocas de `putc()` y `fputc()`. Éstas, `getc()` y `fgetc()`, leen un carácter (el siguiente carácter) del archivo asociado al puntero a `FILE`. Devuelven el carácter leído o EOF si es fin de archivo (o si ha habido un error). El formato de llamada es:

```

getc (puntero_archivo);
fgetc (puntero_archivo);

```

#### Ejercicio 13.2

El archivo `SALIDA.PTA`, creado en el Problema 13.1, se desea leer para mostrarlo por pantalla y contar las líneas que tiene.

##### Análisis

Una vez abierto el archivo de texto en modo lectura, un bucle **mientras** no sea fin de archivo (macro EOF) lee carácter a carácter y se escribe en pantalla. En el caso de leer el carácter de fin de línea se debe saltar a la línea siguiente y contabilizar una línea más.

```

#include <stdio.h>
int main()
{
 int c, n=0;
 FILE* pf;
 char *nombre = "\\SALIDA.TXT";

 if ((pf = fopen(nombre,"rt")) == NULL)
 {
 puts ("ERROR EN LA OPERACION DE APERTURA");
 return 1;
 }

```

```

while ((c=getc(pf))!=EOF)
{
 if (c == '\n')
 {
 n++; printf("\n");
 }
 else
 putchar (c);
}

printf("\nNúmero de líneas del archivo: %d",n);
fclose(pf);
return 0;
}

```

---

El prototipo de ambas funciones se encuentra en `stdio.h` y es el siguiente:

```

int getc(FILE* pf);
int fgetc(FILE* pf);

```

### 13.4.3. Funciones `fputs()` y `fgets()`

Estas funciones escriben/leen una cadena de caracteres en el archivo asociado. La función `fputs()` escribe una cadena de caracteres. La función devuelve `EOF` si no ha podido escribir la cadena, un valor no negativo si la escritura es correcta; el formato de llamada es:

```
fputs(cadena, puntero-archivo);
```

La función `fgets()` lee una cadena de caracteres del archivo. Termina la captación de la cadena cuando lee el carácter de fin de línea, o bien cuando ha leído  $n-1$  caracteres, siendo  $n$  un argumento entero de la función. La función devuelve un puntero a la cadena devuelta, o `NULL` si ha habido un error. El formato de llamada es:

```
fgets(cadena, n, puntero_archivo);
```

---

#### Ejemplo 13.8

*Lectura de un máximo de 80 caracteres de un archivo:*

```

#define T 81
char cad[T];
FILE *f;

fgets(cad, T, f);

```

---

#### Ejercicio 13.3

*El archivo `CARTAS.DAT` contiene un texto al que se le desea añadir nuevas líneas, de longitud mínima 30 caracteres, desde el archivo `PRIMERO.DAT`.*

##### Análisis

El problema se resuelve abriendo el primer archivo en modo añadir ("`a`") , el segundo archivo en modo lectura ("`r`"). Las líneas se leen con `fgets()`, si cumplen la condición de longitud se escriben en el

archivo CARTAS. Al tener que realizar un proceso completo del archivo, se realizan iteraciones mientras no fin de archivo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MX 121
#define MN 30

int main()
{
 FILE *in, *out;
 char nom1[] = "\\CARTAS.DAT";
 char nom2[] = "\\PRIMERO.DAT";
 char cad[MX];

 in = fopen(nom2, "rt");
 out = fopen(nom1, "at");
 if (in==NULL || out==NULL)
 {
 puts("Error al abrir archivos. ");
 exit(-1);
 }

 while (fgets(cad, MX, in)) /*itera hasta que devuelve puntero NULL*/
 {
 if (strlen(cad) >= MN)
 fputs(cad, out)
 else
 puts(cad);
 }

 fclose(in);
 fclose(out);
 return 0;
}
```

El prototipo de ambas funciones está en `stdio.h`, es el siguiente:

```
int fputs(char* cad, FILE* pf);
char* fgets(char* cad, int n, FILE* pf);
```

#### 13.4.4. Funciones `fprintf()` y `fscanf()`

Las funciones `printf()` y `scanf()` permiten escribir o leer variables cualquier tipo de dato estándar, los códigos de formato (`%d`, `%f...`) indican a *C* la transformación que debe de realizar con la secuencia de caracteres (conversión a entero...). La misma funcionalidad tiene `fprintf()` y `fscanf()` con los flujos (archivos asociados) a que se aplican. Estas dos funciones tienen como primer argumento el puntero a *file* asociado al archivo de texto.

#### Ejercicio 13.4

Se desea crear el archivo de texto `PERSONAS.DAT` de tal forma que cada línea contenga un registro con los datos de una persona que contenga los campos nombre, fecha de nacimiento (*dia(nn)*, *mes(nn)*, *año(nnnn)*) y *mes* en **ASCII**.

**Análisis**

En la estructura `persona` se declaran los campos correspondientes. Se define una función que devuelve una estructura `persona` leída del teclado. El mes en **ASCII** se obtiene de una función que tiene como entrada el número de mes y devuelve una cadena con el mes en **ASCII**. Los campos de la estructura son escritos en el archivo con `fprintf()`.

```
#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
/* declaración de tipo global estructura */
typedef struct {
 char* nm;
 int dia;
 int ms;
 int aa;
 char mes[11];
} PERSONA;

void entrada(PERSONA* p
char* mes_asci(short n)

int main()

 FILE *pff;
 char nf[] = "\\PERSONS DAT";
 char r = 'S';

 if ((pff = fopen(nf, "wt")) == NULL)
 {
 puts("Error al abrir archivos. ");
 exit(-1);
 }

 while (toupper(r) == 'S')
 {
 PERSONA pt;
 entrada(&pt);
 printf("%s %d-%d-%d %s\n", pt.nm, pt.dia, pt.ms, pt.aa, pt.mes);
 fprintf(pff, "%s %d-%d-%d %s\n", pt.nm, pt.dia, pt.ms, pt.aa, pt.mes);
 printf("Otro registro?: "); scanf("%c%c", &r);

 fclose(pff);
 return 0;
 }

void entrada(PERSONA* p)
{ char bf[81];
 printf("Nombre: "); gets(bf);
 p->nm = (char*)malloc((strlen(bf)+1)*sizeof(char));
 strcpy(p->nm, bf);
 printf("Fecha de nacimiento(dd mm aaaa): ");
 scanf("%d %d %d%c", &p->dia, &p->ms, &p->aa);
 printf("\n %s\n", mes_asci(p->ms));
 strcpy(p->mes, mes_asci(p->ms));
}

char* mes_asci(short n)
```

```

{
 static char *mes[12]= {
 "Enero", "Febrero", "Marzo", "Abril",
 "Mayo", "Junio", "Julio", "Agosto",
 "Septiembre", "Octubre", "Noviembre", "Diciembre"};

 if (n >=1 && n <= 12)
 return mes[n-1];
 else
 return "Error mes";
}

```

---

El prototipo de ambas funciones esta en `stdio.h`, y es el siguiente:

```

int fprintf(FILE* pf, const char* formato, . . .);
int fscanf(FILE* pf, const char* formato, . . .);

```

### 13.4.5. Función `feof()`

Diversas funciones de lectura de caracteres devuelven EOF cuando leen el carácter de fin de archivo. Con dicho valor, que es una macro definida en `stdio.h`, ha sido posible formar bucles para leer un archivo completo. La función `feof()` realiza el cometido anterior, devuelve un valor distinto de 0 (`true`) cuando se lee el carácter de fin de archivo, en caso contrario devuelve 0 (`false`).

---

#### Ejemplo 13.9

*El siguiente ejemplo transforma el bucle del ejercicio 13.2, utilizando la función `feof()`*

```

int c, n=0;
FILE* pf;
char *nombre = "\\SALIDA.TXT";

. . .
while (!feof(pf))
{
 c=getc(pf);
 if (c == '\n')
 {
 n++; printf("\n");
 }
}

```

---

El prototipo de la función está en `stdio.h`, es el siguiente:

```

int feof(FILE* pf);

```

### 13.4.6. Función `rewind()`

Una vez que se alcanza el fin de un archivo, nuevas llamadas a `feof()` siguen devolviendo un valor distinto de cero (`true`). Con la función `rewind()` se sitúa el puntero del archivo al inicio de éste. El formato de llamada es

```

rewind(puntero_archivo).

```

El prototipo de la función se encuentra en `stdio.h`:

```
void rewind(FILE*pf);
```

---

### Ejemplo 13.10

Este ejemplo lee un archivo de texto, cuenta el número de líneas que contiene y a continuación sitúa el puntero del archivo al inicio para una lectura posterior.

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

FILE* pg;
char nom[]="PLUVIO.DAT";
char buf[121];
int nl = 0;

if ((pg = fopen(nom,"rt"))==NULL)
{
 puts("Error al abrir el archivo.");
 exit(-1);
}

while (!feof(pg))
{
 fgets(buf,121,pg); nl++;
}

rewind(pg);
/* De nuevo puede procesarse el archivo */
while (!feof(pg))
{
 // ...
}
}
```

---

## 13.5. Archivos binarios en C

Para abrir un archivo en modo binario hay que especificar la opción **b** en el modo. Los archivos binarios son secuencias de 0,s y 1,s. Una de las características de los archivos binarios es que optimizan la memoria ocupada por un archivo, sobre todo con campos numéricos. Así, almacenar en modo binario un entero supone una ocupación de 2 bytes o 4 bytes (depende del sistema), y un número real 4 bytes o 8 bytes; en modo texto primero se convierte el valor numérico en una cadena de dígitos (%6d, %8.2f...) y después se escribe en el archivo. La mayor eficiencia de los archivos binarios se contrapone con el hecho de que su lectura se tiene que hacer en modo binario y que sólo se pueden visualizar desde el entorno de un programa C. Los modos para abrir un archivo binario son los mismos que para abrir un archivo de texto, sustituyendo la **t** por **b**:

```
"rb", "wb", "ab", "r+b", "w+b", "a+b"
```

---

### Ejemplo 13.11

En este ejemplo se declaran 3 punteros a `FILE`. A continuación se abren tres archivos en modo binario.

```
FILE *pf1, *pf2, *pf3;
pf1 = fopen("gorjal.arr", "rb"); /*Lectura de archivo binario */
pf2 = fopen("tempes.feb","w+b"); /*lecr/escribir archivo binario*/
```

```
pf3 = fopen("telcon.fff","ab"); /*añadir a archivo binario*/
```

La biblioteca de C proporciona dos funciones especialmente dirigidas al proceso de entrada y salida de archivos binarios con *buffer*, son `fread()` y `fwrite()`.

### 13.5.1. Función de salida `fwrite()`

La función `fwrite()` escribe un *buffer* de cualquier tipo de dato en un archivo binario. El formato de llamada es:

```
fwrite(direction-buffer,tamaño,num_elementos,puntero_archivo);
```

---

#### Ejemplo 13.12

En el ejemplo se abre un archivo en modo binario para escritura. Se escriben números reales en doble precisión en el bucle `for`. El *buffer* es la variable `x`, el tamaño lo devuelve el operador `sizeof`.

```
FILE *fd;
double x;

fd = fopen("reales.num","wb");
for (x=0.5; x>0.01;)

 fwrite(&x, sizeof(double), 1, fd);
 x = pow(x,2.);
```

---

El prototipo de la función está en `stdio.h`:

```
size_t fwrite(const void *ptr,size_t tam,size_t n,FILE *pf);
```

El tipo `size_t` está definido en `stdio.h` y es un tipo `int`.

---

#### Ejercicio 13.5

Se dispone de una muestra de las coordenadas de puntos de un plano representada por pares de números enteros  $(x,y)$ , tales que  $1 \leq x \leq 100$  e  $1 \leq y \leq 100$ . Se desea guardar en un archivo binario todos los puntos disponibles.

##### Análisis

El nombre del archivo es `PUNTOS.DAT`. Según se lee un punto se comprueba la validez del punto y se escribe en el archivo con una llamada a la función `fwrite()`. La condición de terminación del bucle es la lectura del punto  $(0,0)$ .

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

 int x,y;
};
typedef struct punto PUNTO;

int main()
{
 PUNTO p;
 char *nom = "C:\PUNTOS.DAT";
```

```

FILE *pp;
if ((pp = fopen(nom, "wb")) == NULL)
{
 puts("\nError en la operación de abrir archivo.");
 return -1;

puts("\nIntroduce coordenadas de puntos, para acabar: (0,0)");
do {
 scanf("%d %d", &p.x, &p.y);
 while (p.x < 0 || p.y < 0)

 printf ("Coordenas deben ser >=0 :");
 scanf("%d %d", &p.x, &p.y);
 }
 if (p.x > 0 || p.y > 0)
 {
 fwrite(&p, sizeof(PUNTO), 1, pp);
 }
} while (p.x > 0 || p.y > 0);

fclose(pp);
return 0;
}

```

Los archivos binarios están indicados especialmente para guardar registros, estructuras en C. El método habitual es la escritura sucesiva de estructuras en el archivo asociado al puntero, la lectura de estos archivos es similar.

### 13.5.2. Función de lectura fread()

Esta función lee de un archivo *n* bloques de bytes y los almacena en un *buffer*. El número de bytes de cada bloque (tamaño) se pasa como parámetro, al igual que el número *n* de bloques y la dirección del buffer (*o* variable) donde se almacena. El formato de llamada:

```
fread(direccion_buffer, tamaño, n, puntero_archivo);
```

La función devuelve el número de bloques que lee y debe de coincidir con *n*. El prototipo de la función está en `stdio.h`:

```
size_t fread(void *ptr, size_t tam, size_t n, FILE *pf);
```

#### Ejemplo 13.13

*En el ejemplo se abre un archivo en modo binario para lectura. El archivo se lee hasta el final del archivo; cada lectura de un número real se acumula en la variable s.*

```

FILE *fd;
double x, s=0.0;

if ((fd= fopen("reales.num", "rb")) == NULL)
 exit(-1);
while (!eof(fd))
{
 fread(&x, sizeof(double), 1, fd);
 s+= x;
}

```

**Ejercicio 13.6**

En el Ejercicio 13.5 se ha creado un archivo hinario de puntos en el plano. Se desea escribir un programa para determinar los siguientes valores:

- $n_{i,j}$ , número de veces que aparece un punto dado  $(i,j)$  en el archivo.
- Dado un valor de  $j$ , obtener la media de  $i$  para los puntos que contienen  $a_j$ .

$$i = \frac{\sum_{i=1}^{100} n_{i,j} \cdot i}{\sum_{i=1}^{100} n_{i,j}}$$

**Análisis**

La primera instrucción es abrir el archivo binario para lectura. A continuación se solicita el punto donde se cuentan las ocurrencias en el archivo. En la función `cuenta_pto()` se determina dicho número; para lo cual hay que leer todo el archivo. Para ejecutar el segundo apartado, se solicita el valor de  $j$ . Con un bucle desde  $i=1$  hasta 100 se cuenta las ocurrencias de cada punto  $(i,j)$  llamando a la función `cuenta_pto()`; antes de cada llamada hay que situar el puntero del archivo al inicio, llamando para ello a la función `rewind()`.

```
#include <stdio.h>
struct punto
{
 int i, j;
};
typedef struct punto PUNTO;
FILE *pp;
int cuenta_pto(PUNTO w);

int main()
{
 PUNTO p;
 char *nom = "C:\PUNTOS.DAT";
 float media, nmd, dnm;

 if ((pp = fopen(nom, "rb")) == NULL)
 {
 puts("\nError al abrir archivo para lectura.");
 return -1;
 }

 printf("\nIntroduce coordenadas de punto a buscar: ");
 scanf("%d %d", &p.i, &p.j);
 printf("\nRepeticiones del punto (%d,%d): %d\n",
 p.i, p.j, cuenta_pto(p));

 /* Cálculo de la media i para un valor j */

 printf ("Valor de j: "); scanf("%d", &p.j);
 media=nmd=dnm= 0.0;
```

```

for (p.i=1; p.i<= 10; p.i++)
{
 int st;
 rewind(pp);
 st = cuenta_pto(p);
 nmd += (float)st*p.i;
 dnm += (float)st;
}

if (dnm>0.0)
 media = nmd/dnm;
printf("\nMedia de los valores de i para %d = %.2f",p.j,media);
return 0;
}

int cuenta_pto(PUNTO w)
{
 PUNTO p;
 int r;
 r = 0;
 while (!feof(pp))
 {
 fread(&p,sizeof(PUNTO),1,pp);
 if (p.i==w.i && p.j==w.j) r++;
 }
 return r;
}

```

## 13.6. Funciones para acceso aleatorio

El acceso directo —aleatorio— a los datos de un archivo se hace mediante su posición, es decir, el lugar relativo que ocupan. Tiene la ventaja de que se pueden leer y escribir registros en cualquier orden y posición. Son muy rápidos de acceso a la información que contienen. El principal inconveniente que tiene la organización directa es que necesita programar la relación existente entre el contenido de un registro y la posición que ocupan.

Las funciones `fseek()` y `ftell()` se usan principalmente para el acceso directo a archivos en C. Éstas consideran el archivo como una secuencia de bytes; el número de byte es el índice del archivo. Según se va leyendo o escribiendo registros o datos en el archivo, el programa mantiene a través de un puntero la posición actual. Con la llamada a la función `ftell()` se obtiene el valor de dicha posición. La llamada a `fseek()` permite cambiar la posición del puntero al archivo a una dirección determinada.

### 13.6.1. Función `fseek()`

Con la función `fseek()` se puede tratar un archivo en C como un array que es una estructura de datos de acceso aleatorio. `fseek()` sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos. En el Ejemplo 13.14 se supone que existe un archivo de productos, se pide el número de producto y se sitúa el puntero del archivo para leer el registro en una operación de lectura posterior.

#### Ejemplo 13.14

*Declarar una estructura (registro) PRODUCTO, y abrir un archivo para lectura. Se desea leer un registro cuyo número (posición) se pide por teclado.*

```

typedef struct
{
 char nombre[41];
 int unidades;
 float precio;
 int pedidos;
} PRODUCTO;
PRODUCTO uno;
int n, stat;
FILE* pfp;

if ((pfp = fopen("conservas.dat","r"))==NULL)
{
 puts("No se puede abrir el archivo.");
 exit(-1);
}

/* Se pide el número de registro */
printf("Número de registro: "); scanf("%d",&n);
/* Sitúa el puntero del archivo */
stat = fseek(pfp, n*sizeof(PRODUCTO),0);
/* Comprueba que no ha habido error */
if (stat != 0)
{
 puts("Error, puntero del archivo movido fuera de este");
 exit(-1);
}
/* Lee el registro */
fread(&uno, sizeof(PRODUCTO), 1, pfp);
. . .

```

---

El segundo argumento de `fseek()` es el desplazamiento, el tercero es el origen del desplazamiento, el 0 indica que empiece a contar desde el principio del archivo.

El formato para llamar a `fseek()`:

```
fseek(puntero_archivo, desplazamiento, origen);
```

desplazamiento: es el número de bytes a mover; tienen que ser de tipo *long*.

origen : es la posición desde la que se cuenta el número de bytes a mover. Puede tener tres valores, que son:

- 0 : Cuenta desde el inicio del archivo.
- 1 : Cuenta desde la posición actual del puntero al archivo.
- 2 : Cuenta desde el final del archivo.

Estos tres valores están representados por tres identificadores (macros):

```
0: SEEK-SET
1: SEEK-CUR
2: SEEK-END
```

La función `fseek()` devuelve un valor entero, distinto de cero si se comete un error en su ejecución; cero si no hay error. El prototipo se encuentra en `stdio.h`:

```
int fseek(FILE *pf, long dsplz, int origen);
```

---

### Ejercicio 13.7

Para celebrar las fiestas patronales de un pueblo se celebra una carrera popular de 9 Km. Se establecen las categorías masculina (M) y femenina (F), y por cada una de ellas, senior y veterano. Los nacidos

antes de 1954 son *veteranos* (tanto para hombres como para mujeres) y el resto *seniors*. Según se realizan inscripciones se crea el archivo binario `CARRERA.POP`, de tal forma que el número de dorsal es la posición que ocupa el registro en el archivo. La carrera se celebra; según llegan los corredores se toman los tiempos realizados y los números de dorsales.

Se desea escribir un programa para crear el archivo `CARRERA.POP` y un segundo programa que actualice cada registro, según el número de dorsal, con el tiempo realizado en la carrera.

### Análisis

En una estructura se agrupan los campos necesarios para cada participante: nombre, año de nacimiento, sexo, categoría, tiempo empleado (minutos, segundos), número de dorsal y puesto ocupado. El primer programa abre el archivo en modo binario para escribir los registros correspondientes a los participantes en la posición del número de dorsal. Los números de dorsal se asignan según la categoría, para las mujeres veteranas del 51 al 100; para mujeres senior de 101 al 200. Para hombres veteranos de 251 al 500, y para senior del 501 al 1000.

El programa, en primer lugar inicializa los nombres de los registros del archivo a blancos. Los dorsales se asignan aleatoriamente, comprobando que no estén previamente asignados. El segundo programa abre el archivo en modo modificación, accede a un registro, según dorsal y escribe el tiempo y puesto. Los tipos de datos que se crean para la aplicación, estructura fecha, estructura tiempo, estructura atleta, se incluyen en el archivo `atleta.h`.

```

/* Archivo atleta.h */

typedef struct fecha
{
 int d, m, a;
}FECHA;
typedef struct tiempo
{
 int h, m, s;
}TIEMPO;
struct atleta
{
 char nombre[28];
 FECHA f;
 char sx; /* Sexo */
 char cat; /* Categoría */
 TIEMPO t;
 unsigned int dorsal;
 unsigned short puesto;
};
typedef struct atleta ADTA;
#define desplz(n) (n-1)*sizeof(ADTA)

/* Programa para dar entrada en el archivo de atletas. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include "atleta.h"

void inicializar(FILE*);
void unatleta(ADTA* at, FILE*);
unsigned numdorsal(char s, char cat, FILE* pf);

int main()

```

```

FILE *pf;
ADTA a;
char *archivo= "C:\CARRERA.POL";
randomize();

if ((pf=fopen(archivo, "wb+"))==NULL)
{
 printf("\nError al abrir el archivo %s, fin del proceso.\n");
 return -1;
}
inicializar(pf);

/* Se introducen registros hasta teclear como nombre: FIN */
unatleta(&a,pf);
do {
 fseek(pf,desplz(a.dorsal) ,SEEK-SET);
 fwrite(&a,sizeof(ADTA),1,pf);
 unatleta(&a,pf);
}while (strcmpi(a.nombre, "FIN"));

fclose(pf);
return 0;
}

void unatleta(ADTA* at, FILE*pf)
{
 printf("Nombre: ");gets(at->nombre);
 if (strcmpi(at->nombre,"fin")j)
 {
 printf("Fecha de nacimiento: ");
 scanf("%d %d %d%c",&at->f.d&at->f.m&at->f.a);
 if (at->f.a<1954)
 at->cat = 'V';
 else
 at->cat = 'S';

 printf("Sexo: ");scanf("%c%c",&at->sx);
 at->sx=(char)toupper(at->sx);

 at->t.h = 0; at->t.m = 0; at->t.s = 0;
 at->dorsal = numdorsal(at->sx,at->cat,pf);
 printf("Dorsal asignado: %u\n",at->dorsal);
 }
}

unsigned numdorsal(char s, char cat, FILE* pf)
{
 unsigned base, tope, d;
 ADTA a;

 if (s=='M' && cat=='V')
 {
 base = 251; tope = 500;

 else if (s=='M' && cat=='S')

 base = 501; tope = 1000;
 }
}

```

```

else if (s=='F' && cat=='V')
{
 base = 51; tope = 100;
}
else if (s=='P' && cat=='S')
{
 base = 101; tope = 200;
}
d = (unsigned) random(tope+1-base)+base;

fseek(pf,desplz(d),SEEK_SET);
fread(&a,sizeof(ADTA),1,pf);

if (!(*a.nombre)) /* Cadena nula: está vacío */
 return d;
else
 return numdorsal(s,cat,pf);
}

void inicializar(FILE*pf)
{
 int k;
 ADTA a;

 a.nombre[0] = '\0';

 for (k=1; k<=1000; k++)
 fwrite(&a,sizeof(ADTA),1,pf);
}

/* Programa para dar entrada a los tiempos de los atletas. Primero, dado
un numero de dorsal se visualiza el registro del atleta, a continuación
se introduce los minutos y segundos realizados por el atleta.
*/

#include <stdio.h>
#include <string.h>
#include "atleta.h"

void datosatleta(ADTA at);

int main()

 FILE *pf;
 ADTA a;
 TIEMPO h={0,0,0};
 char *archivo="C:\CARRERA.POL";
 unsigned dorsal=1;

 if ((pf=fopen(archivo, "rb+"))==NULL)

 printf("\nError al abrir el archivo %s, fin del proceso.\n");
 return -1;
 }

 /* El proceso iterativo termina con el dorsal 0 */
 printf("\n Dorsal del atleta: "); scanf("%u",&dorsal);
 for (; dorsal ;)

```

```

 /* Se sitúa el puntero en el registro */
 fseek(pf,desplz(dorsal),SEEK_SET);
 fread(&a,sizeof(ADTA),1,pf);

 if (*a.nombre)
 {
 datosatleta(a);

 printf("\n Tiempo realizado en minutos y segundos: ");
 scanf("%d %d",&h.m,&h.s);
 a.t = h;

 fseek(pf,desplz(dorsal),SEEK_SET);
 fwrite(&a,sizeof(ADTA),1,pf);
 }
 else
 printf("Este dorsal no está registrado.\n");

 printf("\n Dorsal del atleta: "); scanf("%u",&dorsal);
}

fclose(pf);
return 0;

```

```
void datosatleta(ADTA at)
```

```

{
 printf("Nombre :%s\n",at.nombre);
 printf("Fecha de nacimiento:%d-%d-%d:\n",at.f.d,at.f.m,at.f.a);
 printf("Categoría :%c\tDorsal: %u\n",at.cat,at.dorsal);
 if (at.t.m>0)
 printf("Tiempo de carrera :%d min %d seg\n",at.t.m,at.t.s);
}

```

### 13.6.2. Función ftell()

La posición actual del archivo se puede obtener llamando a la función `ftell()` y pasando un puntero al archivo como argumento. La función devuelve la posición como número de bytes (en entero largo: *long int*) desde el inicio del archivo (byte 0).

#### Ejemplo 13.15

*En este ejemplo se puede observar cómo se desplaza el puntero del archivo según se escriben datos en él.*

```

#include <stdio.h>
int main(void)
{
 FILE *pf;
 float x = 123.5;
 pf = fopen("CARTAS.TXT", "w");
 printf("Posición inicial: %ld\n",ftell(pf));/*muestra 0*/
 fprintf(pf,"Caracteres de prueba");
 printf("Posición actual: %ld\n",ftell(pf));/*muestra 20*/
 fwrite(&x,sizeof(float),1,pf);
}

```

```
printf("Posición actual: %ld\n",ftell(pf));/*muestra 24*/
fclose(pf);
return 0;
```

---

Para llamar a la función se pasa como argumento el puntero a FILE. El prototipo se encuentra en `stdio.h`:

```
long int ftell(FILE *pf);
```

### 13.7. DATOS EXTERNOS AL PROGRAMA CON ARGUMENTOS DE `main()`

La línea de comandos o de órdenes es una línea de texto desde la que se puede ejecutar un programa. Por ejemplo, si se ha escrito el programa `matrices.c`, una vez compilado da lugar a `matrices.exe`. Su ejecución desde la línea de órdenes:

```
C:>matrices
```

La línea de órdenes puede ser una fuente de datos al programa, así se podría pasar las dimensiones de la matriz:

```
C:>matrices 4 5
```

Para que un programa C pueda captar datos, información en la línea de órdenes, la función `main()` tiene dos argumentos opcionales: el primero es un argumento entero que contiene el número de parámetros transmitidos al programa (incluyendo el mismo número de programa). El segundo argumento contiene los parámetros transmitidos, en forma de cadenas de caracteres; por lo que el tipo de este argumento es un array de punteros a `char`. Puede haber un tercer argumento que contiene las variables de entorno, definido también como array de punteros a carácter que no se va a utilizar. Un prototipo válido de la función `main()`:

```
int main(int argc, char*argv[]);
```

También puede ser

```
int main(int argc, char**argv);
```

Los nombres de los argumentos pueden cambiarse, tradicionalmente siempre se pone `argc`, `argv`.

---

#### Ejemplo 13.16

*En este ejemplo se escribe un programa que muestra en pantalla los argumentos escritos en la línea de órdenes.*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int i;

 printf("Número de argumentos %d \n\n", argc);
 printf("Argumentos de la línea de ordenes pasados a main:\n\n");

 for (i = 0; i < argc; i++)
 printf(" argv[%d]: %s\n\n", i, argv[i]);

 return 0;
```

---

En el supuesto que el nombre del programa ejecutable sea ARGMTOS .EXE, y que esté en la unidad de disco C:, la ejecución se realiza con esta instrucción:

```
C:\ARGMTOS Buenas palabras "el amigo agradece" 6 7 Adios.
```

Los argumentos se separan por un blanco. Para que el blanco forme parte del argumento se debe de encerrar entre dobles comillas. La salida de la ejecución de ARGMTOS (ARGMTOS .EXE):

```
Numero de argumentos 7
```

```
Argumentos de la línea de ordenes pasados a main:
```

```
argv[0]: C:\ARGMTOS.EXE
argv[1]: Buenas
argv[2]: palabras
argv[3]: el amigo agradece
argv[4]: 6
argv[5]: 7
argv[6]: Adios.
```

---

### Ejercicio 13.8

*Se desea escribir un programa para concatenar archivos. Los nombres de los archivos han de estar en la línea de órdenes, el nuevo archivo resultante de la concatenación ha de ser el último argumento de la línea de órdenes.*

#### Análisis

El número mínimo de argumentos de la línea de Órdenes ha de ser 3, nombre del programa ejecutable, primer archivo, segundo archivo, etc. y el archivo nuevo. El programa debe de comprobar este hecho. Para copiar un archivo se utiliza la función `fgets()` que lee una línea del archivo de entrada, y la función `fputs()` que escribe la línea en el archivo de salida. En una función, `copia-archivo()`, se realiza la operación de copia, que se llamará tantas veces como archivos de entrada se introduzcan desde la línea de órdenes.

```
#include <stdio.h>
#define MAX-LIN 120
void copia_archivo(FILE*, FILE*);
int main (int argc, char *argv[])
{
 FILE *pfe, *pfw;
 int i;

 if (argc < 3)
 {
 puts("Error en la línea de ordenes, archivos insuficientes.");
 return -2;
 }
 /* El último archivo es donde se realiza la concatenación */
 if ((pfw = fopen(argv[argc-1], "w")) == NULL)
 {
 printf ("Error al abrir el archivo %s ", argv[argc-1]);
 return -3;
 }
 for (i=1; i<argc-1; i++)
 {
 if ((pfe = fopen(argv[i], "r")) == NULL)
```

```

 {
 printf ("Error al abrir el archivo %s ",argv[i]);
 return -1;
 }
 copia_archivo(pfe,pfw);
 fclose(pfe);
}

fclose(pfw);
return 0;
}

/* Función copia un fichero en otro fichero */
/* utiliza fputs() y fgets() */

void copia_archivo(FILE*f1,FILE* f2)
{
 char cad[MAX_LIN];
 while (!feof(f1))
 {
 fgets(cad, MAX-LIN, f1);
 if (!feof(f1)) fputs(cad, f2);
 }
}

```

---

## 13.8. RESUMEN

Este capítulo explora las operaciones fundamentales de la biblioteca estándar de entrada y salida para el tratamiento y manejo de archivos externos.

El lenguaje C, además de las funciones básicas de E/S, contiene un conjunto completo de funciones de manipulación de archivos, de tipos y macros, que se encuentran en el archivo `stdio.h`. Estas funciones se identifican porque empiezan todas por `f` de file, excepto aquellas que proceden de versiones anteriores de C. Las funciones más utilizadas:

- `fopen()` y `fclose()` abren o cierran el archivo.
- `fputc()`, `fgetc()` para acceder al archivo carácter a carácter (byte a byte).
- `fputs()`, `fgets()` para acceder al archivo de caracteres línea a línea.
- `fread()` y `fwrite()` para leer y escribir por bloques, generalmente por registros.
- `ftell()` y `fseek()` para desplazar el puntero a una posición dada (en bytes).

Con estas funciones y otras que están disponibles se puede hacer cualquier tratamiento de un archivo,

modo texto o modo binario; modo secuencial o modo directo.

Para asociar un archivo externo con un flujo, o también podríamos decir con el nombre interno en el programa (puntero a `FILE`) se utiliza `fopen()`. La función `fclose()` termina la asociación y vuelca el buffer del archivo; los archivos que se han manejado son todos con buffer intermedio para aumentar la efectividad.

Un archivo de texto almacena toda la información en formato carácter. Por ejemplo, los valores numéricos se convierten en caracteres (dígitos) que son la representación numérica. Se indica el modo texto en el segundo argumento de `fopen()`, con una `t`. Las funciones más usuales con los archivos de texto son `fputc()`, `fgetc()`, `fputs()`, `fgets()`, `fscanf()` y `fprintf()`.

Un archivo binario almacena toda la información utilizando la misma representación binaria que la computadora utiliza internamente. Los archivos binarios son más eficientes, no hay conversión entre la representación en la computadora y la representación en el archivo; también ocupan menos espacio. Sin

embargo son menos transportables que los archivos de texto. Se indica el modo binario en el segundo argumento de `fopen()`, con una `b`. Las funciones `fputc()`, `fgetc()` y sobre todo `fread()` y `fwrite()` son las que soportan entrada y salida binaria.

Para proporcionar un acceso aleatorio se dispone de las funciones `ftell()` y `fseek()`. También

hay otras funciones como `fsetpos()` y `fgetpos()`.

Con las funciones expuestas se puede hacer todo tipo de tratamiento de archivos, sobre todo archivos con direccionamiento `hash`, archivos secuenciales indexados.

## 13.9. EJERCICIOS

**13.1.** Escribir las sentencias necesarias para abrir un archivo de caracteres cuyo nombre y acceso se introduce por teclado en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

**13.2.** Señalen los errores del siguiente programa:

```
#include <stdio.h>
int main()
{
 FILE* pf;
 pf = fopen("almacen.dat");

 fputs("Datos de los almacenes
 TIESO", pf);

 fclose(pf);
 return 0;
}
```

**13.3.** Se tiene un archivo de caracteres de nombre "SALAS.DAT". Escribir un programa para crear el archivo "SALAS.BIN" con el contenido del primer archivo pero en modo binario.

**13.4.** La función `rewind()` sitúa el puntero del archivo en el inicio del archivo. Escribir una sentencia, con la función `fseek()` que realice el mismo cometido.

**13.5.** Utiliza los argumentos de la función `main()` para dar entrada a dos cadenas; la primera representa una máscara, la segunda el nombre de un archivo de caracteres. El programa tiene que localizar las veces que ocurre la máscara en el archivo.

**13.6.** Las funciones `fgetpos()` y `fsetpos()` devuelven la posición actual del puntero del archivo, y establecen el puntero en una posición dada. Escribir las funciones `pos-actual()` y `mover_pos()`, con los prototipos:

```
int pos-actual (FILE*pf, long* p);
int movergos (FILE*pf, const long p);
```

La primera función devuelve en `p` la posición actual del archivo. La segunda función establece el puntero del archivo en la posición `P`.

**13.7.** Un archivo contiene enteros positivos y negativos. Utiliza la función `fscanf()` para leer el archivo y determinar el número de enteros negativos.

**13.8.** Un archivo de caracteres quiere escribirse en la pantalla. Escribir un programa para escribir el archivo, cuyo nombre viene dado en la línea de órdenes, en pantalla.

**13.9.** Escribir una función que devuelva una cadena de caracteres de longitud `n`, del archivo cuyo puntero se pasa como argumento. La función termina cuando se han leído los `n` caracteres o es fin de archivo. Utilizar la función `fgetc()`.

El prototipo de la función solicitada:

```
char* leer-cadena(FILE* pf, int n);
```

**13.10.** Se quiere concatenar archivos de texto en un nuevo archivo. La separación entre archivo y

archivo ha de ser una línea con el nombre del archivo que se acaba de procesar. Escribir el programa correspondiente de tal forma que los nombres de los archivos se encuentren en la línea de órdenes.

- 13.11. Escribir una función que tenga como argumentos: un puntero de un archivo de texto, un

número de línea inicial y otro número de línea final. La función debe de mostrar las líneas del archivo comprendidas entre los límites indicados.

- 13.12. Escribir un programa que escriba por pantalla las líneas de texto de un archivo, numerando cada línea del mismo.

### 13.10. PROBLEMAS

- 13.1. Escribir un programa que compare dos archivos de texto. El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.
- 13.2. Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones.
- 13.3. Se desea obtener una estadística de un archivo de caracteres. Escribir un programa para contar el número de palabras de que consta un archivo, así como una estadística de cada longitud de palabra.
- 13.4. En un archivo binario se encuentran pares de valores que representan la intensidad en miliamperios y el correspondiente voltaje en voltios para un diodo. Por ejemplo:

|     |      |
|-----|------|
| 0.5 | 0.35 |
| 1.0 | 0.45 |
| 2.0 | 0.55 |
| 2.5 | 0.58 |
| .   | .    |
| .   | .    |

Nuestro problema es que dado un valor del voltaje  $v$ , comprendido entre el mínimo valor y el máximo encontrar el correspondiente valor de la intensidad. Para ello el programa debe leer el archivo, formar una tabla y aplicar un método de interpolación, por ejemplo el método de polinomios de Lagrange. Una vez calculada la intensidad, el programa debe de escribir el par de valores en el archivo.

- 13.5. Un profesor tiene 30 estudiantes y cada estudiante tiene tres calificaciones en el primer parcial. Almacenar los datos en un archivo, dejando espacio para dos notas más y la nota final. Incluir un menú de opciones, para añadir más estudiantes, visualizar datos de un estudiante, introducir nuevas notas y calcular nota final.
- 13.6. Se desea escribir una carta de felicitación navideña a los empleados de un centro sanitario. El texto de la carta se encuentra en el archivo CARTA.TXT. El nombre y dirección de los empleados se encuentra en el archivo binario EMPLA.DAT, como una secuencia de registros con los campos nombre, dirección, etc. Escribir un programa que genere un archivo de texto por cada empleado, la primera línea contienen el nombre, la segunda está en blanco, la tercera la dirección y en la quinta empieza el texto CARTA.TXT.

- 13.7.** Se desea crear un archivo binario formado por registros que representan productos de perfumería. Los campos de cada registro son código de producto, descripción, precio y número de unidades. La dirección de cada registro viene dada por una función hash que toma como campo clave el código del producto (tres dígitos):

$$\text{hash}(\text{clave}) = (\text{clave} \bmod 97) + 1$$

El número máximo de productos distintos es 100. Las colisiones, de producirse, se situarán secuencialmente a partir del registro número 120.

- 13.8.** Escribir un programa para listar el contenido de un determinado subdirectorio, pasado como parámetro a la función `main()`.
- 13.9.** Modificar el Problema 13.2 para añadir un menú con opciones de añadir al archivo nuevos entrenamientos, obtener el tiempo que se está por encima del umbral aeróbico (data pedido por teclado) para un día determinado y media de las pulsaciones.
- 13.10.** Un archivo de texto consta en cada línea de dos cadenas de enteros separadas por el operador `+`, `-`, `.`. Se desea formar un archivo binario con los resultados de la operación que se encuentra en el archivo de texto.

# CAPÍTULO 14

## LISTAS ENLAZADAS

---

### CONTENIDO

- |                                                     |                                 |
|-----------------------------------------------------|---------------------------------|
| <b>14.1.</b> Fundamentos teóricos.                  | <b>14.5.</b> Listas circulares. |
| <b>14.2.</b> Clasificación de las listas enlazadas. | <b>14.6.</b> <i>Resumen.</i>    |
| <b>14.3.</b> Operaciones en listas enlazadas.       | <b>14.7.</b> <i>Ejercicios.</i> |
| <b>14.4.</b> Listas doblemente enlazadas.           | <b>14.8.</b> <i>Problemas.</i>  |

---

## INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la **lista enlazada** (*ligadao encadenada*, (*linkedlist*)) que es una colección de elementos (denominados *nodos*) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un «enlace» o ((puntero)). Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación.

---

## CONCEPTOS CLAVE

- Búsqueda de un nodo en una lista enlazada.
- Lista doblemente enlazada.
- Estructura de una lista enlazada.
- Operaciones en listas enlazadas.
- Eliminación de nodos en una lista enlazada.
- Recorrido de una lista.
- Fundamentos teóricos de listas enlazadas.
- Variables puntero y variables apuntadas.
- Lista circular.

## 14.1. FUNDAMENTOS TEÓRICOS

En capítulos anteriores se han estudiado estructuras lineales de elementos homogéneos (listas, tablas, vectores) y se utilizaban *arrays* para implementar tales estructuras. Esta técnica obliga a fijar por adelantado el espacio a ocupar en memoria, de modo que cuando se desea añadir un nuevo elemento que rebase el tamaño prefijado del array, no es posible realizar la operación sin que se produzca un error en tiempo de ejecución. Ello se debe a que los arrays hacen un uso ineficiente de la memoria. Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la tabla. Para ello se recurre a los **punteros** (*apuntadores*) que hacen un uso más eficiente de la memoria como ya se ha visto con anterioridad.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un «enlace» o «puntero». La idea básica consiste en construir una lista cuyos elementos llamados **nodos** se componen de dos partes o *campos*: la primera parte o campo contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato*, *TipoElemento*, *Info*, etc.) y la segunda parte o *campo* es un puntero (denominado *enlace* o *sgte*) que apunta al siguiente elemento de la lista.



Figura 14.1. Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja (un rectángulo) con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato y en la segunda sección, el enlace o puntero mediante una flecha que sale de la caja y apunta al nodo siguiente.



Una **lista enlazada** consta de un número de elementos y cada elemento tiene dos componentes (*campos*), un puntero al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo.

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. En la Figura 14.2 los nodos forman una secuencia desde el primer elemento ( $e_1$ ) al último elemento ( $e_n$ ). El primer nodo se enlaza al segundo nodo, el segundo nodo se enlaza al tercero y así sucesivamente hasta llegar al último nodo. El nodo último ha de ser representado de forma diferente para significar que este nodo no se enlaza a ningún otro. La Figura 14.3 muestra diferentes representaciones gráficas que se utilizan para dibujar el campo enlace del último nodo.

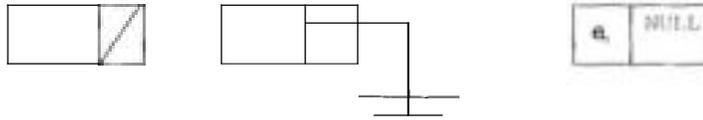


Figura 14.3. Diferentes representaciones gráficas del nodo Último

## 14.2. CLASIFICACIÓN DE LAS LISTAS ENLAZADAS

Las listas se pueden dividir en cuatro categorías :

- *Listas simplemente enlazadas.* Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos («adelante»).
- *Listas doblemente enlazadas.* Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo («adelante») como en recorrido inverso («atrás»).
- *Lista circular simplemente enlazada.* Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular («en anillo»).
- *Lista circular dohlenzente enlazada.* Una lista doblemente enlazada en la **que** el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa («adelante») como inversa («atrás»).

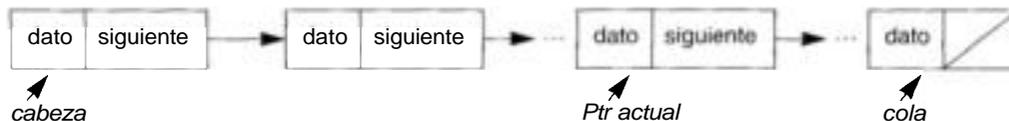
Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arrays o una implementación basada en punteros. Como ya se ha comentado estas implementaciones difieren en el modo en que asigna la memoria para los datos de los elementos, cómo se enlazan juntos los elementos y cómo se accede a dichos elementos. De forma más específica, las implementaciones pueden hacerse con cualquiera de éstas:

- *Asignación fija, o estática, de memoria* mediante array.
- *Asignación dinámica de memoria* mediante punteros.

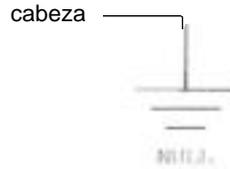
Dado que la asignación fija de memoria mediante arrays es más ineficiente, utilizaremos en este capítulo y siguientes, la asignación de memoria mediante punteros, dejando como ejercicio al lector la implementación mediante arrays.

### Conceptos básicos sobre listas

Una lista enlazada consta de un conjunto de nodos. Un nodo consta de un campo dato y un puntero que apunta al «siguiente» elemento de la lista.



El primer nodo, **frente**, es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo puntero tiene el valor `NULL = 0`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición actual se referencia por el puntero `Ptr_actual`. En el caso en que la lista no contiene ningún nodo (está vacía), el puntero cabeza es nulo.



### 14.3. OPERACIONES EN LISTAS ENLAZADAS

Una lista enlazada requiere unos controles para la gestión de los elementos contenidos en ellas. Estos controles se manifiestan en forma de operaciones que tendrán las siguientes funciones:

- *Declaración de los tipos nodo y puntero a nodo.*
- *Inicialización o creación.*
- *Insertar elementos en una lista.*
- *Eliminar elementos de una lista.*
- *Buscar elementos de una lista* (comprobar la existencia de elementos en una lista).
- *Recorrer una lista enlazada* (visitar cada nodo de la lista).
- *Comprobar si la lista está vacía.*

#### 14.3.1. Declaración de un nodo

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. Cada nodo es una combinación de dos partes: un tipo de dato (entero, real, doble, carácter o tipo predefinido) y un enlace (puntero) al siguiente nodo. En C, se puede declarar un nuevo tipo de dato por un nodo mediante las palabras reservadas `struct` que contiene las dos partes citadas.

```

struct Nodo
{
 int dato;
 struct Nodo* enlace;
};

typedef struct Nodo
{
 int dato;
 struct Nodo "enlace;
}NODO;

```

La declaración utiliza el tipo `struct` que permite agrupar campos de diferentes tipos, el campo `dato` y el campo `enlace`. Con `typedef` se puede declarar a la vez un nuevo identificador de `struct` `Nodo`, en el caso anterior se ha elegido `NODO`.

Dado que los tipos de datos que se puede incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o incluso cadenas), con el objeto de que el tipo de dato de cada nodo se pueda cambiar con facilidad, se suele utilizar una sentencia `typedef` para declarar el nombre de `Elemento` como un sinónimo del tipo de dato de cada campo. El tipo `Elemento` se utiliza entonces dentro de la estructura `nodo`, como se muestra a continuación:

```

typedef double Elemento;
struct nodo
{
 Elemento dato;
};

```

```

 struct nodo *enlace;
};

```

Entonces, si se necesita cambiar el tipo de elemento en los nodos, sólo tendrá que cambiar la sentencia de declaración de tipos que afecta a `Elemento`. Siempre que una función necesite referirse al tipo del dato del nodo, puede utilizar el nombre `Elemento`.

### Ejemplo 14.1

En este ejemplo se declara un tipo denominado `PUNTO` que representa un punto en el plano con su coordenada  $x$  e  $y$ . También se declara el tipo `NODO` con el campo `dato` del tipo `PUNTO`. Por último, se define un puntero a `NODO`.

```

#include <stdlib.h>

typedef struct punto
{
 float x, y;
} PUNTO;

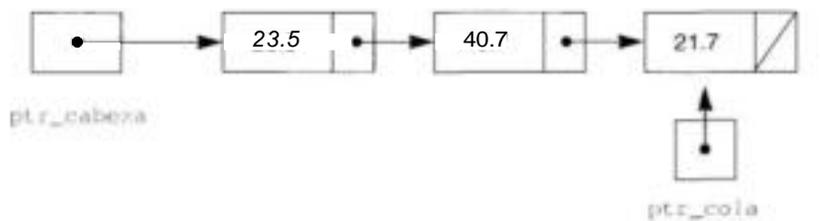
typedef struct Nodo
{
 PUNTO dato;
 struct Nodo* enlace;
} NODO;

NODO* cabecera;
cabecera = NULL;

```

### 14.3.2. Puntero de cabecera y cola

Normalmente, los programas no declaran realmente variables de nodos. En su lugar, cuando se construye y manipula una lista enlazada, a la lista se accede a través de uno o más *punteros* a los nodos. El acceso más frecuente a una lista enlazada es a través del primer nodo de la lista que se llama **cabeza** o **cabecera** de la lista. Un puntero al primer nodo se llama **puntero cabeza**. En ocasiones, se mantiene también un puntero al último nodo de una lista enlazada. El último nodo es la **cola** de la lista, y un puntero al último nodo es el **puntero cola**. También se pueden mantener punteros otros nodos de una lista enlazada.



Declaración del nodo

```

typedef double elemento;
struct nodo
{
 elemento dato;
 struct nodo *enlace;
};

```

Definición de punteros

```

struct nodo *ptr_cabeza;

struct nodo *ptr_cola;

```

Figura 14.4. Declaraciones de tipo en lista enlazada

Cada puntero a un nodo debe ser declarado como una variable puntero. Por ejemplo, si se mantiene una lista enlazada con un puntero de cabecera y otro de cola, se deben declarar dos variables puntero:

```
struct nodo *ptr_cabeza;
struct nodo *ptr_cola;
```

El tipo `struct nodo` a veces se simplifica utilizando la declaración `typedef`. Así podemos escribir:

```
typedef struct nodo NODO;
typedef struct nodo* ptrnodo;
ptrnodo ptr_cabeza;
ptrnodo ptr_cola;
```

La **construcción y manipulación de una lista enlazada** requiere el acceso a los nodos de la lista a través de uno o **más** punteros a nodos. Normalmente, **un** programa incluye un puntero ai primer nodo (*cabeza*)y un puntero al Último nodo (*cola*).

En cualquier forma, el último elemento de la lista contiene un valor de 0, esto es, un puntero nulo (NULL) que señala el final de la lista.

### 14.3.3. El puntero nulo

La Figura 14.4 muestra una lista con un puntero cabeza y un puntero nulo al final de la lista sobre el que se ha escrito la palabra `NULL`. La palabra `NULL` representa el **puntero nulo**, que es una constante especial de C. Se puede utilizar el puntero nulo para cualquier valor de puntero que no apunte a ningún sitio. El puntero nulo se utiliza, normalmente, en dos situaciones:

- Usar el puntero nulo en el campo enlace o siguiente del nodo final de una lista enlazada.
- Cuando una lista enlazada no tiene ningún nodo, se utiliza el puntero `NULL` como puntero de cabeza y de cola. Tal lista se denomina **lista vacía**.

En un programa, el puntero nulo se puede escribir como `NULL`, que es una constante de la biblioteca estándar `stdlib.h`<sup>1</sup>. El puntero nulo se puede asignar a una variable puntero con una sentencia de asignación ordinaria. Por ejemplo:

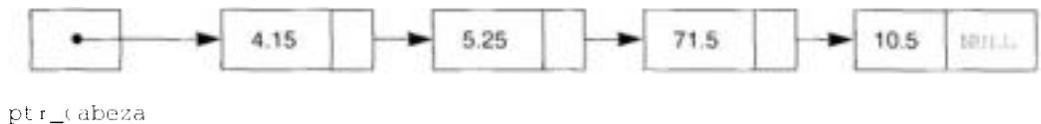


Figura 14.5. Puntero NULL

El puntero de cabeza y de cola en una lista enlazada puede ser `NULL`, lo que indicará que la lista es vacía (no tiene nodos). Éste suele ser un método usual para construir una lista. Cualquier función que se escribe para manipular listas enlazadas debe poder manejar **un** puntero de cabeza y un puntero de cola nulos.

<sup>1</sup> A veces algunos programadores escriben el puntero nulo como 0, pero pensamos es un **estilo** más claro escribirlo como `NULL`.

### 14.3.4. El operador `->` de selección de un miembro

Si `p` es un puntero a una estructura y `m` es un miembro de la estructura, entonces `p -> m` accede al miembro `m` de la estructura apuntada por `p`.

El símbolo “`->`” se considera como un operador simple (en vez de compuesto, al constar de dos símbolos independientes “`-`” y “`>`”). Se denomina *operador de selección de miembro* o también *operador de selección de componente*. De modo visual el operador `p -> m` recuerda a una flecha que apunta del puntero `p` al objeto que contiene al miembro `m`.

Suponiendo que un programa ha de construir una lista enlazada y crear un puntero de cabecera `ptr_cabeza` a un nodo `Nodo`, el operador `*` de indirección aplicado a una variable puntero representa el contenido del nodo apuntado por `ptr_cabeza`. Es decir, `*ptr_cabeza` es un tipo de dato `Nodo`.

Al igual que con cualquier objeto, se puede acceder a los dos miembros de `*ptr_cabeza` en la Figura 14.5. Por ejemplo, la sentencia siguiente escribe los datos del nodo cabecera.

```
printf("%lf", (*ptr_cabeza).dato);
(*ptr_cabeza) miembro dato del nodo apuntado por ptr_cabeza
```

#### Precaución

Los paréntesis son necesarios alrededor de la primera parte de la expresión `(*ptr_cabeza)` ya que los operadores unitarios que aparecen a la derecha tienen prioridad más alta que los operadores unitarios que aparecen en el lado izquierdo (el asterisco de indirección).

Sin los paréntesis, el significado de `ptr_cabeza` producirá un error de sintaxis, al intentar evaluar `ptr_cabeza.dato` antes de la indirección o desreferencia.

`P -> m` significa lo mismo que `(*p).m`

Utilizando el operador de selección `->` se pueden imprimir los datos del primer nodo de la lista

```
printf("%lf", ptr_cabeza->dato);
```

#### Error

Uno de los errores típicos en el tratamiento de punteros es escribir la expresión `*p` o bien `p->` cuando el valor del puntero `p` es el puntero nulo, ya que como se sabe el puntero nulo no apunta a nada.

### 14.3.5. Construcción de una lista

Un algoritmo para la creación de una lista enlazada entraña los siguientes pasos:

- Paso 1. Declarar el tipo de dato y el puntero de cabeza o primero.
- Paso 2. Asignar memoria para un elemento del tipo definido anteriormente utilizando alguna de las funciones de asignación de memoria (`malloc()`, `calloc()`, `realloc()`) y un `cast` para la conversión de `void*` al tipo puntero a nodo; la dirección del nuevo elemento es `ptr_nuevo`.

*Paso 3.* Crear iterativamente el primer elemento (cabeza) y los elementos sucesivos de una lista enlazada simplemente.

*Paso 4.* Repetir hasta que no haya más entrada para el elemento.

### Ejemplo 14.2

*Crear una lista enlazada de elementos que almacenen datos de tipo entero.*

Un elemento de la lista se puede definir con la ayuda de la estructura siguiente:

```
struct Elemento
{
 int dato;
 struct Elemento * siguiente;
};
typedef struct Elemento Nodo;
```

En la estructura `Elemento` hay dos miembros, `dato` y `siguiente` que es un puntero al siguiente nodo y `dato` que contiene el valor del elemento de la lista. También se declara un nuevo tipo: `Nodo` que es sinónimo de `struct Elemento`. El siguiente paso para construir la lista es declarar la variable `Primero` que apuntará al primer elemento de la lista:

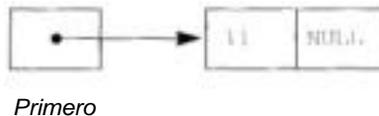
```
Nodo Primero = NULL /* o bien = 0 */
```

El puntero `Primero` (también se puede llamar `Cabeza`) se ha inicializado a un valor nulo, lo que implica que la lista está vacía (no tiene elementos). Ahora se crea un elemento de la lista, para ello hay que reservar memoria, tanta como tamaño tiene cada nodo, y asignar la dirección de la memoria reservada al puntero `Primero`:

```
Primero = (Nodo*)malloc(sizeof(Nodo));
```

Con el operador `sizeof` se obtiene el tamaño de cada nodo de la lista, la función `malloc()` devuelve un puntero genérico (`void*`), por lo que se convierte a `Nodo*`. Ahora se puede asignar un valor al campo `dato`:

```
Primero -> dato = 11;
Primero -> siguiente = NULL;
```



El puntero `Primero` apunta al nuevo elemento, que se inicializa a 11. El campo `siguiente` del nuevo elemento toma el valor nulo, por no haber un nodo siguiente. La operación de *crear un nodo* se puede hacer en una función a la que se pasa el valor del campo `dato` y del campo `siguiente`. La función devuelve un puntero al nodo creado:

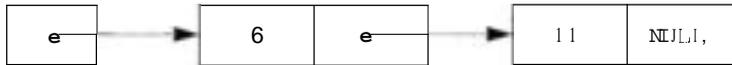
```
Nodo* Crearnodo(int x, Nodo* enlace)
{
 Nodo *p;
 p = (Nodo*)malloc(sizeof(Nodo));
 p->dato = x;
 p->siguiente = enlace;
 return p;
}
```

La llamada a la función `Crearnodo()` para crear el primer nodo de la lista:

```
Primero = Crearnodo(11, NULL);
```

Si ahora se desea añadir un nuevo elemento con un valor 6, y situarlo en el primer lugar de la lista se escribe simplemente:

```
Primero = Crearnodo(6,Primero);
```



Por último para obtener una lista compuesta de 4, 6, 11 se habría de ejecutar:

```
Primero = Crearnodo(4,Primero);
```



### 14.3.6. Insertar un elemento en una lista

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- o En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- o Antes de un elemento especificado.
- o Después de un elemento especificado.

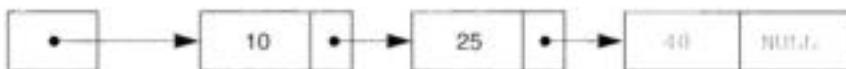
#### Insertar un nuevo elemento en la cabeza de una lista

Aunque normalmente se insertan nuevos datos al final de una estructura de datos, es más fácil y más eficiente insertar un elemento nuevo en la cabeza de una lista. El proceso de inserción se puede resumir en este algoritmo:

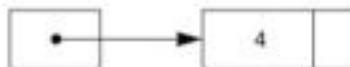
1. Asignar un nuevo nodo apuntado por `nuevo` que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.
2. Situar el nuevo elemento en el campo `dato (Info)` del nuevo nodo.
3. Hacer que el campo `enlace siguiente` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original.
4. Hacer que `cabeza` (puntero cabeza) apunte al nuevo nodo que se ha creado.

#### Ejemplo 14.3

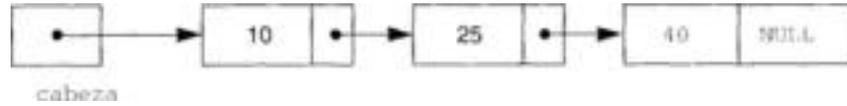
Una lista enlazada contiene tres elementos, 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista.



Pasos 1 y 2



INFO



**Código C**

```

typedef int Item;
typedef struct tipo-nodo
{
 Item dato;
 struct tipo-nodo* siguiente;
} Nodo; /* declaración del tipo Nodo */

Nodo* nuevo;
nuevo = (Nodo*)malloc(sizeof(Nodo)); /* se asigna un nuevo nodo */
nuevo -> dato = entrada;

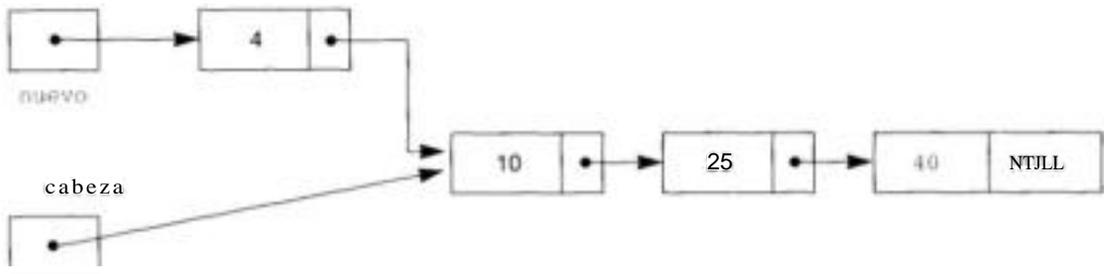
```

**Paso 3**

El campo enlace (siguiente) del nuevo nodo apunta a la cabeza actual de la lista

**Código C**

```
nuevo -> siguiente = cabeza;
```

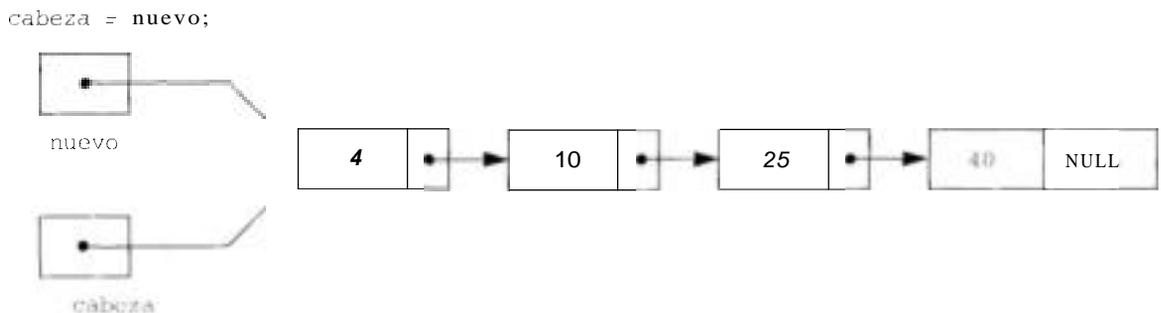


**Paso 4**

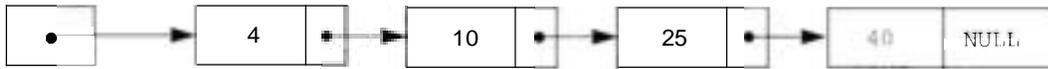
Se cambia el puntero de cabeza para apuntar al nuevo nodo creado: es decir, el puntero de cabeza apunta al mismo sitio que apunte nuevo

**Código C**

```
cabeza = nuevo;
```



En este momento, la función de insertar un elemento en la lista termina su ejecución y la variable local `nuevo` desaparece y sólo permanece el puntero de cabeza `cabeza` que apunta a la nueva lista enlazada



El código fuente de la función `InsertarCabezaLista`:

```

void InsertarCabezaLista(Nodo** cabeza, Item entrada)

 Nodo *nuevo ;
 nuevo = (Nodo*)malloc(sizeof(Nodo)); /* asigna nuevo nodo */
 nuevo -> dato = entrada; /* pone elemento en nuevo */
 nuevo -> siguiente = *cabeza; /* enlaza nuevo nodo al frente de
 la lista */

 cabeza = nuevo; / mueve puntero cabeza y apunta
 al nuevo nodo */

```

### Casoparticular

La función `InsertarCabezaLista` actúa también correctamente si se trata el caso de añadir un primer nodo o elemento a una lista vacía. En este caso, y como ya se ha comentado `cabeza` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

### Ejercicio 14.1

Crear una lista de números aleatorios e insertar **los nuevos nodos por la cabeza de la lista**. Un vez creada la lista, se ha de recorrer **los nodos para mostrar los números pares**.

### Análisis

La función `InsertarCabezaLista()` añade un nodo a la lista, siempre como nodo cabeza. El primer argumento es un puntero a puntero porque tiene que modificar la variable `cabeza`, que es a su vez un puntero a `Nodo`. La función `NuevoNodo()` reserva memoria para un nodo, asigna el campo `dato` y devuelve la dirección del nodo creado.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 99
typedef int item;
typedef struct Elemento
{
 Item dato;
 struct Elemento* siguiente;
}Nodo;

void InsertarCabezaLista(Nodo** cabeza, Item entrada);
Nodo* NuevoNodo(Item x);
void main()

```

```

{
 Item d;
 Nodo *cabeza, *ptr;
 int k;

 cabeza = NULL; /* Inicializa cabeza a lista vacía */
 randomize();
 /* El bucle termina cuando se genera el número aleatorio 0 */
 for (d=random(MX); d;)
 {
 InsertarCabezaLista(&cabeza,d);
 d = random(MX);
 }

 /* Ahora se recorre la lista para escribir los pares */
 for (k=0,ptr=cabeza; ptr;)
 {
 if (ptr->dato%2 == 0)
 {
 printf("%d ",ptr->dato);
 k++;
 printf("%c", (k%12?' ':'\n')); /*cada 12 datos salta de línea */
 }
 ptr = ptr->siguiente;
 }
 printf("\n\n");
}

void InsertarCabezaLista (Nodo** cabeza, Item entrada)
{
 Nodo *nuevo ;
 nuevo = NuevoNodo(entrada);
 nuevo -> siguiente = *cabeza; /* enlaza nuevo nodo al
 frente de la lista */
 cabeza = nuevo; / mueve puntero cabeza y apunta al nuevo
 nodo */
}

Nodo* NuevoNodo (Item x)
{
 Nodo *a ;
 a = (Nodo*)malloc(sizeof(Nodo)); /* asigna nuevo nodo */
 a -> dato = x;
 a -> siguiente = NULL;
 return a;
}

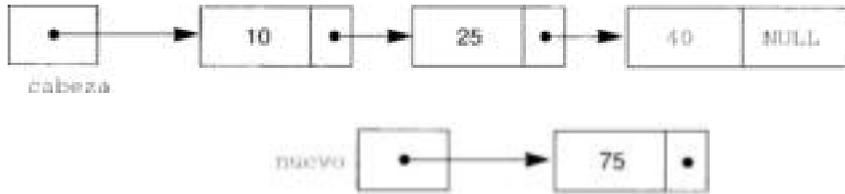
```

### Inserción de un nuevo nodo que no está en la cabeza de lista

La inserción de un nuevo nodo no siempre se realiza al principio (en cabeza) de la lista. Se puede insertar en el centro o al final de la lista.

### Ejemplo 14.4

Se desea insertar un nuevo elemento 75 entre el elemento 25 y el elemento 40 en la lista enlazada [25, 40, ...].

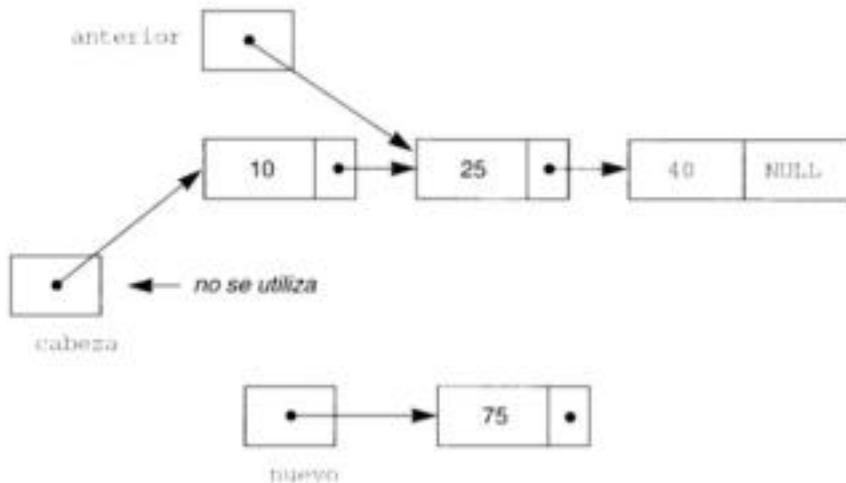


El algoritmo de la nueva operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo apuntado por el puntero nuevo .
2. Situar el nuevo elemento en el campo *dato* (Info) del nuevo nodo.
3. Hacer que el campo enlace siguiente del nuevo nodo apunte al nodo que va después de la posición del nuevo nodo (o bien a NULL si no hay ningún nodo después de la nueva posición).
4. En la variable puntero anterior tener la dirección del nodo que está antes de la posición deseada para el nuevo nodo. Hacer que anterior-> siguiente apunte al nuevo nodo que se acaba de crear.

### Etapas 1 y 2

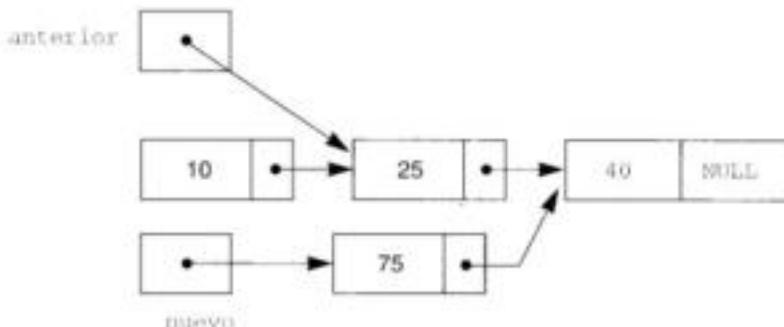
Se crea un nuevo nodo que contiene a 75



### Código C

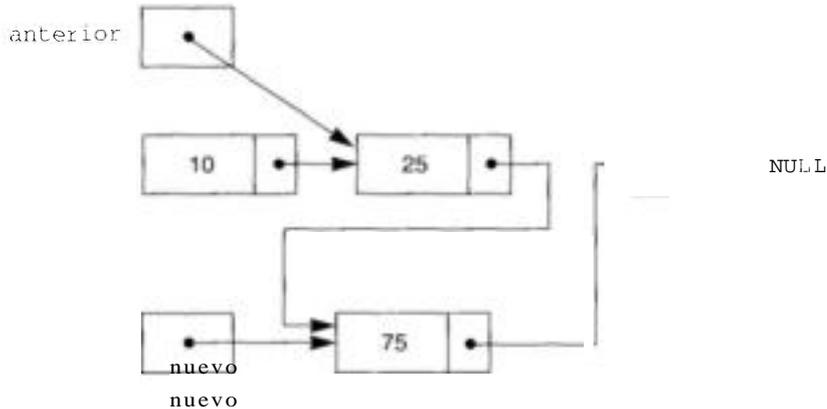
```
nuevo = (Nodo*)malloc (sizeof(Nodo)) ;
nuevo -> dato = entrada ;
```

### Etapas 3



**Código C**

```
nuevo -> siguiente = anterior -> siguiente
```

**Etapas 4**

Después de ejecutar todas las sentencias de las sucesivas etapas, la nueva lista comenzaría en el nodo 10, seguiría 25, 75 y, por último, 40.

**Código C**

```
void InsertarLista(Nodo* anterior, Item entrada)
{
 Nodo *nuevo;

 nuevo = (Nodo*)malloc(sizeof(Nodo));
 nuevo -> dato = entrada;
 nuevo -> siguiente = anterior -> siguiente;
 anterior -> siguiente = nuevo;
}
```

**Inserción al final de la lista**

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último elemento de la lista y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo de la lista y a continuación realizar la inserción. Cuando `ultimo` es una variable puntero que apunta al último nodo de la lista, las sentencias siguientes insertan un nodo al final de la lista.

```
ultimo -> siguiente = (Nodo*)malloc(sizeof(Nodo));
ultimo -> siguiente -> dato = entrada;
ultimo -> siguiente -> siguiente = NULL;
ultimo = ultimo -> siguiente;
```

La primera sentencia asigna un nuevo nodo que está apuntado por el campo `siguiente` al último nodo de la lista (antes de la inserción) de modo que el nuevo nodo ahora es el último nodo de la lista. La segunda sentencia establece el campo `dato` del nuevo último nodo al valor de `entrada`. La tercera sentencia establece el campo `siguiente` del nuevo último nodo a `NULL`. La última sentencia pone la variable `ultimo` al nuevo último nodo de la lista.

### 14.3.7. Búsqueda de un elemento

Dado que una función en C puede devolver un puntero, el algoritmo que sirva para localizar un elemento en una lista enlazada puede devolver un puntero a ese elemento.



La función `BuscarLista` utiliza una variable puntero denominada `indice` que va recorriendo la lista nodo a nodo. Mediante un bucle, `Indice` apunta a los nodos de la lista de modo que si se encuentra el nodo buscado, se devuelve un puntero al nodo buscado con la sentencia de retorno (`return`); en el caso de no encontrarse el nodo buscado la función debe devolver `NULL` (`return NULL`).

#### Código C

```

Nodo* BuscarLista (Nodo* cabeza, Item destino)
/* cabeza: puntero de cabeza de una lista enlazada.
 destino: dato que se busca en la lista.
 indice: valor de retorno, puntero que apunta al primer
 nodo que contiene el destino (elemento buscado);
 si no existe el nodo, se devuelve puntero nulo.
*/
{
 Nodo *indice;

 for (indice = cabeza; indice != NULL; indice = indice ->
 siguiente)

 if (destino == indice -> dato)
 return indice;
 return NULL;
}

```

### Ejemplo 14.5

*En este ejemplo se escribe una función para encontrar la dirección de un nodo dada su posición en una lista enlazada.*

#### Análisis

El nodo o elemento se especifica por su posición en la lista; para ello se considera posición 1, la correspondiente al nodo de cabeza, posición 2, la correspondiente al siguiente nodo, y así sucesivamente. El algoritmo de búsqueda del elemento comienza con el recorrido de la lista mediante un puntero `indice` que comienza apuntando al nodo cabeza de la lista. Un bucle mueve el `indice` hacia adelante el número correcto de sitios (lugares). A cada iteración del bucle se mueve el puntero `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e `indice` apunta al nodo correcto. El bucle también se puede terminar si `indice` apunta a `NULL`, lo que indicará que la posición solicitada era más grande que el número de nodos de la lista.

#### Código C

```

Nodo* BuscarPosicion(Nodo *cabeza, size_t posicion)
/* El programa que llame a esta función ha de incluir
 biblioteca stdlib.h (para implementar tipo size_t)
*/
{
 Nodo *indice;

```

```

size_t i;
if (0 < posicion) /* posición ha de ser mayor que 0 */
 return NULL;
indice = cabeza;
for (i = 1 ; (i < posicion) && (indice != NULL) ; i++)
 indice = indice -> siguiente;
return indice;

```

I

### 14.3.8. Supresión de un nodo en una lista

La operación de eliminar un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo para eliminar un nodo que contiene un dato se puede expresar en estos pasos:

1. Búsqueda del nodo que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior.
2. El puntero `siguiente` del nodo anterior ha de apuntar al siguiente del nodo a eliminar.
3. En caso de que el nodo a eliminar sea el primero, `cabeza`, se modifica `cabeza` para que tenga la dirección del nodo siguiente.
4. Por último, se libera la memoria ocupada por el nodo.

A continuación se escribe una función que recibe la cabeza de la lista y el dato del nodo que se quiere borrar.

```

void eliminar (Nodo** cabeza, item entrada)
{
 Nodo* actual, "anterior;
 int encontrado = 0;

 actual = *cabeza; anterior = NULL;
 /* Bucle de búsqueda */
 while ((actual != NULL) && (!encontrado))
 {
 encontrado = (actual->dato == entrada);
 if (!encontrado)
 {
 anterior = actual;
 actual = actual -> siguiente;
 }
 }
 I
 /* Enlace de nodo anterior con siguiente */
 if (actual != NULL)
 {
 /* Se distingue entre que el nodo sea el cabecera o del
 resto de la lista */
 if (actual == *cabeza)

 "cabeza = actual->siguiente;
 I
 else {
 anterior -> siguiente = actual -> siguiente
 }

 free(actual);
 }
}

```

**Ejercicio 14.2**

Se desea crear una lista enlazada de números enteros ordenada. La lista va estar organizada de tal forma que el nodo cabecera tenga el menor elemento, y así en orden creciente los demás **nod**os. Una vez creada la lista, se recorre para escribir los datos por pantalla.

**Análisis**

La función `InsertaOrden()` añade los nuevos elementos. Inicialmente la lista se crea con el primer valor. El segundo elemento se ha de insertar antes del primero o después, dependiendo de que sea menor o mayor. Así, en general, para insertar un nuevo elemento, primero se busca la posición de inserción en la lista actual, que en todo momento está ordenada, del nodo a partir del cual se ha de enlazar el nuevo nodo para que la lista siga ordenada. La función `recorrer()` avanza por cada uno de los nodos de la lista con la finalidad de escribir el campo dato.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 101
typedef int Item;
typedef struct Elemento
{
 Item dato;
 struct Elemento* siguiente;
}Nodo;

void InsertaOrden(Nodo** cabeza, ltcn entrada);
Nodo* NuevoNodo(Item x);
void recorrer(Nodo* cabeza);

void main()
{
 Item d;
 Nodo* cabeza;

 cabeza = NULL; /* Inicia lza cabeza a lista vacía */
 randomize();
 /* El bucle termina cuando se genera el número aleatorio 0 */
 for (d=random(MX); d;)
 {
 InsertaOrden(&cabeza,d);
 d = random(MX);
 }

 recorrer(cabeza);
}

void InsertaOrden(Nodo** cabeza, Item entrada)
{
 Nodo *nuevo;

 nuevo = NuevoNodo(entrada);

 if ("cabeza == NULL,")
 *cabeza = nuevo;
 else if (entrada < (*cabeza)->dato)
 {
 nuevo -> siguiente = *cabeza;
 *cabeza = nuevo;
 }
}
```

```

 }
 else /* búsqueda del nodo anterior a partir del que
 se debe insertar */
 {
 Nodo* anterior, *p;
 anterior = p = "cabeza;

 while ((p->siguiente!= NULL,) && (entrada > p->dato))
 {
 anterior = p;
 p = p->siguiente;
 }

 if (entrada > p->dato) /* se inserta después del último nodo */
 anterior = p;

 /* Se procede al enlace del nuevo nodo */
 nuevo -> siguiente = anterior -> siguiente;
 anterior -> siguiente = nuevo;
 }

Nodo* NuevoNodo(Item x)
{
 Nodo *a ;
 a = (Nodo*)malloc(sizeof(Nodo)); /* asigna nuevo nodo */
 a -> dato = x; /* pone elemento en nuevo */
 a -> siguiente = NULL;
 return a;
}

void recorrer(Nodo* cabeza)
{
 int k;
 printf("\n\t\t Lista Ordenada \n");
 for (k=0; cabeza; cabeza=cabeza->siguiente)
 {
 printf("%d ",cabeza->dato);
 k++;
 printf("%c", (k%15 ? '\n' : '\n'));
 }
 printf("\n\n");
}

```

---

## 14.4. LISTA DOBLEMENTE ENLAZADA

Hasta ahora el recorrido de una lista se realizaba en sentido directo (*adelante*), en algunos casos, en sentido inverso (*hacia atrás*). Sin embargo, existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden. En este caso se recomienda el uso de una **lista doblemente enlazada**. En tal lista, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento. Un puntero apunta al siguiente elemento de la lista y el otro puntero apunta al elemento anterior. La Figura 14.6 muestra una lista doblemente enlazada y un nodo de dicha lista.

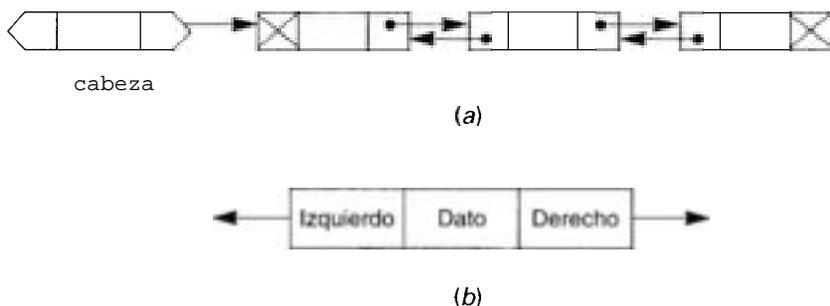


Figura 14.6. Lista doblemente enlazada. (a) Lista con tres nodos; (b) nodo.

Existe una operación de *insertar* y *eliminar* (borrar) en cada dirección. La Figura 14.7 muestra el problema de insertar un nodo *p* a la derecha del nodo actual. Deben asignarse cuatro nuevos enlaces

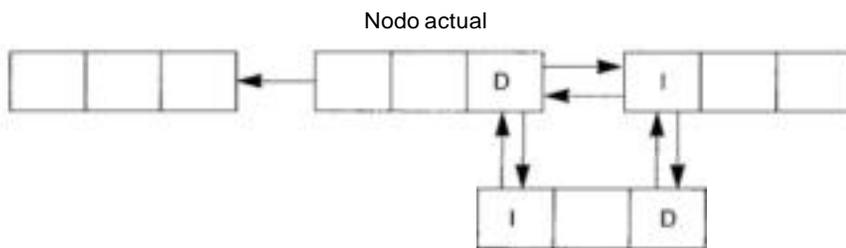


Figura 14.7. Inserción de un nodo en una lista doblemente enlazada

En el caso de eliminar (borrar) un nodo de una lista doblemente enlazada es preciso cambiar dos punteros.

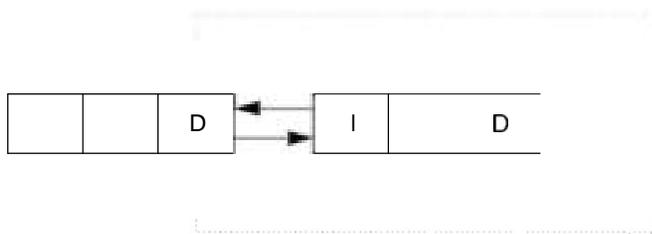


Figura 14.8. Eliminación de un nodo en una lista doblemente enlazada.

### 14.4.1. Declaración de una lista doblemente enlazada

Una lista doblemente enlazada con valores de tipo `int` necesita dos punteros y el valor del campo datos. En una estructura se agrupan estos datos del modo siguiente:

```
typedef int Item;
struct unnodo
{
 Item dato;
 struct unnodo *adelante;
```

```

 struct unnodo *atras;
};
typedef struct unnodo Nodo;

```

#### 14.4.2. Insertar un elemento en una lista doblemente enlazada

El algoritmo empleado para añadir o insertar un elemento en una lista doble varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento Último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

##### Insertar un nuevo elemento en la cabeza de una lista doble

El proceso de inserción se puede resumir en este algoritmo:

1. Asignar un nuevo nodo apuntado por nuevo que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista doble.
2. Situar el nuevo elemento en el campo `dato` (Info) del nuevo nodo.
3. Hacer que el campo enlace adelante del nuevo nodo apunte a la cabeza (primer nodo) de la lista original, y que el campo enlace `atras` del nodo cabeza apunte al nuevo nodo.
4. Hacer que `cabeza` (puntero cabeza) apunte al nuevo nodo que se ha creado.

##### Código C

```

typedef int Item;
typedef struct tipo_nodo
{
 Item dato;
 struct tipo_nodo* adelante;
 struct tipo_nodo* atras;
}Nodo;

Nodo* nuevo;

nuevo = (Nodo*)malloc(sizeof(Nodo));
nuevo -> dato = entrada;
nuevo -> adelante = cabeza;
nuevo -> atras = NULL;
cabeza -> atras = nuevo;
cabeza = nuevo;

```

En este momento, la función de insertar un elemento en la lista termina su ejecución y la variable local `nuevo` desaparece y sólo permanece el puntero de cabeza `cabeza` que apunta a la nueva lista doblemente enlazada.

##### Inserción de un nuevo nodo que no está en la cabeza de lista

La inserción de un nuevo nodo en una lista doblemente enlazada se puede realizar en un nodo intermedio de ella. El algoritmo de la nueva operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo apuntado por el puntero `nuevo`.
2. Situar el nuevo elemento en el campo `dato` (Info) del nuevo nodo.
3. Hacer que el campo enlace adelante del nuevo nodo apunte al nodo que va después de la posición del nuevo nodo (o bien a `NULL` si no hay ningún nodo después de la nueva posición). El campo `atras` del nodo siguiente al nuevo tiene que apuntar a nuevo.

4. La dirección del nodo que está antes de la posición deseada para el nuevo nodo está en la variable puntero anterior. Hacer que anterior->adelante apunte al nuevo nodo. El enlace atras del nuevo nodo debe de apuntar a anterior.

#### Código C

```
nuevo = (Nodo*)malloc (sizeof(Nodo)) ;
nuevo -> dato = entrada ;
nuevo -> adelante = anterior -> adelante;
anterior -> adelante -> atras = nuevo; /* campo atras del siguiente
 apunta al nodo nuevo creado */
anterior -> adelante = nuevo;
nuevo -> atras = anterior;
```

### 14.4.3. Supresión de un elemento en una lista doblemente enlazada

La operación de eliminar un nodo de una lista doble supone realizar el enlace de dos punteros, el nodo anterior con el nodo siguiente al que se desea eliminar con el puntero adelante y el nodo siguiente con el anterior con el puntero atras y liberar la memoria que ocupa.

El algoritmo para eliminar un nodo que contiene un dato es similar al algoritmo de borrado para una lista simple. Ahora la dirección del nodo anterior se encuentra en el puntero atras del nodo a borrar. Los pasos a seguir:

1. Búsqueda del nodo que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior.
2. El puntero adelante del nodo anterior tiene que apuntar al puntero adelante del nodo a eliminar, esto en el caso de no ser el nodo cabecera.
3. El puntero atras del nodo siguiente a borrar tiene que apuntar al puntero atras del nodo a eliminar, esto en el caso de no ser el nodo último.
4. En caso de que el nodo a eliminar sea el primero, cabeza, se modifica cabeza para que tenga la dirección del nodo siguiente.
5. Por último, se libera la memoria ocupada por el nodo.

La codificación se presenta en la siguiente función:

```
void eliminar (Nodo** cabeza, item entrada)

Nodo* actual;
int encontrado = 0;

actual = *cabeza;
/* Bucle de búsqueda */
while ((actual!=NULL) && (!encontrado))

 encontrado = (actual->dato == entrada);
 if (!encontrado)
 actual = actual -> adelante;
}
/* Enlace de nodo anterior con siguiente */
if (actual != NULL)
{
 /* Se distingue entre que el nodo sed el cabecera o del
 resto de la lista */
 if (actual == *cabeza)
 {
 *cabeza = actual->adelante;
```

```

 if (actual->adelante != NULL)
 actual->adelante->atras = NULL;
 }
 else if (actual->adelante != NULL) /* No es el último nodo */
 {
 actual -> atras ->adelante = actual -> adelante;
 actual -> adelante -> atras = actual -> atras;
 }
 else { /* último nodo */
 actual -> atras -> adelante = NULL;

 free(actual);
 }
}

```

### Ejercicio 14.3

*Se va a crear una lista doblemente enlazada con números enteros obtenidos aleatoriamente. Una vez creada la lista se desea eliminarse los nodos que estén fuera de un rango determinado.*

#### Análisis

La inserción de elementos en la lista se hace por el nodo cabecera. El número de elementos de la lista se pide para ser introducido por teclado. También se pide por teclado el rango de valores que deben de estar en la lista. Para eliminar los elementos se recorre la lista, los nodos que no están dentro del rango se borran de la lista. Para borrar los nodos se utiliza la función `eliminar()`, teniendo en cuenta que la dirección del nodo a suprimir ya se tiene.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef int Item;
typedef struct Elemento
{
 Item dato;
 struct Elemento* adelante;
 struct Elemento* atras;
}Nodo;
void InsertarCabezaLista(Nodo** cabeza, Item entrada);
Nodo* NuevoNodo(Item x);
void eliminar(Nodo** cabeza, Nodo* actual);
void recorrer(Nodo* ptr);

void main()
{
 Nodo* cabeza,*ptr;
 int x,y;

 cabeza = NULL; /* Inicializa cabeza a lista vacía */
 randomize();
 printf("\n Número de elementos a generar: ");
 scanf("%d",&x);
 /* Se genera la lista doble */
 for (; x-->0)
 {
 InsertarCabezaLista(&cabeza,rand());
 }
}

```

```

recorrer (cabeza);

printf("\nRango de los valores que va a tener la lista: ");
scanf("%d %d",&x,&y);

/* Recorre la lista para eliminar nodos que no están en
 el rango de valores */
printf("\n\tNodos eliminados\n");
for (ptr=cabeza; ptr;)

 if ((ptr->dato<x) || (ptr->dato>y))
 {
 Nodo* t;
 t = ptr->adelante; /* Guarda el nodo por el que seguir */
 printf("%d ",ptr->dato);
 eliminar(&cabeza,ptr);
 ptr = t;
 }
 else
 ptr = ptr->adelante;
}

/* Recorre la lista para mostrar sus elementos */
recorrer(cabeza);

void eliminar (Nodo** cabeza, Nodo* actual)
{
 /* Elimina el nodo de direccion actual.
 Se distingue entre que el nodo sea el cabecera o del
 resto de la lista.
 */
 if (actual == *cabeza)
 {
 *cabeza = actual->adelante;
 if (actual->adelante != NULL)
 actual->adelante->atras = NULL;
 }
 else if (actual->adelante != NULL) /* No es el último nodo */
 {
 actual->atras->adelante = actual->adelante;
 actual->adelante->atras = actual->atras;

 }
 else { /* Último nodo */
 actual->atras->adelante = NULL;
 }
 free(actual);
}

void recorrer(Nodo* ptr)
{
 int k = 0;

 printf("\n\n\tElementos de la lista\n");

 for (; ptr ;)
 {
 k++;
 printf("%-5d",ptr->dato);
 printf("%c", (k%12==0?' \n':' '));
 }
}

```

```

 ptr = ptr -> adelante;
 }
}

void InsertarCabezaLista (Nodo** cabeza, Item entrada)
{
 Nodo* nuevo;

 nuevo = NuevoNodo(entrada);
 nuevo -> adelante = *cabeza;
 nuevo -> atras = NULL;
 if (*cabeza != NULL)
 (*cabeza) -> atras = nuevo;
 *cabeza = nuevo;
}

Nodo* NuevoNodo(Item x)
{
 Nodo *a ;
 a = (Nodo*)malloc (sizeof(Nodo));
 a -> dato = x;
 a -> adelante = a -> atras = NULL;
 return a;
}

```

## 14.5. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo y un último nodo que tiene el campo de enlace a nulo. Una lista circular, por propia naturaleza *no tiene ni principio ni fin*. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos. La Figura 14.9 muestra una lista circular con enlace simple; podría considerarse como una lista lineal, de tal manera que el último nodo apunta al primero.

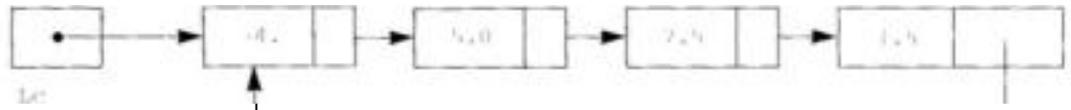


Figura 14.9. Lista circular.

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que el *último* nodo no apunta a nulo sino al *primero*. La creación de una lista circular se puede hacer con un enlace simple o un enlace doble. Consideramos que la lista circular se enlaza con un solo enlace, la realización con enlace *adelante* y *atrás* es similar (se puede consultar el Apartado 14.4).

### 14.5.1. Insertar un elemento en una lista circular

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede variar, consideramos que

se hace como nodo anterior al del nodo de acceso a la lista `Lc`, y que `Lc` tiene la dirección del último nodo insertado. A continuación se escribe la declaración de un nodo, una función que crea un nodo y la función que inserta el nodo en la lista circular.

```
typedef char* Item;
typedef struct Elemento
{
 Item dato;
 struct Elemento* siguiente;
}Nodo;

Nodo* NuevoNodo(Item x
{
 Nodo *a ;
 a = (Nodo*)malloc(sizeof(Nodo));
 a -> dato = x;
 a -> siguiente = u; /* apunta así mismo, es un nodo circular */
 return a;
}

void InsertaCircular(Nodo** lc,Item entrada)
{
 Nodo* nuevo;

 nuevo = NuevoNodo(entrada);
 if (*lc != NULL) /* lista circular no vacía */
 nuevo -> siguiente = (*lc) -> siguiente;
 (*lc) -> siguiente = nuevo;
 }
 *lc = nuevo;
}
```

### 14.5.2. Supresión de un elemento en una lista circular

La operación de eliminar un nodo de una lista circular sigue los mismos pasos que los dados para eliminar un nodo en una lista lineal. Hay que enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo para eliminar un nodo de una lista circular:

1. Búsqueda del nodo que contiene el dato.
2. Se enlaza el nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea el referenciado por el puntero de acceso a la lista, `Lc`, se modifica `Lc` para que tenga la dirección del nodo anterior.
4. Por último, se libera la memoria ocupada por el nodo.

En la función de eliminar hay que tener en cuenta la característica de lista circular, así para detectar si la lista es de un solo nodo ocurre que se apunta a él mismo.

`Lc == Lc->siguiente` si esta expresión es cierta la lista consta de un solo nodo.

A continuación se escribe el código de la función eliminar para una lista circular. Para ello recorre la lista con un puntero al nodo anterior, por esa razón se accede al dato con la sentencia `actual->siguiente->dato`.

Esto permite, en el caso de encontrarse el nodo, tener en `actual` el nodo anterior. Después del bucle es necesario volver a preguntar por el campo `dato`, ya que no se comparó el nodo `Lc` y el bucle puede haber terminado sin encontrar el nodo:

*Código C*

```

void eliminar (Nodo** Lc, Item entrada)
{
 Nodo* actual;
 int encontrado = 0;

 actual = *Lc;
 /* Bucle de búsqueda */
 while ((actual->siguiente != *Lc) && (!encontrado))
 {
 encontrado = (actual->siguiente->dato == entrada);
 if (!encontrado)
 {
 actual = actual -> siguiente;
 }
 }
 encontrado = (actual->siguiente->dato == entrada);

 /* Enlace de nodo anterior con siguiente */
 if (encontrado)
 {
 Nodo* p;
 p = actual->siguiente; /* Nodo a eliminar */
 if (*Lc == (*Lc)->siguiente) /* Lista con un solo nodo */
 *Lc = NULL;
 else {
 if (p == *Lc)
 {
 Lc = actual; / Se borra el elemento referenciado por Lc;
 el nuevo acceso a la lista es el anterior */
 }
 actual->siguiente = p->siguiente;
 }
 free(p);
 }
}

```

**Ejercicio 14.4**

*Este ejercicio crea una lista circular con palabras leídas del teclado. El programa debe tener un conjunto de opciones para:*

- a) Mostrar las cadenas que forman la lista;*
- b) Borrar una palabra dada;*
- c) Al terminar la ejecución, recorrer la lista eliminando los nodos.*

**Análisis**

Los nodos de la lista tienen como campo dato un puntero a una cadena que es la palabra. Desde el teclado se lee la palabra en un buffer suficientemente amplio; se ha de reservar memoria para tantos caracteres como longitud (`strlen()`) tenga la cadena leída y asignar su dirección al puntero del nodo. A continuación se copia el buffer a la memoria reservada (campo dato del nodo). El nodo se inserta llamando a la función `InsertaCircular()`. Para borrar una palabra se llama a la función `eliminar()`.

```

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

```

```

typedef char* Item;
typedef struct Elemento
{
 Item dato;
 struct Elemento* siguiente;
}Nodo;

Nodo* NuevoNodo(Item x);
void InsertaCircular(Nodo** Lc, Item entrada);
void eliminar(Nodo** Lc, Item entrada);
void recorrer(Nodo* Lc);
void borrarlista(Nodo** Lc);

int main()

 char cadena[81];
 Nodo *Lc; int opc;

 Lc = NULL;
 printf("\n\n Entrada de Nombres. Termina con ^Z.\n");
 while (gets(cadena))
 {
 InsertaCircular(&Lc,cadena);
 }
 recorrer(Lc);

 puts("\n\n\t Opciones para manejar la lista");
 do {
 puts("\n 1. Elimar una palabra de la lista circular.\n");
 puts("\n 2. Mostrar todos los elementos de la lista.\n");
 puts("\n 3. Salir y eliminar los nodos de la lista.\n");
 do {
 scanf("%d%c",&opc);
 }while (opc<1 || opc>3);

 switch (opc) {
 case 1: printf("Palabra a eliminar: ");
 gets(cadena);
 eliminar(&Lc,cadena);
 break;
 case 2: printf("\nPalabras que continen la Lista:\n");
 recorrer(Lc);
 break;
 case 3: puts("Eliminación de los nodos de la lista.");
 }
 } while opc != 3);

 return 0;
}

Nodo* NuevoNodo(Item x)
{
 Nodo *a ;
 a = (Nodo*)malloc(sizeof(Nodo));
 /* Se reserva memoria para la cadena */
 a -> dato = (char*)malloc((strlen(x)+1)*sizeof(char));
 strcpy(a->dato,x);
 a -> siguiente = a; /* apunta así mismo, es un nodo circular */
 return a;
}

```

```

}

void InsertaCircular(Nodo** Lc, Item entrada)

Nodo* nuevo;

nuevo = NuevoNodo(entrada);
if (*Lc != NULL) /* lista circular no vacía */
{
 nuevo -> siguiente = (*Lc) -> siguiente;
 (*Lc) -> siguiente = nuevo;
}
*Lc = nuevo;
}

void eliminar (Nodo** Lc, Item entrada)
{
 Nodo* actual;
 int encontrddo = 0;

 actual = *Lc;
 /* Bucle de búsqueda */
 while ((actual->siguiente != *Lc) && (!encontrado))
 {
 encontrado = strcmp(actual->siguiente->dato, entrada)==0;
 if (!encontrado)
 {
 actual = actual -> siguiente;
 }
 }
 encontrddo = strcmp(actual->siguiente->dato, entrada)==0;

 /* Enlace de nodo anterior con siguiente */
 if (encontrddo)
 {
 Nodo* p;
 printf("\nNodo de la palabra \" %s \" encontrado.\n", entrada);
 p = actual->siguiente; /* Nodo a eliminar */
 if (*Lc == (*Lc)->siguiente) /* Lista con un solo nodo */
 *Lc = NULL;
 else {
 if (p == *Lc)
 {
 Lc = actual; / Se borra el elemento referenciado por Lc;
 el nuevo acceso a la lista es; el anterior */
 }
 actual->siguiente = p->siguiente;
 free(p);
 }
 }
}

void recorrer(Nodo* Lc)

Nodo* p;
if (Lc != NULL)
{

```

```

p = Lc->siguiente; /* Lc tiene el último nodo, el siguiente es
 el primero que se insertó */
do {
 printf("\t\t%s",p->dato);
 p = p->siguiente;
}while(p != Lc->siguiente);

else
 printf("\n\t Lista vacía.\n");

void borrarlista(Nodo** Lc)
{
 Nodo* p;

 if (Lc != NULL)
 {
 p = *Lc;
 do {
 Nodo* t;
 t = p; p = p->siguiente;
 free(t);
 }while(p != *Lc);
 }
 else
 printf("\n\t Lista vacía.\n");
 *Lc = NULL;
}

```

## 14.6. RESUMEN

La estructura de datos *lista* se puede implementar, bien como un array, bien como una lista enlazada.

Una **lista enlazada** es una estructura de datos dinámica en la que sus componentes están ordenados lógicamente por sus campos punteros en vez de ordenados físicamente como están en un array. El final de la lista se señala mediante una constante o puntero especial llamado NULL.

La gran ventaja de una lista enlazada sobre un array es que la lista enlazada puede crecer y decrecer en tamaño, ajustándose al número de elementos.

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.

Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro de la lista.

El recorrido de una lista enlazada significa pasar por cada nodo (visitar) y procesarlo. El proceso puede ser escribir su contenido, modificar el campo de datos.

Una **lista doblemente enlazada** es aquella en la que cada nodo tiene un puntero a su sucesor y otro a su predecesor.

Las **listas doblemente enlazadas** se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, borrado y recorrer la lista; similares a las listas simples.

Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.

## 14.7. EJERCICIOS

- 14.1. Escribir una función que devuelva cierto ( $\neq 0$ ) si la lista está vacía.
- 14.2. Escribir una función entera que devuelva el número de nodos de una lista enlazada.
- 14.3. En una lista enlazada de números enteros se desea añadir un nodo entre dos nodos consecutivos con campos dato de distinto signo; el valor del campo dato del nuevo nodo que sea la diferencia en valor absoluto.
- 14.4. Escribir una función que elimine el nodo que ocupa la posición  $i$ , siendo el nodo cabecera el que ocupa la posición 0.
- 14.5. Escribir una función que tenga como argumento el puntero `cabeza` al primer nodo de una lista enlazada. La función debe devolver un puntero a una lista doble con los mismos campos dato pero en orden inverso.
- 14.6. Se tiene una lista simplemente enlazada de números reales. Escribir una función para obtener una lista doble ordenada respecto al campo dato, con los valores reales de la lista simple.
- 14.7. Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. La función debe tener un argumento puntero `Ld` en el que se devuelva la dirección del nodo que está en la posición intermedia.
- 14.8. Se tiene que `Lc` es una lista circular de palabras. Escribir una función que cuente el número de veces que una palabra dada se encuentra en la lista.
- 14.9. Escribir una función entera que tenga como argumento una lista circular de números enteros. La función debe devolver el dato del nodo con mayor valor.
- 14.10. Se tiene una lista de simple enlace, el campo dato es un registro (estructura) con los campos de un alumno: nombre, edad, sexo. Escribir una función para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino el siguiente sea de sexo femenino.
- 14.11. Una lista circular de cadenas está ordenada alfabéticamente. El puntero `Lc` tiene la dirección del nodo alfabéticamente mayor, apunta al nodo alfabéticamente menor. Escribir una función para añadir una nueva palabra, en el orden que le corresponda, a la lista.
- 14.12. Dada la lista del Ejercicio 14.11 escribir una función que elimine una palabra dada.

## 14.8. PROBLEMAS

- 14.1. Escribir un programa o funciones individuales que realicen las siguientes tareas:
- Crear una lista enlazada de números enteros positivos al azar, la inserción se realiza por el último nodo.
  - Recorrer la lista para mostrar los elementos por pantalla.
  - Eliminar todos los nodos que superen un valor dado.
- 14.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

- 14.3. Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene



como campo dato el coeficiente del término y el exponente.

Por ejemplo, el polinomio  $3x^4 - 4x^2 + 11$  se representa

Escribir un programa que permita dar entrada a polinomios en  $x$ , representándolos con una lista enlazada simple. A continuación obtener una tabla de valores del polinomio para valores de  $x = 0.0, 0.5, 1.0, 1.5, \dots, 5.0$ .

- 14.4. Teniendo en cuenta la representación de un polinomio propuesta en el Problema 14.3, hacer los cambios necesarios para que la lista enlazada sea circular. El puntero de acceso debe de tener la dirección del Último término del polinomio, el cual apuntará al primer término.
- 14.5. Según la representación de un polinomio propuesta en el Problema 14.4, escribir un programa para realizar las siguientes operaciones:
- Obtener la lista circular suma de dos polinomios.
  - Obtener el polinomio derivada.
  - Obtener una lista circular que sea el producto de dos polinomios.
- 14.6. Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter.
- Una vez que se tiene la lista ordenarla alfabéticamente y escribirla por pantalla.
- 14.7. Un conjunto es una secuencia de elementos todos del mismo tipo, sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las operaciones:
- Cardinal del conjunto.
  - Pertenencia de un elemento al conjunto.
  - Añadir un elemento al conjunto.
  - Escribir en pantalla los elementos del conjunto.
- 14.8. Con la representación propuesta en el Problema 14.7, añadir las operaciones básicas de conjuntos:
- Unión de dos conjuntos.
  - Intersección de dos conjuntos.
  - Diferencia de dos conjuntos.
  - Inclusión de un conjunto en otro.
- 14.9. Escribir un programa en el que dados dos archivos F1, F2 formados por palabras separadas por un blanco o tin de línea, se creen dos conjuntos con las palabras de F1 y F2, respectivamente. Posteriormente encontrar las palabras comunes y mostrarlas por pantalla.
- 14.10. Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final), eliminar un pasajero de la lista. A la lista se accede por un puntero ai primer nodo y otro al último nodo.
- 14.11. Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular teniendo el campo dato de cada nodo un dígito del entero largo. Escribir un programa en el que se introduzcan dos enteros largos y se obtenga su suma.
- 14.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.

# CAPÍTULO 15

## PILAS Y COLAS

---

### CONTENIDO

- 15.1.** Concepto de pila.
- 15.2.** El tipo pila implementado con arrays.
- 15.3.** Concepto de cola.
- 15.4.** Colas implementadas con arrays.
- 15.5.** Realización de una cola con una *lista* enlazada.
- 15.6.** *Resumen.*
- 15.7.** *Ejercicios.*
- 15.8.** *Problemas.*

---

## INTRODUCCIÓN

En este capítulo se estudian en detalle las estructuras de datos pilas y colas que son probablemente las utilizadas mas frecuentemente en los programas más usuales. **Son** estructuras de datos que almacenan y recuperan sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, first-out*, último en entrar-primeramente en salir) y las colas como estructuras **FIFO** (*First-in, First-out*, primero en entrar-primeramente en **salir**). Entre las numerosas aplicaciones de las pilas destaca la evaluación de expresiones algebraicas, así como la organización de la memoria. **Las** colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

---

## CONCEPTOS CLAVE

- Concepto de tipo abstracto de datos.
- Concepto de una cola.
- Concepto de una pila.
- Listas enlazadas.

### 15.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por su parte superior (**cima**) de la pila. Éste es el caso de una pila de platos, una pila de libros, etc.

**Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado cima.**

Cuando se dice que *la pila está ordenada*, lo que se quiere decir es que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador «menor que» (<) y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

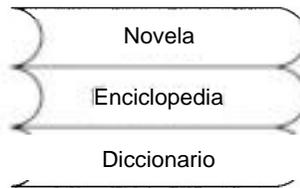


Figura 15.1. Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y, por último, el diccionario. Debido a su propiedad específica «último en entrar, primero en salir» se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*). Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **Insertar** (push) añade un elemento en la cima de la pila y la operación **Quitar** (pop) elimina o saca un elemento de la pila. La Figura 15.3 muestra una secuencia de operaciones *Insertar* y *Quitar*. El último elemento añadido a la pila es el primero que se quita de la pila.

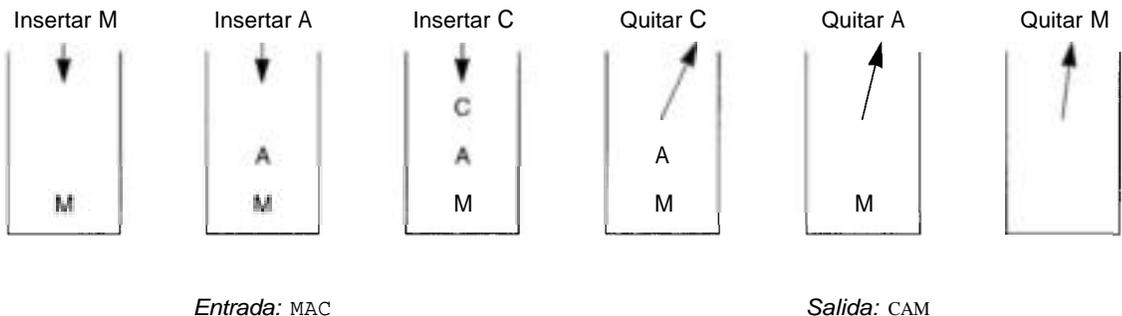


Figura 15.2. Poner y quitar elementos de la pila.

La operación Insertar (*push*) sitúa un elemento dato en la cima de la pila y Quitar (*pop*) elimina o quita el elemento de la pila.

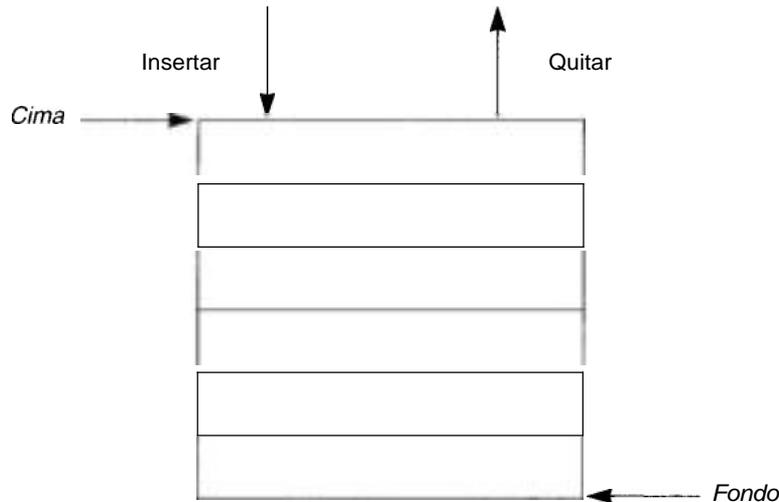


Figura 15.3. Operaciones básicas de una pila.

La pila se puede implementar mediante arrays en cuyo caso su dimensión o longitud es fija, y mediante punteros o listas enlazadas en cuyo caso se utiliza memoria dinámica y no existe limitación en su tamaño.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en el caso de tener tamaño fijo, si no cabe más elementos en la pila). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila se produce un error llamado **desbordamiento** (*overflow*) o *rechosamiento*. Para evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

### 15.1.1. Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir una pila).

|                          |                                                        |
|--------------------------|--------------------------------------------------------|
| <i>Tipo de dato</i>      | Dato que se almacena en la pila.                       |
| <i>Insertar (push)</i>   | Insertar un dato en la pila.                           |
| <i>Quitar (pop)</i>      | Sacar (quitar) un dato de la pila.                     |
| <i>Pila vacía</i>        | Comprobar si la pila no tiene elementos.               |
| <i>Pila llena</i>        | Comprobar si la pila está llena de elementos.          |
| <i>Limpiar pila</i>      | Quitar todos sus elementos y dejar la pila vacía.      |
| <i>Tamaño de la pila</i> | Número de elementos máximo que puede contener la pila. |
| <i>Cima</i>              | Obtiene el elemento cima de la pila.                   |

## 15.2. EL TIPO PILA IMPLEMENTADO CON ARRAYS

Una pila se puede implementar mediante *arrays* o mediante listas enlazadas. Una implementación estática se realiza utilizando un array de tamaño fijo y una implementación dinámica mediante una lista enlazada.

En C para definir una pila con arrays se utiliza una estructura. Los miembros de la estructura pila incluyen una lista (*array*) y un índice o puntero a la cima de la pila; además una constante con el máximo número de elementos. El tipo pila junto al conjunto de operaciones de la pila se pueden encerrar en un archivo de inclusión (*pila.h*). Al utilizar un array para contener los elementos de la pila hay que tener en cuenta que el tamaño de la pila no puede exceder el número de elementos del array y la condición *pila llena* será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición 0 del array y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array (en la pila) de modo que el primer elemento añadido se introduce en una pila vacía y en la posición 0, el segundo elemento en la posición 1, el siguiente en la posición 2 y así sucesivamente. Con estas operaciones el puntero (apuntador) que apunta a la cima de la pila se va incrementando en 1 cada vez que se añade un nuevo elemento; es decir, el *puntero de la pila* almacena el índice del array que se está utilizando como cima de la pila. Los algoritmos de introducir «insertar» (*push*) y quitar «sacar» (*pop*) datos de la pila utilizan el índice del array como puntero de la pila son:

#### Insertar (*push*)

1. Verificar si la pila no está llena.
2. Incrementar en 1 el puntero de la pila.
3. Almacenar elemento en la posición del puntero de la pila.

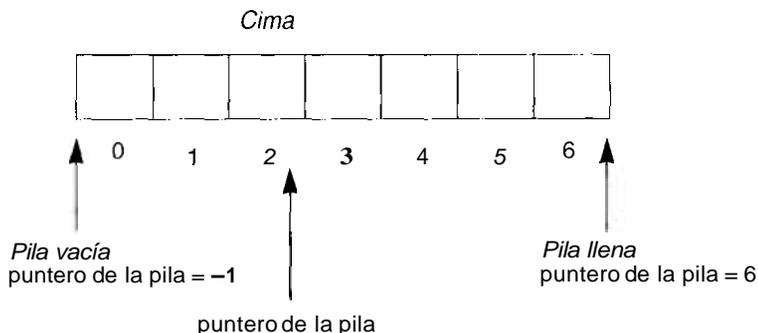
#### Quitar (*pop*)

1. Si la pila no está vacía.
2. Leer el elemento de la posición del puntero de la pila.
3. Decrementar en 1 el puntero de la pila.

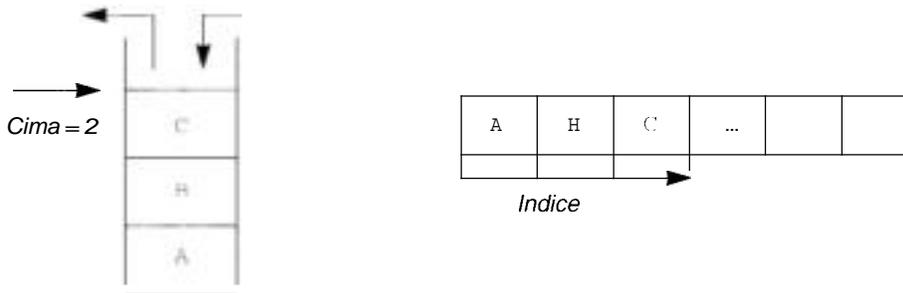
En el caso de que el *array* que define la pila tenga *TamañoPila* elementos, las posiciones del array, es decir, el índice o puntero de la pila, estarán comprendidas en el rango 0 a *TamañoPila*-1 elementos, de modo que *en una pila llena* el puntero de la pila apunta a *TamañoPila*-1 y *en una pila vacía* el puntero de la pila apunta a -1, ya que 0, teóricamente, será el índice del primer elemento.

### Ejemplo 15.1

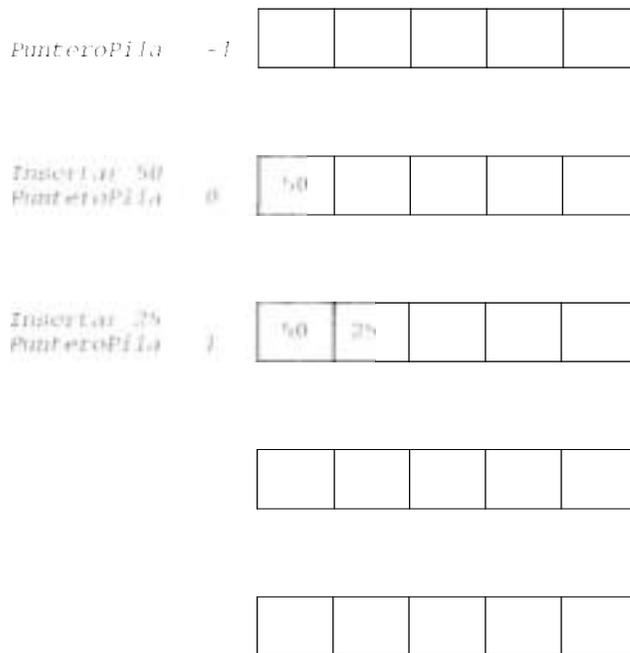
Una *pila* de 7 elementos se puede representar gráficamente así:



Si se almacenan los datos A, B, C, ... en la pila se puede representar gráficamente por alguno de estos métodos



Veamos ahora como queda la pila en función de diferentes situaciones de un posible programa.



### 15.2.1. Especificación del tipo `pila`

La declaración de una pila incluye los datos y operaciones ya citados anteriormente.

1. Datos de la pila (tipo `TipoData`, que es conveniente definirlo mediante `typedef`).
2. Verificar que la pila no está llena antes de intentar insertar o poner («push») un elemento en la pila ; verificar que una pila no está vacía antes de intentar quitar sacar («pop») un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error y el programa debe terminar.
3. `PilaVacía` devuelve 1 (verdadero) si la pila está vacía y 0 (falso) en caso contrario.
4. `PilaLlena` devuelve 1 (verdadero) si la pila está llena y 0 (falso) en caso contrario. Estas funciones se utilizan para verificar las operaciones del párrafo 2.
5. `LimpiarPila`. Se limpia o vacía la pila, dejándola sin elementos y disponible para otras tareas.
6. `Cima`, devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que la pila queda intacta.

**Declaración**

```

/* archivo pilaarray.h */
#include <stdio.h>
#include <stdlib.h>

#define MaxTamaPila 100

typedef struct

 TipoDato listapila[MaxTamaPila];
 int cima;
}Pila;

/* Operaciones sobre la Pila */

void CrearPila(Pila* P);
void Insertar(Pila* P, const TipoDato elemento);
TipoDato Quitar(Pila* P);
void LimpiarPila(Pila* P);

/* Operación de acceso a Pila */
TipoDato Cima(Pila P);

/* verificación estado de la Pila */

int PilaVacía(Pila P);
int PilaLlena(Pila P);

```

**Antes de incluir el archivo pilaarray.h debe de declararse el TipoDato. Así si se quiere una pila de enteros:**

```

typedef int TipoDato;
#include "pilaarray.h"

```

**En el caso de que la pila fuera de números complejos:**

```

typedef struct

 float x,y;
}TipoDato;
#include "pilaarray.h"

```

**Ejemplo 15.2**

*Escribir un programa que manipule una Pila de enteros, con el tipo definido anteriormente e introduzca un dato de tipo entero.*

**El programa crea una pila de números enteros, inserta en la pila un dato leído del teclado y visualiza el elemento cima.**

```

typedef int TipoDato;
#include "pilaarray.h";
#include <stdio.h>
void main()
{
 Pila P;
 int x;

 CrearPila(&P); /* Crea una pila vacía */
 scanf("%d",&x);

```

```

Insertar(&P,x); /* inserta x en la pila P */
printf("%d \n",Cima(P)); /* visualiza el último elemento */

/* Elimina el elemento cima (x) y deja la pila vacía */
if (!Pilavacia(P))
 aux = Quitar(&P);
printf("%d \n",aux);
LimpiarPila(&P); /* limpia la pila, queda vacía */
}

```

## 15.2.2. Implementación de las operaciones sobre pilas

Las operaciones de la pila definidas en la especificación se implementan en el archivo `pilaarray.c` para después formar un proyecto con otros módulos y la función principal.

```

/* archivo pilaarray.c */
#include "pilaarray.h"

/* Inicializa la pila a pila vacía */
void CrearPila(Pila* P)

 P -> cima = -1;

```

Las otras operaciones de la pila declaradas en el archivo `pilaarray.h` son: `Insertar`, `Quitar` y `Cima`. La operación `Insertar` y `Quitar`, insertan y eliminan un elemento de la pila; la operación `Cima` permite a un cliente recuperar los datos de la cima de la pila sin quitar realmente el elemento de la misma.

La operación de `Insertar` un elemento en la pila incrementa el puntero de la pila (`cima`) en 1 y asigna el nuevo elemento a la lista de la pila. Cualquier intento de añadir un elemento en una pila llena produce un mensaje de error «Desbordamiento pila» y debe terminar el programa.

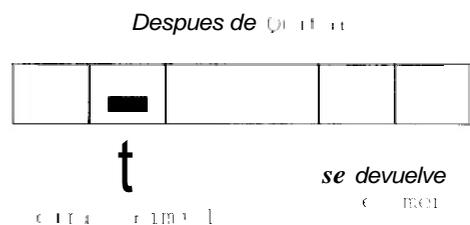
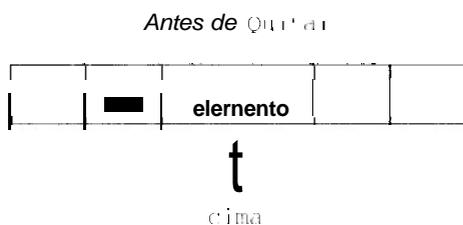
```

/* poner un elemento en la pila */
void Insertar(Pila* P,const TipoDato elemento)

 /* si la pila está llena, termina el programa */
 if (P->cima == MaxTamaPila-1)
 {
 puts("Desbordamiento pila");
 exit (1);
 }

 /* incrementar puntero cima y copiar elemento en listapila */
 P->cima++;
 P->listapila[P->cima] = elemento;
}

```



La operación `Quitar` elimina un elemento de la pila copiando primero el valor de la cima de la pila en una variable local `aux` y a continuación decreenta el puntero de la pila en 1. La variable `aux` se devuelve en la ejecución de la operación `Quitar`. Si se intenta eliminar o borrar un elemento en una pila vacía se debe producir un mensaje de error y el programa debe terminar.

```

/* Quitar un elemento de la pila */
TipoDato Quitar(Pila* P)
{
 TipoDato aux;
 /* si la pila está vacía, termina el programa */
 if (P->cima == -1)
 {
 puts("Se intenta sacar un elemento en pila vacía");
 exit (1);
 }

 /* guardar elemento de la cima */
 aux = P->listapila[P->cima];

 /* decrementar cima y devolver volor del elemento */
 P->cima--;
 return aux;
}

```

### 15.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben el estado de la pila: *pila vacía* o *pila llena*. Asimismo se ha de definir una operación que restaure la condición inicial de la pila, que fue determinada por el constructor `CrearPila` (*cima de la pila a -1*), `LimpiarPila`.

La función `PilaVacía` comprueba (verifica) si la cima de la pila es `-1`. En ese caso, la pila está vacía y se devuelve un `1` (verdadero); en caso contrario, se devuelve `0` (falso).

```

/* verificar pila vacía */
int PilaVacía(Pila P)
{
 /*devuelve el valor lógico resultante de expresión cima == -1 */
 return P.cima == -1;
}

```

La función `PilaLlena` comprueba (verifica) si la cima es `MaxTamaPila-1`. En ese caso, la pila está llena y se devuelve un `1` (verdadero); en caso contrario, se devuelve `0` (falso).

```

/* verificar si la pila está llena */
int PilaLlena (Pila P)
{
 /* devuelve valor lógico de la expresión cima == MaxTamaPila-1 */
 return P.cima == MaxTamaPila-1;
}

```

Por último la operación `LimpiarPila` reinicializa la cima a su valor inicial con la pila vacía (`-1`).

```

/* quitar todos los elementos de la pila */
void LimpiarPila(Pila* P)
{
 P->cima = -1;
}

```

## Ejercicio 15.1

Escribir un programa que utilice la clase *Pila* para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. **Nota.** Una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor: *alila*, es un palíndromo; *cara (arac)* no es un palíndromo.

### Análisis

La palabra se lee carácter a carácter, de tal forma que a la vez que se añade a un `string` se inserta en una pila de caracteres. Una vez leída la palabra, se compara el primer carácter del `string` con el carácter que se extrae de la pila, si son iguales sigue la comparación con siguiente carácter del `string` y de la pila; así hasta que la pila se queda vacía o hay un carácter no coincidente.

Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones son entre caracteres que están en orden inverso: primero con Último...

La codificación consta de tres archivos, el archivo `pilarray.h` con las declaraciones de la pila; el archivo `pilarray.c` con la implementación de las operaciones de la pila y el archivo `paldromo.c` para leer la palabra y comprobar con ayuda de la pila si es palíndromo.

```

/* Archivo pilarray.h */
#include <stdio.h>
#include <stdlib.h>

#define MaxTamaPila 100

typedef struct
{
 TipoDato listapila[MaxTamaPila];
 int cima;
} Pila;

/* Operaciones sobre la Pila */

void CrearPila(Pila* P);
void Insertar(Pila* P, const TipoDato elemento);
TipoDato Quitar(Pila* P);
void LimpiarPila(Pila* P);

/* Operación de acceso a Pila */

TipoDato Cima(Pila P);

/* verificación estado de la Pila */

int Pilavacia(Pila P);
int PilaLlena(Pila P);

/* Archivo pilarray.c
 Implementación de operaciones sobre pilas
*/

typedef char TipoDato;
#include "pilarray.h"

/* Inicializa la pila a pila vacía */
void CrearPila(Pila* P)
{
 P -> cima = -1;
}

/* poner un elemento en la pila */

```

```

void Insertar(Pila* P, const TipoDato elemento)
{
 /* si la pila está llena, termina el programa */
 if (PilaLlena (*P))
 {
 puts("Desbordamiento pila");
 exit (1);
 }

 /* incrementar puntero cima y copiar elemento en listapila */
 P->cima++;
 P->listapila[P->cima] = elemento;
}

/* Quitar un elemento de la pila */
TipoDato Quitar (Pila P)
{
 TipoDato Aux;
 /* si la pila está vacía, termina el programa */
 if (Pilavacia (*P))
 {
 puts("Se intenta sacar un elemento en pila vacía");
 exit (1);
 }

 /* guardar elemento de la cima */
 Aux = P->listapila[P->cima];

 /* decrementar cima y devolver valor del elemento */
 P->cima--;
 return Aux;
}

/* verificar pila vacía */
int Pilavacia (Pila P)
{
 /* devuelve el valor lógico resultante de expresión cima == -1 */
 return P.cima == -1;
}

/* verificar si la pila está llena */
int PilaLlena (Pila P)
{
 return P.cima == MaxTamaPila-1;
}

/* quitar todos los elementos de la pila */
void LimpiarPila (Pila* P)
{
 P->cima = -1;
}

TipoDato Cima (Pila P)
{
 if (P.cima == -1)
 {
 puts("Se intenta sacar un elemento en pila vacía");
 exit (1);
 }
 return P.listapila[P.cima];
}

```

```

I
/* Archivo paldromo.c */
typedef char TipoDato;
#include 'pilarray.h'
#include <ctype.h>
int main()
{
 char palabra[100], ch;
 Pila P;
 int j, palmo;
 CrearPila(&P);
 /* Lee la palabra */
 do {
 puts("\n Palabra a comprobar si es palíndromo");
 for (j=0; (ch=getchar())!='\n'; j)
 {
 palabra[j++] = ch;
 Insertar(&P,ch); /* pone en la pila */
 }
 palabra[j] = '\0';
 /* comprueba si es palíndromo */
 palmo = 1;
 for (j=0; palmo && !PilaVacía(P);)
 {
 palmo = palabra[j++] == Quitar(&P);
 }
 LimpiarPila(&P);
 if (palmo)
 printf("\n La palabra %s es un palíndromo \n",palabra);
 else
 printf("\n La palabra %s no es un palíndromo \n",palabra);

 printf("\n ¿ Otra palabra ?: "); scanf("%c%c",&ch);
 }while (tolower(ch) == 's');

 return 0;
I

```

---

### 15.3. COLAS

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (Fig. 15.4). Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por la frente (parte inicial, cabeza) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia

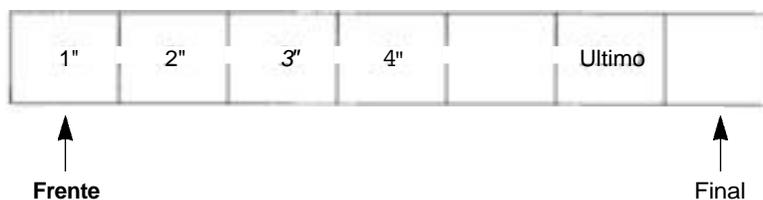


Figura 15.4. Una cola.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** (*first-in/first-out*, *primero en entrar/primero en salir* o bien *primero en llegar/primero en ser servido*). El servicio de atención a clientes en un almacén es un ejemplo típico de cola. La acción de gestión de memoria intermedia (*buffering*) de trabajos o tareas de impresora en un distribuidor de impresoras (*spooler*) es otro ejemplo típico de cola<sup>1</sup>. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una Única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas. deberá esperar a la impresión de esas 300 páginas antes de que se imprima su página.

Desde el punto de vista de estructura de datos, una cola es similar a una pila, en donde los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola. Las acciones que están permitidas en una cola son:

- o Creación de una cola vacía.
- o Verificación de que una cola está vacía.
- o Añadir un dato al final de una cola.
- o Eliminación de los datos de la cabeza de la cola.

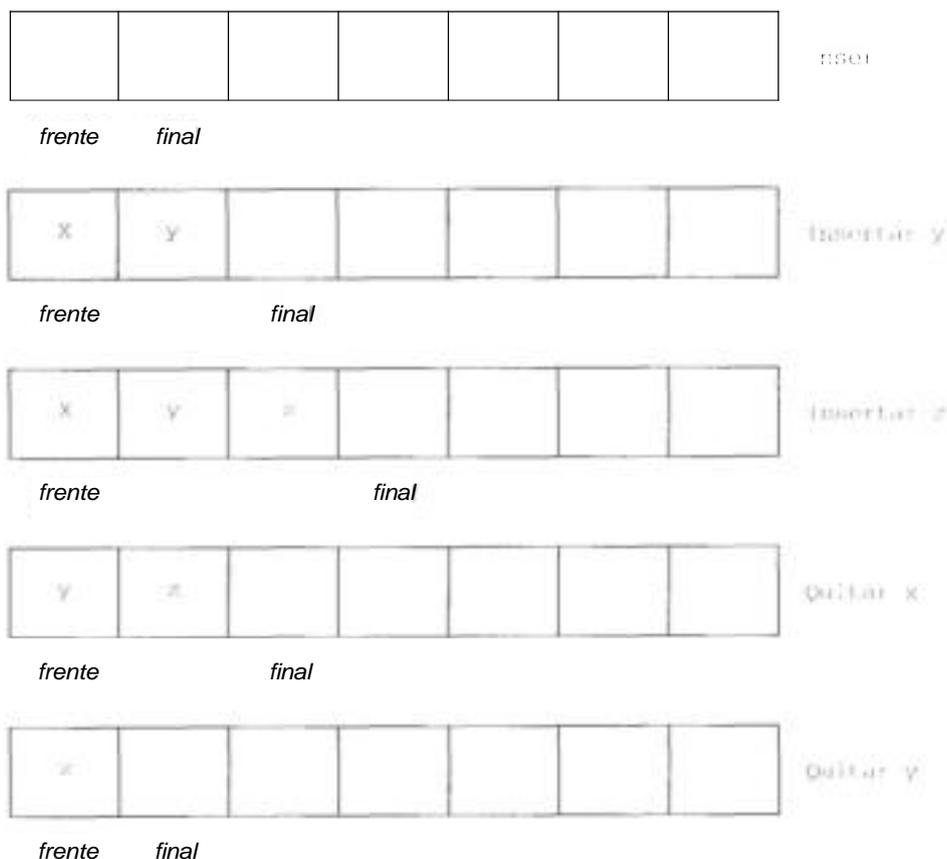


Figura 15.5. Operaciones de insertar y quitar en una Cola

<sup>1</sup> Recuerde que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se «encolan» en la cola de impresión.

## 15.4. EL TIPO COLA IMPLEMENTADA CON ARRAYS

Al igual que las pilas, las colas se pueden implementar utilizando arrays o listas enlazadas. En esta sección se considera la implementación utilizando arrays.

La definición de una `Cola` ha de contener un array para almacenar los elementos de la cola, y dos marcadores o punteros (variables) que mantienen las posiciones frente y final de la cola; es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. Cuando un elemento se añade a la cola, se verifica si el inarcador final apunta a una posición válida, entonces se añade el elemento a la cola y se incrementa el marcador final en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1. Este procedimiento funciona bien hasta la primera vez que el puntero de cabeza o cabecera alcanza el extremo del array y el array queda o bien vacío o bien lleno.

### 15.4.1. Definición de la especificación de una cola

Una cola debe manejar diferentes tipos de datos; por esta circunstancia, se define en primer lugar el tipo genérico `TipoDato`. La clase `Cola` contiene una lista (`listaQ`) cuyo máximo tamaño se determina por la constante `MaxTamQ`. Se definen dos tipos de variables puntero o marcadores, `frente` y `final`. Éstas son los punteros de cabecera y cola o final respectivamente.

Las operaciones típicas de la cola son: `InsertarQ`, `EliminarQ`, `Qvacía`, `Qllena`, y `FrenteQ`. `InsertarQ` toma un elemento del tipo `TipoDato` y lo inserta en el final de la cola. `EliminarQ` elimina (quita) y devuelve el elemento de la cabeza o frente de la cola. La operación `FrenteQ` devuelve el valor del elemento en el frente de la cola, sin eliminar el elemento y, por tanto, no modifica la cola.

La operación `Qvacía` comprueba si la cola está vacía, es necesario esta comprobación antes de eliminar un elemento. `Qllena` comprueba si la pila esta llena, esta comprobación se realiza antes de insertar un nuevo miembro. Si las precondiciones para `InsertarQ` y `EliminarQ` se violan, el programa debe imprimir un mensaje de error y terminar.

### 15.4.2. Especificación del tipo `cola`

La declaración del tipo de dato `Cola` y los prototipos de las operaciones de la cola se almacena en un archivo de cabecera "colaarray.h".

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

#define MaxTamQ 100
typedef struct
{
 int frente;
 int final;
 TipoDato listaQ[MaxTamQ];
}Cola;

/* Operaciones del tipo de datos Cola */

/* operaciones de modificación de la cola */
void CrearCola(Cola* Q); /* inicializa la cola como vacía */
void InsertarQ(Cola* Q, TipoDato elemento);
TipoDato EliminarQ(Cola* Q);
void BorrarCola (Cola* Q);
```

```

/* acceso a la cola */
TipoDato FrenteQ(Cola Q);

/* métodos de verificación del estado de la cola */
int LongitudQ(Cola Q);
int Qvacía(Cola Q);
int Qllena(Cola Q);

```

### 15.4.3. Implementación del tipo cola

La declaración que se ha hecho del tipo Cola contiene un array para el almacenamiento de los elementos de la cola y dos marcadores o punteros: uno apuntando a la posición de la cabeza o cabecera de la cola y la otra al primer espacio vacío a continuación del final de la cola. Cuando un elemento se añade a la cola, se hace un test (prueba) para ver si el marcador final apunta a una posición válida, a continuación se añade el elemento a la cola y el marcador final se incrementa en 1. Cuando se quita (elimina) un elemento de la cola, se realiza un test (prueba) para ver si la cola está vacía, y si no es así, se recupera el elemento que se encuentra en la posición apuntada por el marcador de cabeza y el marcador de cabeza se incrementa en 1.

Este procedimiento funciona bien hasta la primera vez que el marcador final alcanza el final del array. Si durante este tiempo se han producido eliminaciones, habrá espacio vacío al principio del array. Sin embargo, puesto que el marcador final apunta al extremo del array, implicará que la cola está llena y ningún dato más se añadirá. Se pueden desplazar los datos de modo que la cabeza de la cola vuelve al principio del array cada vez que esto sucede, pero el desplazamiento de datos es costoso en términos de tiempo de computadora, especialmente si los datos almacenados en el array son estructuras de datos grandes.

El medio más eficiente, sin embargo, para almacenar una cola en un array, es utilizar un tipo especial de array que junte el extremo final de la cola con su extremo cabeza. Tal array se denomina *array circular* y permite que el array completo se utilizará para almacenar elementos de la cola sin necesidad de que ningún dato se desplace. Un array circular con  $n$  elementos se visualiza en la Figura 15.6.

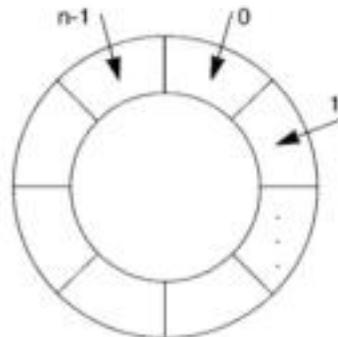


Figura 15.6. Un array circular.

El array se almacena de modo natural en la memoria tal como un bloque lineal de  $n$  elementos. Se necesitan dos marcadores (punteros) *cabeza* y *final* para indicar la posición del elemento que precede a la cabeza y la posición del final, donde se almacenó el último elemento añadido. Una cola vacía se representa por la condición `cabeza = final`.

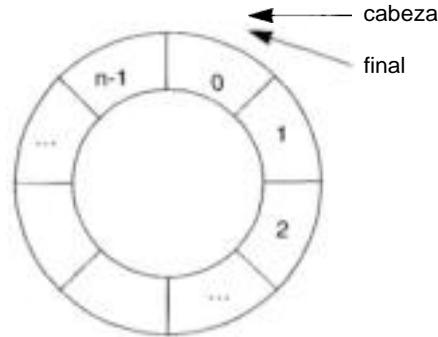


Figura 15.7. Una cola vacía.

La variable frente o cabeza es siempre la posición del elemento que precede al primero de la cola y se avanza en el sentido de las agujas del reloj. La variable `final` es la posición en donde se hizo la última inserción. Después que se ha producido una inserción, `final` se mueve circularmente a la derecha. La implementación del movimiento circular se realiza utilizando la *teoría de los restos*:

Mover `final` adelante =  $(\text{final} + 1) \% \text{MaxTamQ}$   
 Mover `cabeza` adelante =  $(\text{frente} + 1) \% \text{MaxTamQ}$

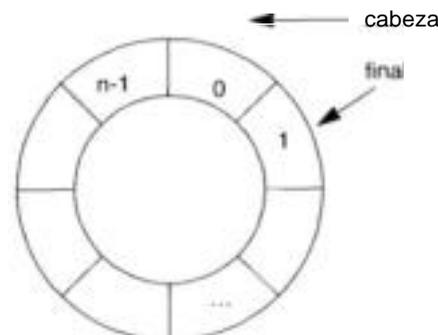


Figura 15.8. Una cola que contiene un elemento

Los algoritmos que formalizan la gestión de colas en un array circular han de incluir al menos las siguientes tareas:

- Creación de una cola vacía: `cabeza = final = 0`.
- Comprobar si una cola está vacía:  
`es cabeza == final ?`
- Comprobar si una cola está llena:  
`(final + 1) % MaxTamQ == cabeza ?`
- Añadir un elemento a la cola: si la cola no está llena, añadir un elemento en la posición siguiente a `final` y se establece:  
`final = (final + 1) % MaxTamQ (% operador resto)`
- Eliminación de un elemento de una cola: si la cola no está vacía, eliminarlo de la posición siguiente a `cabeza` y establecer `cabeza = (cabeza + 1) % MaxTamQ`.

### 15.4.4. Operaciones de la cola

Una cola permite un conjunto limitado de operaciones, para inicializar la cola, para añadir un nuevo elemento (`InsertarQ`) o quitar/eliminar un elemento (`EliminarQ`). El tipo `Cola` proporciona también `frenteQ`, que permite «ver» el primer elemento de la cola. Para esta implementación, con array circular, el tipo cola es el siguiente:

```
#define MaxTamQ 100
typedef struct

 int frente;
 int final;
 TipoDato listaQ[MaxTamQ];
}Cola;
```

#### Crearcola

La primera operación que se realiza sobre una cola es inicializarla para que a continuación puedan añadirse elementos a la cola.

```
void CrearCola(Cola* Q)
{
 Q->frente = 0;
 Q->final = 0;
}
```

#### InsertarQ

Antes de que comience el proceso de inserción, el índice `final` apunta al último elemento insertado. El nuevo elemento se sitúa en la posición siguiente. El cálculo de las posiciones sucesivas se consigue mediante el operador resto (%). Después de situar el elemento de la lista, el índice `final` se debe actualizar para apuntar en la siguiente posición.

```
/* insertar elemento en la cola */

void InsertarQ(Cola* Q, TipoDato elemento)
{ /* terminar si la cola está llena */
 if (Qllena(Q))
 {
 puts ("desbordamiento cold");
 exit (1);
 }

 /* asignar elemento a listaQ y actualizar final */
 Q->final = (Q->final + 1)% MaxTamQ;
 Q->listaQ[Q->final] = elemento;
```

#### EliminarQ

La operación `EliminarQ` borra o elimina un elemento del frente de la cola, una posición que se referencia por el índice `frente`. Comienza el proceso de eliminación avanzando `frente` ya que se estableció que referencia al anterior elemento.

```
frente = (frente + 1)% MaxTamQ;
```

En el modelo circular, la cabeza se debe volver a posicionar en el siguiente elemento de la lista utilizando el operador resto (%). El código fuente es:

```
/* borrar elemento del frente de la cold y devuelve su valor */
TipoDato EliminarQ(Cola* Q)
```

```

{
 TipoDato aux;
 /*si listaQ está vacía, terminar el programa */
 if (Qvacía(Q))
 {
 puts("Eliminación de una cold vacía");
 exit (1);
 }
 /* avanzar frente y devolver primero del frente */
 Q->frente = (Q->frente + 1) % MaxTamQ;
 aux = Q->listaQ[Q->frente];
 return aux;
}

```

### FrenteQ

La operación FrenteQ obtiene el elemento del frente de la cola, una posición que se referencia por el índice frente.

```

TipoDato FrenteQ(Cola Q)
{
 TipoDato aux;
 /*si la cola está vacía, terminar el programa */
 if (Qvacía(Q))
 puts("Elemento frente de una cold vacía");
 exit (1);
 return (Q.listaQ[(Q.frente+1)%MaxTamQ]);
}

```

### Qvacía

Las operaciones que preguntan por el estado de la cola pueden implementarse preguntando por los campos frente y final. La operación Qvacía prueba si la cola no tiene elementos.

```

int Qvacía(Cola Q)
{
 return (Q.frente == Q.final);
}

```

### Qllena

La operación Qllena prueba si la cola no puede contener mas elementos.

```

int Qllena(Cola Q)
{
 return (Q.frente == (Q.final+1)%MaxTamQ);
}

```

## 15.5. REALIZACIÓN DE UNA COLA CON UNA LISTA ENLAZADA

La realización de una cola mediante una lista enlazada permite ajustarse exactamente al número de elementos de la cola. Esta implementación utiliza dos punteros para acceder a la lista. El puntero Frente y el puntero Finu.l.

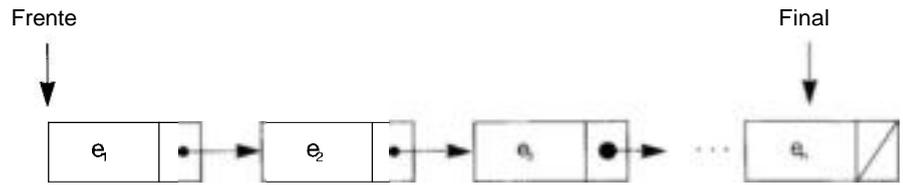


Figura 15.9. Cola con lista enlazada (representación gráfica típica)

El puntero *Frente* referencia al primer elemento de la cola, el primero en ser retirado de la cola. El puntero *Final* referencia al último elemento en ser añadido, el último que será retirado.

Con esta representación no tiene sentido la operación que prueba si la cola está llena. Al ser una estructura dinámica puede crecer y decrecer según las necesidades (el límite está en la memoria libre del computador).

### 15.5.1. Declaración del tipo cola con listas

Para esta representación se declara una estructura que represente al nodo de la lista enlazada, un puntero a esta estructura y la estructura cola con los punteros *Frente* y *Final*. Las operaciones son las mismas, excepto la operación *Qllena* que no es necesaria al ser una estructura dinámica. La declaración se almacena en el archivo *colalist.h*.

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

struct nodo
{
 TipoDato elemento;
 struct nodo* siguiente;
};

typedef struct nodo Nodo;
typedef struct
{
 Nodo* Frente;
 Nodo* Final;
}Cola;

/* Los prototipos de las operaciones */
void CrearCola(Cola* Q); /* Inicializa la cola como vacía */
void InsertarQ(Cola* Q, TipoDato elemento);
TipoDato EliminarQ(Cola* Q);
void BorrarCola(Cola* Q);

/* acceso a la cola */
TipoDato FrenteQ(Cola Q);

/* métodos de verificación del estado de la cola */
int Qvacia(Cola Q);
```

## 15.5.2. Codificación de las operaciones del tipo cola con listas

Estas operaciones se van a almacenar en el archivo fuente `colalist.c`. En primer lugar hay que incluir el archivo `colalist.h` y declarar el tipo de dato de los elementos de la cola.

La inicialización de la cola, al ser una implementación con punteros, consiste en asignar el puntero nulo a `Frente` y `Final`. La operación de insertar se realiza creando un nuevo nodo (función auxiliar `crearnodo()`) y enlazándolo a partir del nodo final. La operación de eliminar se realiza sobre el otro extremo.

### Codificación de las operaciones.

```
typedef char TipoDato;
#include "colalist.h"

void CrearCola(Cola* Q)
{
 Q->Frente = Q->Final = NULL;
}

Nodo* crearnodo(TipoDato elemento)
{
 Nodo* t;
 t = (Nodo*)malloc(sizeof(Nodo));
 t->elemento = elemento;
 t->siguiente = NULL;
 return t;
}

int Qvacía(Cola Q)
{
 return (Q.Frente == NULL);
}

void InsertarQ(Cola* Q, TipoDato elemento)
{
 Nodo* a;
 a = crearnodo(elemento);
 if (Qvacía(*Q))
 {
 Q->Frente = a;
 }
 else
 {
 Q->Final->siguiente = a;
 }
 Q->Final = a;
}

TipoDato EliminarQ(Cola* Q)
{
 TipoDato aux;
 if (!Qvacía(*Q))

 Nodo* a;
 a = Q->Frente;
 aux = Q->Frente->elemento;
 Q->Frente = Q->Frente->siguiente;
 free(a);
}
```

```

else /* error: eliminar de una cola vacía */
{
 puts("Error cometido al eliminar de una cola vacía");
 exit(-1);

 return aux;
}

TipoDato FrenteQ(Cola Q)
{
 if (Q.vacia(Q))
 {
 puts ("Error: cola vacía");
 exit(1);
 }
 return (Q.Frente->elemento);

void BorrarCola (Cola* Q)
{
 /* Elimina y libera todos los nodos de la cola */
 for (; Q->Frente!=NULL;)
 {
 Nodo* n;
 n = Q->Frente;
 Q->Frente = Q->Frente->siguiente;
 free(n);
 }
}

```

---

## Ejercicio 15.2

Una variación del famoso problema matemático llamado «problema de José» permite generar números de la suerte. Se parte de una lista inicial de  $n$  números, esta lista se va reduciendo siguiendo el siguiente algoritmo:

1. Se genera un número aleatorio  $n_i$ .
2. Si  $n_i > n$  fin del algoritmo.
3. Si  $n_i \leq n$  se quitan de la lista los números que ocupan las posiciones  $1, 1 + n_i, 1 + 2*n_i, \dots, n$  toma el valor del número de elementos que quedan en la lista.
4. Se vuelve al paso 1.

### Análisis

El problema se va a resolver utilizando la estructura *Cola*. En primer lugar, se genera una lista de  $n$  números aleatorios que se almacena en una cola. A continuación, se siguen los pasos del algoritmo, en cada pasada se mueven los elementos de la cola a otra cola excepto aquellos que están en las posiciones (múltiplos de  $n_i$ )  $+1$ . Estas posiciones  $i$  se pueden expresar matemáticamente:

$$i \text{ modulo } n_i = 1$$

El tipo cola y las operaciones sobre colas se agrupan en el archivo de inclusión `cola.h` implementado con estructuras dinámicas. Además, se añade la operación de `mostrarcola` para escribir los números que quedan en la lista.

**Archivo con el tipo cola y prototipos de las operaciones**

```

#include <stdio.h>
#include <stdlib.h>

struct nodo
{
 TipoDato elemento;
 struct nodo* siguiente;
};

typedef struct nodo Nodo;
typedef struct
{
 Nodo* Frente;
 Nodo* Final;
} Cola;

/* Los prototipos de las operaciones */
void CrearCola(Cola* Q); /* Inicializa la cola como vacía */
void InsertarQ(Cola* Q, TipoDato elemento);
TipoDato EliminarQ(Cola* Q);
void BorrarCola(Cola* Q);
/* acceso a la cola */
TipoDato FrenteQ(Cola Q);
/* métodos de verificación del estado de la cola */
int Qvacía(Cola Q);

```

**Archivo con la implementación<sup>2</sup> de las operaciones**

```

/* colalist.c */
typedef int TipoDato;
#include "colalist.h"

```

**Archivo con el algoritmo para obtener números de la suerte**

```

typedef int TipoDato;
#include "colalist.h"
#include <time.h>
void MostrarCola(Cola* Q);
int main()
{
 Cola Q;
 int n, n1, n2, n3, i;
 randomize();
 /* Número de elementos de la lista */
 n = 1 + random(50);
 Crearcola(&Q);
 /* Se generan n números aleatorios */
 for (i=1; i<=n; i++)
 InsertarQ(&Q, 1+random(1001));
 n1 = 1+random(11);
 while (n1 <= n)
 {
 printf("\n Se quitan elementos a distancia %d ", n1);
 }
}

```

<sup>2</sup> La implementación está en el Apartado IS.S.2

```

n2 = 0; /* Contador de elementos que quedan */
for (i=1; i<=n; i++)
{
 n3 = EliminarQ(&Q); /* retira el elemento frente */
 if (i%n1 == 1)
 {
 printf("\t %d se quita.",n3);
 }
 else
 {
 InsertarQ(&Q,n3); /* se vuelve a meter en la cola */
 n2++;
 }

 n = n2;
 n1 = 1+random(11);
}

printf("\n Los números de la suerte: ");
MostrarCola(&Q);

return 1;
}

void MostrarCola(Cola* Q)

while (!Qvacía(*Q))

 printf("%d ",EliminarQ(Q));
}
}

```

## 15.6. RESUMEN

- Una **pila** es una estructura de datos tipo LIFO (*last in first out*, último en entrar/primer en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado **cima** de la pila.
- Se definen las siguientes operaciones básicas sobre pilas: *crear*, *insertar*, *cima*, *eliminar*, *pilavacía*, *pilallena* y *liberarpila*.
- *crear*, inicializa la pila como pila vacía.
- *insertar*, añade un elemento en la cima de la pila. Debe de haber espacio en la pila.
- *cima*, devuelve el elemento que está en la cima, sin extraerlo.
- *eliminar*, extrae de la pila el elemento cima de la pila.
- *pilavacía*, determina si el estado de la pila es vacía, en su caso devuelve el valor lógico **true**.
- *pilallena*, determina si existe espacio en la pila para añadir un nuevo elemento. De no haber espacio devuelve **true**. Esta operación se aplica en la representación de la pila mediante array.
- *liberarpila*, el espacio asignado a la pila se libera, queda disponible.
- Una cola es una lista lineal en la que los datos se insertan por un extremo (final) y se extraen por el otro extremo (**frente**). Es una estructura FIFO (*first in first out*, primero en entrar/primer en salir).
- Las operaciones básicas que se aplican sobre colas: *crear*, *qvacia*, *qllena*, *insertarq*, *frenteq*, *eliminarq*.
- *crear*, inicializa a una cola sin elementos.

- `qvacia`, determina si una cola tiene o no elementos. Devuelve *true* si no tiene elementos.
- `qllena`, determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un array para guardar los elementos de la cola.
- `insertarq`, añade un nuevo elemento a la cola, por el extremo final.
- `frenteq`, devuelve el elemento que está en el extremo frente sin sacarlo de la cola.

- \* `eliminarq`, extrae el elemento frente de la cola.
- Numerosos modelos de sistemas del mundo real son de tipo cola: cola de impresión en un servidor de impresoras, programas de simulación, colas de prioridades en organización de viajes. Una cola es la estructura típica que se suele utilizar como almacenamiento de datos, cuando se envían datos desde un componente rápido de una computadora a un componente lento (por ejemplo, una impresora).

## 15.7. EJERCICIOS

- 15.1. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`?

```
Pila P;
int x=4, y;

CrearPila(&P);
Insertar(&P,x);
printf("\n%d ",Cima(P));
y = Quitar(&P);
Insertar(&P,32);
Insertar(&P,Quitar(&P));
do {
 printf("\n%d",Quitar(&P));
}while (!PilaVacía(P));
```

- 15.2. Escribir en el archivo `pila.h` los tipos de datos y los prototipos de las operaciones básicas sobre pilas con estructuras dinámicas.
- 15.3. Escribir la función `MostrarPila()` para escribir los elementos de una pila de cadenas de caracteres, utilizando sólo las operaciones básicas y una pila auxiliar.
- 15.4. Obtener una secuencia de 10 elementos reales, guardarlos en un array y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.
- 15.5. Considerar una cola de nombres representada por una array circular con 6 posiciones, el cam-

po frente con el valor: `Frente = 2`. Y los elementos de la Cola: `Mar`, `Sella`, `Centurión`.

Escribir los elementos de la cola y los campos `Frente` y `Final` según se realizan estas operaciones:

- **Añadir** Gloria y Generosa a la cola.
- Eliminar de la cola.
- **Añadir** Positivo.
- **Añadir** Horche a la cola.
- Eliminar todos los elementos de la cola.

- 15.6. Una bicola es una estructura de datos lineal en la que la inserción y borrado se pueden hacer tanto por el extremo frente como por el extremo final. Suponer que se ha elegido una representación dinámica, con punteros, y que los extremos de la lista se denominan `frente` y `final`. Escribir la implementación de las operaciones: `InsertarFrente()`, `InsertarFinal()`, `EliminarFrente()` y `EliminarFinal()`.

- 15.7. Considere una bicola de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5 final = 7
Bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos F y K por el **final** de la bicola.
- Añadir los elementos R, W y V por el **frente** de la bicola.
- Añadir el elemento M por el **final** de la bicola.
- Eliminar dos caracteres por el **frente**.
- Añadir los elementos K y L por el **final** de la bicola.

- Añadir el elemento S por el **frente** de la bicola. -

15.8. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos que son par de la pila en la cola.

## 15.8. PROBLEMAS

15.1. Escribir una función, **copiarPila**, que copie el contenido de una pila en otra. La función tendrá dos argumentos de tipo **pila**, uno para la pila fuente y otro para la pila destino. Utilizar las operaciones **definidas** sobre el tipo de datos **pila**.

15.3. Con un archivo de texto se quieren realizar las siguientes acciones: formar una lista de colas, de **tal** forma que en cada nodo de la lista tenga la dirección de una cola que tiene todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por **a**, a continuación las de la letra **b**, y así sucesivamente.

15.3. Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma:

$$X \ \& \ Y$$

donde X es una cadena de caracteres e Y es la cadena inversa. El carácter & es el separador.

15.4. Escribir un programa que haciendo uso del tipo **Pila** de caracteres, procese cada uno de los caracteres de una expresión que viene dada en una línea de caracteres. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

$$((a+b)*5) - 7$$

A esta otra expresión le falta un corchete:

$$2*[(a+b)/2.5 + x - 7*y$$

15.5. Se tiene un archivo de texto del cual se quiere determinar las frases que son palíndromo. Para lo cual se ha de seguir la siguiente estrategia:

- Considerar cada línea del texto una frase.
- Añadir cada carácter de la frase a una pila y a la vez a una cola.
- Extraer carácter a carácter, y simultáneamente de la pila y de la cola. Su comparación determina si es palíndromo o no.

Escribir un programa en **C** que lea cada línea del archivo y determine si es palíndromo.

15.6. Escribir un programa en el que se generen 100 números aleatorios en el rango  $-25 \dots +25$  y se guarden en una cola implementada mediante un array considerado circular. Una vez creada la cola, el usuario puede pedir que se forme otra cola con los números negativos que tiene la cola original.

15.7. Escribir una función que tenga como argumentos dos colas del mismo tipo. Devuelva cierto si las dos colas son idénticas.

15.8. Escribir un programa en el que se manejen un total de  $n=5$  pilas: **P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub> y P<sub>5</sub>**. La entrada de datos será pares de enteros ( **i**, **j** )

tal que  $1 \leq \text{abs}(i) \leq n$ . De tal forma que el criterio de selección de pila:

- Si  $i$  es positivo, debe de insertarse el elemento  $j$  en la pila  $P_i$ .
- Si  $i$  es negativo, debe de eliminarse el elemento  $j$  de la pila  $P_i$ .
- Si  $i$  es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe de escribir el contenido de la  $n$  pilas en pantalla.

**15.9.** Modificar el Problema 15.8 para que la entrada sean triplas de números enteros  $(i, j, k)$ , donde  $i, j$  tienen el mismo significado que en 15.8, y  $k$  es un número entero que puede tomar los valores  $-1, 0$  con este significado:

- $-1$ , hay que borrar todos los elementos de la pila.

- $0$ , el proceso es el indicado en 15.8 con  $i$  y  $j$ .

**15.10.** Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, el carro de la compra que tiene queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

# CAPÍTULO 16

## ÁRBOLES

---

### CONTENIDO

- 16.1. Árboles generales.
- 16.2. Resumen de definiciones.
- 16.3. Árboles binarios.
- 16.4. Estructura de un árbol binario.
- 16.5. Operaciones en árboles binarios.
- 16.6. Árbol de expresiones.
- 16.7. Recorrido de un árbol.
- 16.8. Árbol binario de búsqueda.
- 16.9. Operaciones en árboles binarios de búsqueda.
- 16.10. Aplicaciones de árboles en algoritmos de exploración.
- 16.11. *Resumen.*
- 16.12. *Ejercicios.*
- 16.13. *Problemas.*
- 16.14. Referencias bibliográficas.

---

## INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales* al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales*.

Los árboles son muy utilizados en informática para representar fórmulas algebraicas como un método eficiente para búsquedas grandes y complejas, listas dinámicas y aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi **todos** los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

En el capítulo se estudiara el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda. Asimismo se estudiarán algunas aplicaciones típicas del diseño y construcción de árboles.

---

## CONCEPTOS CLAVE

- **Árbol.**
- **Árbol binario.**
- **Árbol binario de búsqueda.**
- **Conceptos teóricos (*nivel, profundidad, raíz, hoja, rama, ...*).**
- **Enorden.**
- **Nodo.**
- **Preorden.**
- **Postorden.**
- **Recorrido de un árbol.**
- **Subárbol.**

### 16.1. ÁRBOLES GENERALES

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 16.1 representa dos ejemplos de árboles generales.

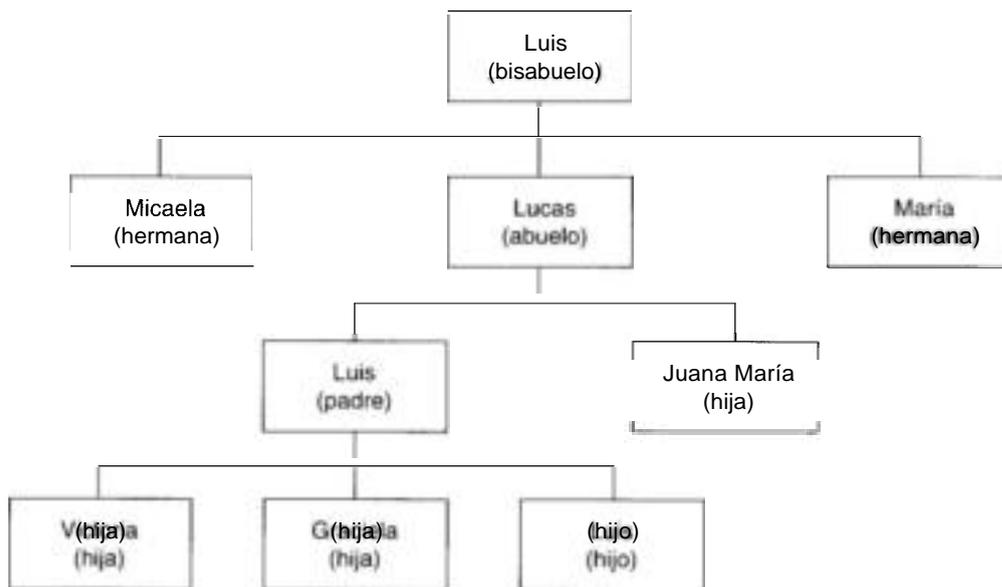


Figura 16.1. Árbol genealógico (bisabuelo-bisnetos).

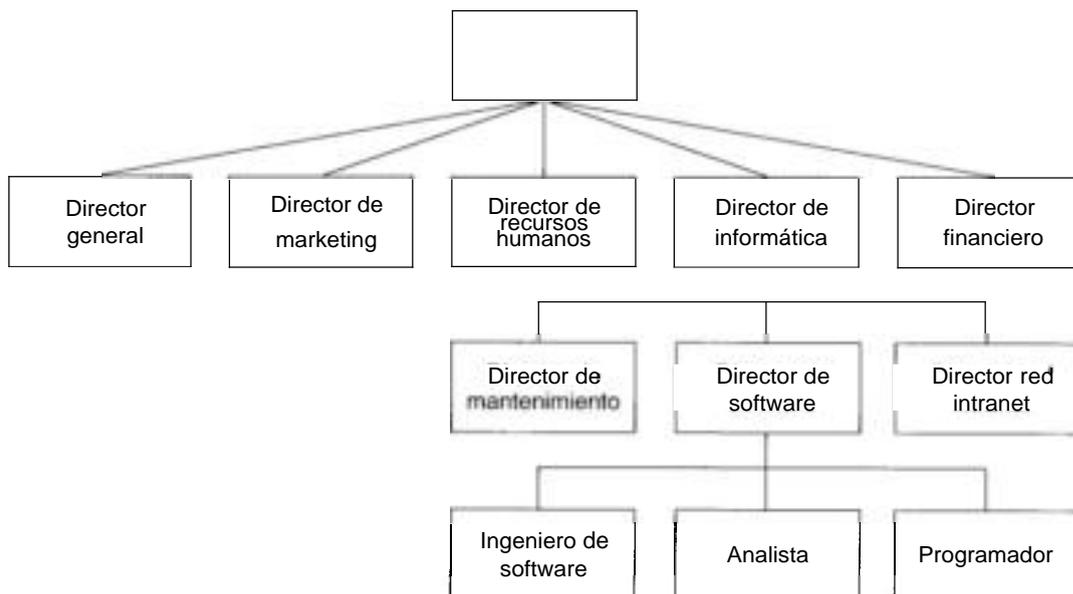


Figura 16.2. Estructura jerárquica tipo árbol.

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

**Definición 1:** Un **árbol** consta de un conjunto finito de elementos, llamados nodos y un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

**Definición 2:** Un **árbol** es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz**.
2. Los nodos restantes se dividen en  $n \geq 0$  conjuntos disjuntos tales que  $T_1, \dots, T_n$ , en donde cada uno de estos conjuntos es un árbol. A  $T_1, T_2, \dots, T_n$ , se les denomina **subárboles** del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz**. Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura 16.3 muestra un árbol.

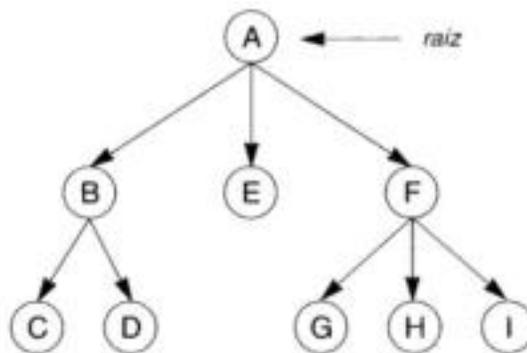


Figura 16.3. Árbol.

### Terminología

Además del raíz existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 16.4, el nodo A es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

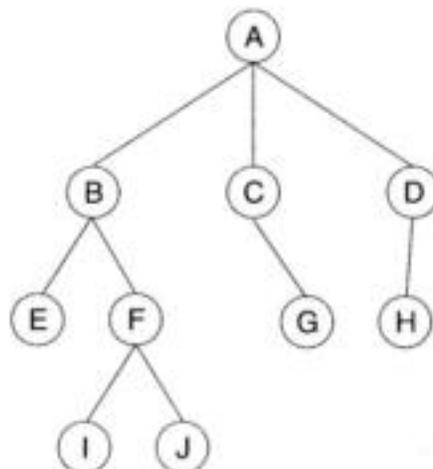


Figura 16.4. Árbol general.

Estos nodos sucesores se llaman **hijos**. Por ejemplo, el nodo B es el padre de los hijos E y F. El padre de H es el nodo D. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes** y el padre y abuelos de un nodo son sus **ascendientes**. Por ejemplo, los nodos E, F, I y J son descendientes de B. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos**. Un nodo sin hijos, tales como E, I, J, G y H se llaman nodos **hoja**.

El **nivel** de un nodo es su distancia al raíz. El raíz tiene una distancia cero de sí misma, por lo que se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y hermanos. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Fig. 16.5), C y D son hermanos, al igual que lo son G, H e I, pero D y G no son hermanos ya que ellos tienen diferentes padres.

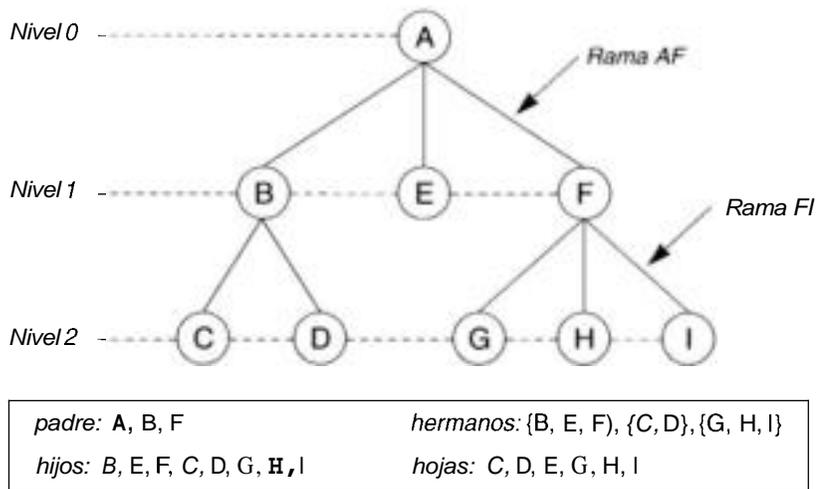


Figura 16.5. Terminología de árboles.

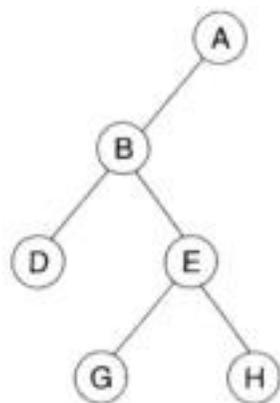
Existen varias formas de dibujar los atributos de los árboles y sus nodos. Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. En la Figura 16.5, el camino desde el raíz a la hoja I, se representa por AFI. Incluye dos ramas distintas AF y FI.

La altura o profundidad de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición la altura de un árbol vacío es 0. La Figura 16.5 contiene nodos en tres niveles : 0, 1 y 2. Su altura es 3.

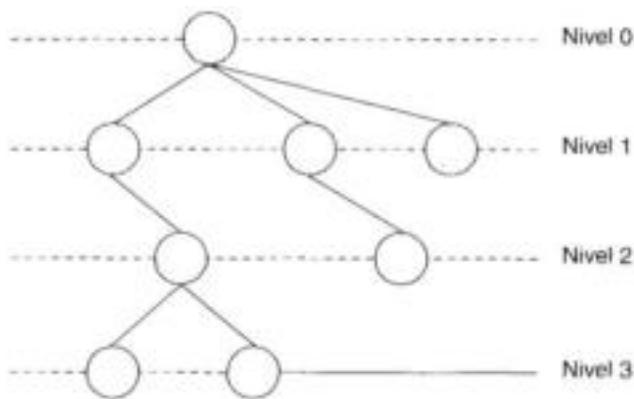
### Definición

El nivel de un nodo es su distancia desde el raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el raíz más uno.

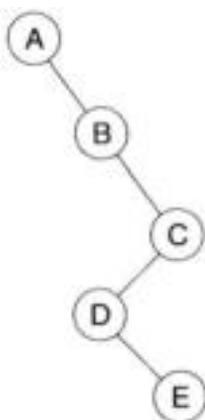
<sup>1</sup> También se suele definir la **profundidad** de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.



(a) Profundidad 4



(b) Profundidad 4



(c) Profundidad 5



Figura 16.6. Árboles de profundidades diferentes.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 16.5, BCD es un subárbol al igual que E y FGHI. Obsérvese que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol B se puede dividir en subárboles C y D mientras que el subárbol F contiene los subárboles G, H e I. Se dice que G, H, I, C y D son subárboles sin descendientes. El concepto de subárbol conduce a una *definición recursiva* de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien
2. Tiene un nodo determinado llamado *raíz* del que jerárquicamente descienden cero o más subárboles, **que** son también árboles.

Un árbol **está equilibrado** cuando, dado un número máximo de  $k$  hijos para cada nodo y la **altura del árbol**  $h$ , cada nodo de nivel  $l < h - 1$  tiene exactamente  $k$  hijos. El árbol está **equilibrado perfectamente** cuando cada nodo de nivel  $l < h$  tiene exactamente  $k$  hijos.

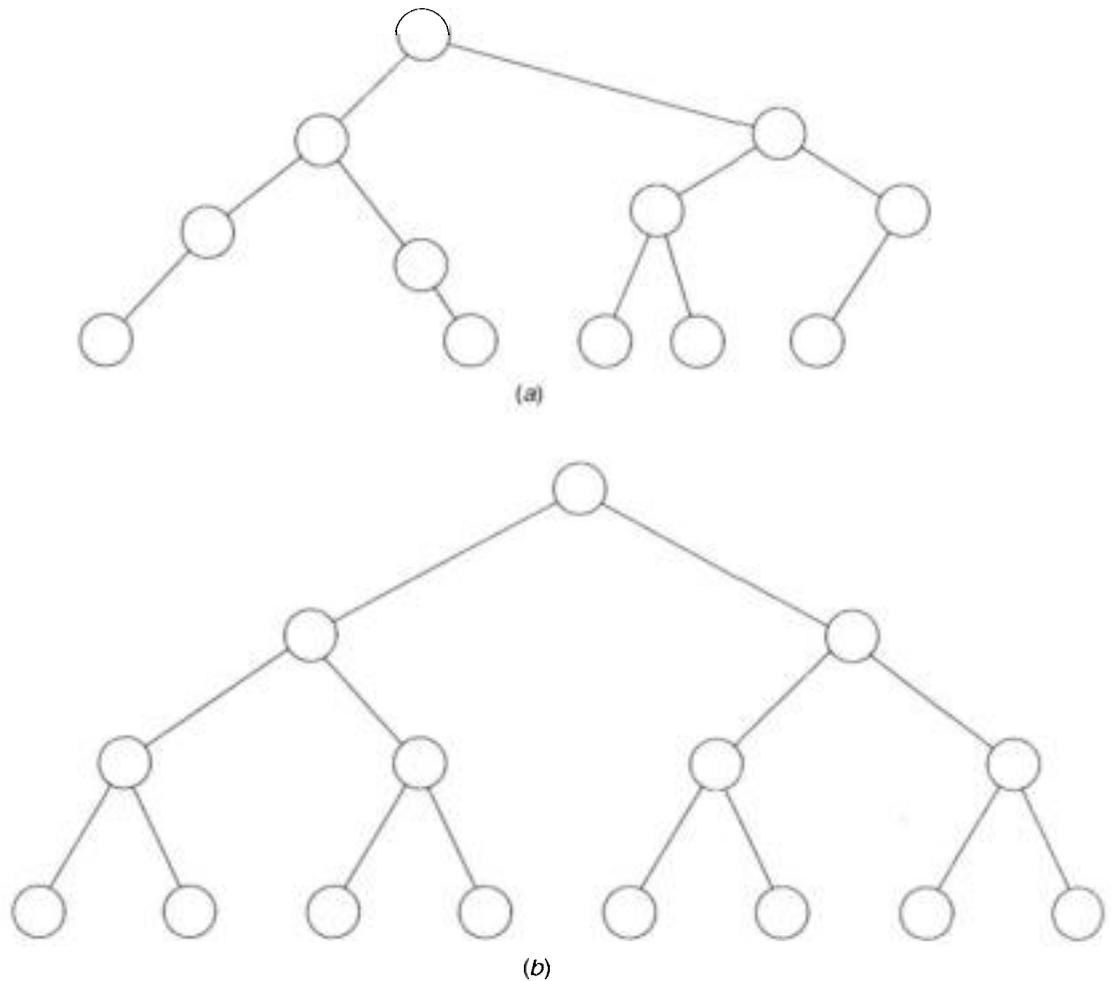


Figura 16.7. (a) Un árbol equilibrado; (b) Un árbol perfectamente equilibrado.

### 16.1.1. Representación de un árbol

Aunque un árbol se implementa en un lenguaje de programación como C mediante punteros, cuando se ha de representar en papel, existen tres formas diferentes de representación. La primera es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El término que se utiliza para esta notación es el de árbol general.

#### *Representación en niveles de profundidad*

Este tipo de representación es el utilizado para representar sistemas jerárquicos en modo texto o número en situaciones tales como facturación, gestión de *stocks* en almacenes, etc.

Por ejemplo, en las Figuras 16.8 y 16.9 se aprecia una descomposición de una computadora en sus diversos componentes en una estructura árbol. Otro ejemplo podría ser una distribución en árbol de las piezas de una tienda de recambios de automóviles distribuidas en niveles de profundidad según los números de parte o códigos de cada repuesto (motor, bujía, batería, piloto, faro, embellecedor, etc.).

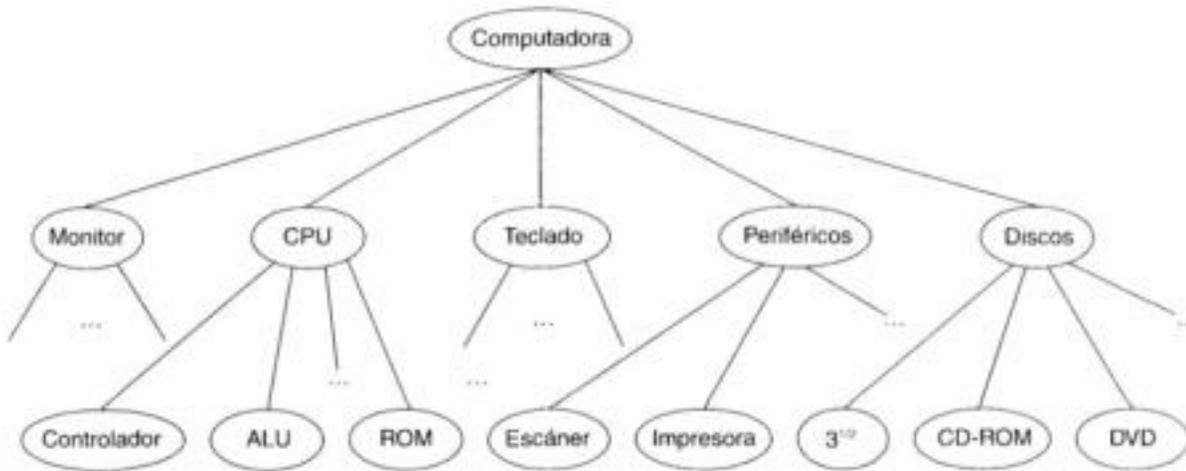


Figura 16.8. Árbol general (computadora).

| Número código | Descripción |
|---------------|-------------|
| 501           | Computadora |
| 501-11        | Monitor     |
| ...           |             |
| 501-21        | CPU         |
| 501-211       | Controlador |
| 501-212       | ALU         |
| ...           |             |
| 501-219       | ROM         |
| 501-31        | Teclado     |
| ...           |             |
| 501-41        | Periféricos |
| 501-411       | Escáner     |
| 501-412       | impresora   |
| 501-51        | Discos      |
| 501-511       | CD-ROM      |
| 501-512       | CD-RW       |
| 501-513       | DVD         |

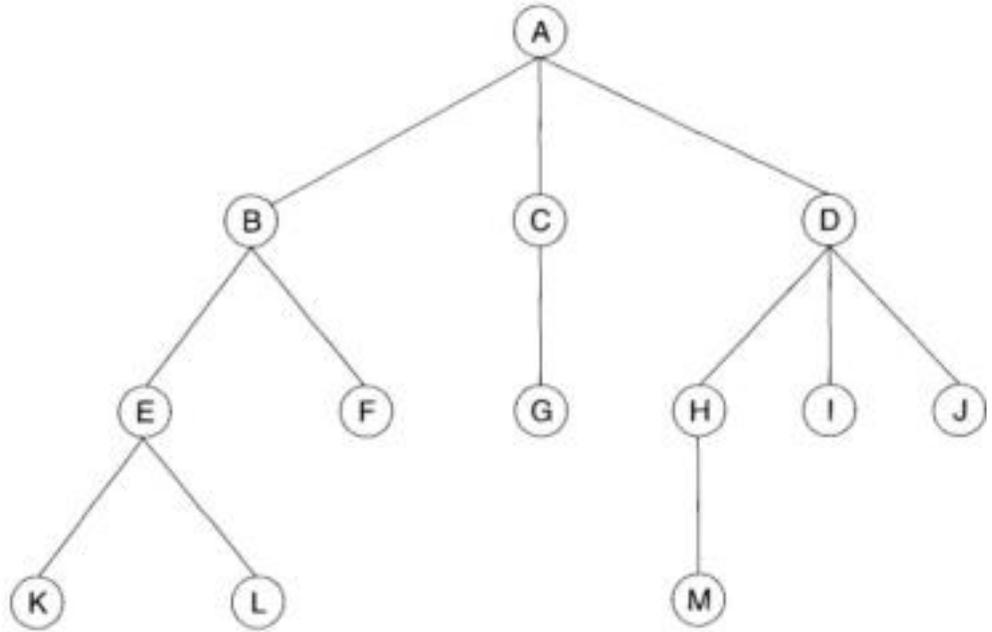
Figura 16.9. Árbol en nivel de profundidad (computadora).

Representación *de* lista

Otro formato utilizado para representar un árbol es la lista entre paréntesis. Ésta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis de la Figura 16.3 es:  $A(B(C, D), E, F, (G, H, I))$ .

**Ejemplo 16.1**

Convertir el árbol general siguiente en representación en lista.



La solución es A ( B ( E ( K, L ), F ), C ( G ), D ( H ( M ), I, J ) ) ).

## 16.2. RESUMEN DE DEFINICIONES

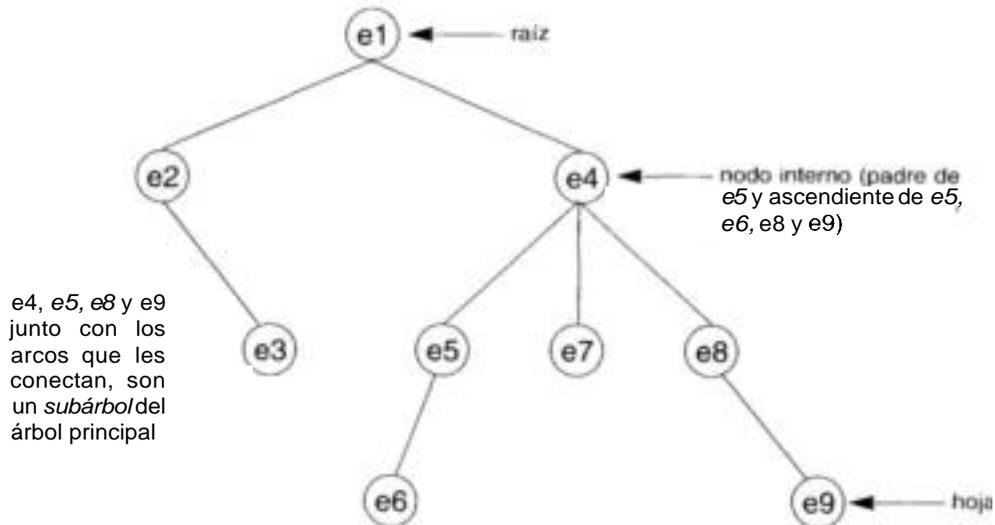
### 1. Dado un conjunto E de elementos:

- Un árbol puede estar *vacío*; es decir, no contiene ningún elemento,
- Un árbol *no vacío* puede constar de un único elemento  $e \in E$  denominado un **nodo**, o bien
- Un árbol consta de un nodo  $e \in E$ , conectado por arcos directos a un número finito de otros árboles.

### 2. Definiciones:

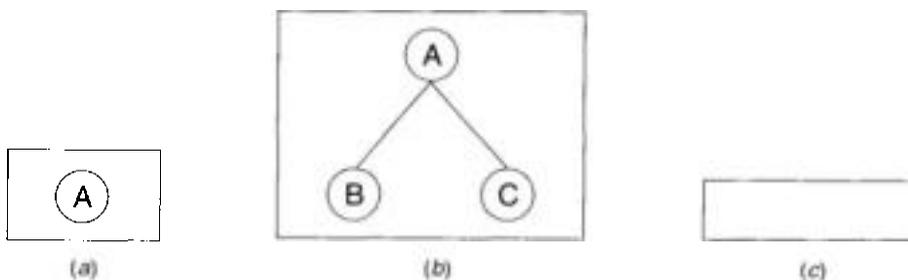
- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas**.
- Los **nodos terminales**, esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas**.
- Los nodos que no son hojas se denominan **nodos internos** o **nodos no terminales**.
- En un árbol una rama va de un nodo  $n_1$  a un nodo  $n_2$ , se dice que  $n_1$  es el padre de  $n_2$  y que  $n_2$  es un **hijo** de  $n_1$ .
- $n_1$  se llama **ascendiente** de  $n_2$  si  $n_1$  es el padre de  $n_2$  o si  $n_1$  es el padre de un ascendiente de  $n_2$ .
- $n_2$  se llama **descendiente** de  $n_1$  si  $n_1$  es un ascendiente de  $n_2$ .
- Un **camino** de  $n_1$  a  $n_2$  es una secuencia de arcos contiguos que van de  $n_1$  a  $n_2$ .
- La **longitud** de un camino es el número de arcos que contiene (en otras palabras el número de nodos - 1).
- El **nivel** de un nodo es la longitud del camino que lo conecta al raíz.
- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja.

- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es a su vez un árbol.
- Sea  $S$  un subárbol de un árbol  $A$ : si para cada nodo  $n$  de  $SA$ ,  $SA$  contiene también todos los descendientes de  $n$  en  $A$ .  $SA$  se llama un **subárbol completo** de  $A$ .
- Un árbol está **equilibrado** cuando, dado un número máximo  $K$  de hijos de cada nodo y la **altura del árbol**  $h$ , cada nodo de nivel  $k < h-1$  tiene exactamente  $K$  hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel  $l < h$  tiene exactamente  $K$  hijos.



### 16.3. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.



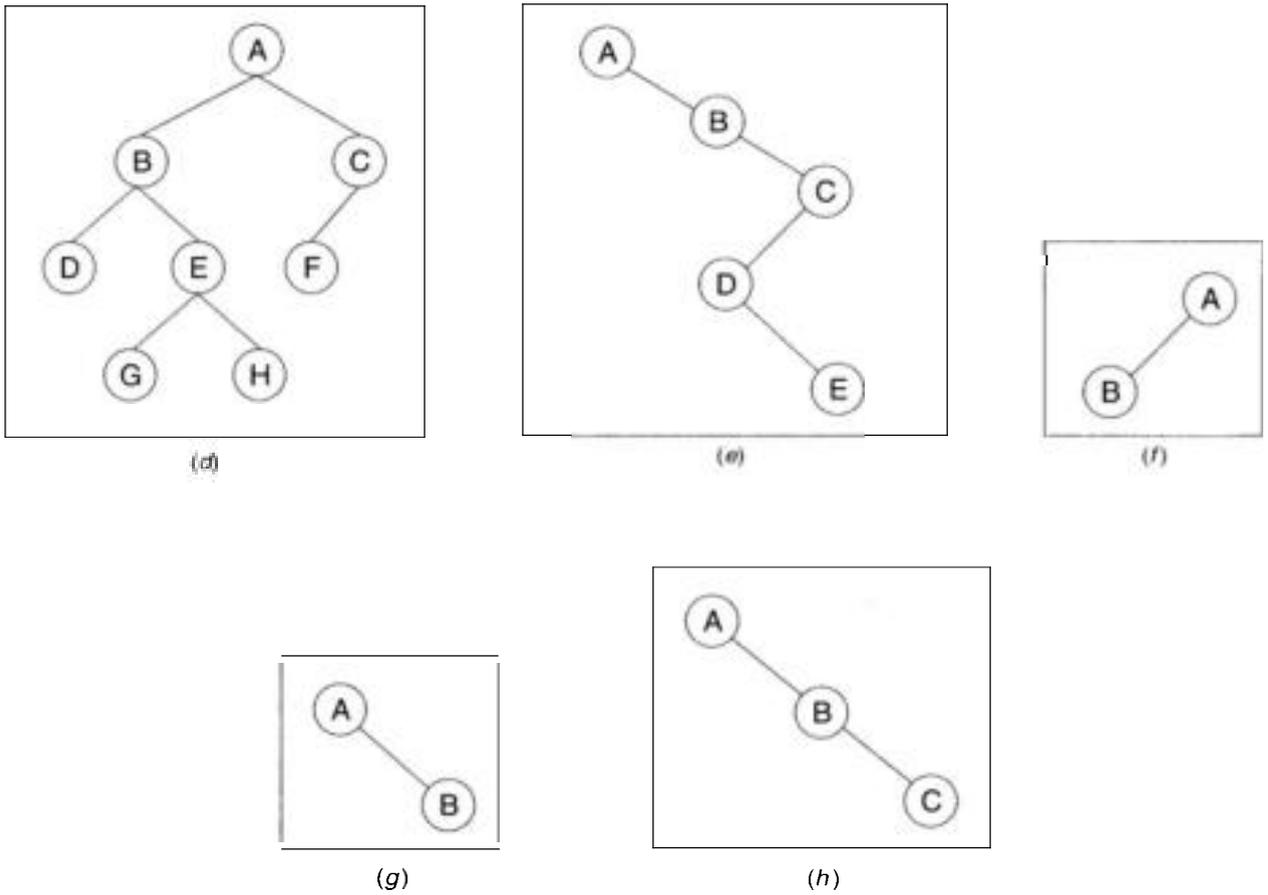


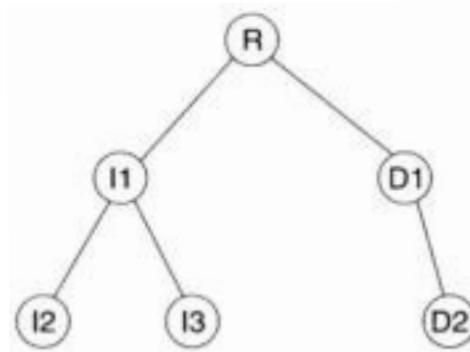
Figura 16.10. Árboles binarios.

**Nota**

Un árbol binario no puede tener más de dos subárboles.

Un árbol binario es una estructura recursiva. Cada nodo es el raíz de su propio subárbol y tiene hijos, que son raíces de árboles llamados los subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos:

- |                            |                                |
|----------------------------|--------------------------------|
| $\{R\}$                    | <i>Nodo raíz</i>               |
| $\{I_1, I_2, \dots, I_n\}$ | <i>Subárbol izquierdo de R</i> |
| $\{D_1, D_2, \dots, D_n\}$ | <i>Subárbol derecho de R</i>   |



Subárbol izquierdo

Figura 16.11. Árbol binario.

En cualquier nivel  $n$ , un árbol binario puede contener de 1 a 2 nodos. El número de nodos por nivel contribuye a la densidad del árbol.

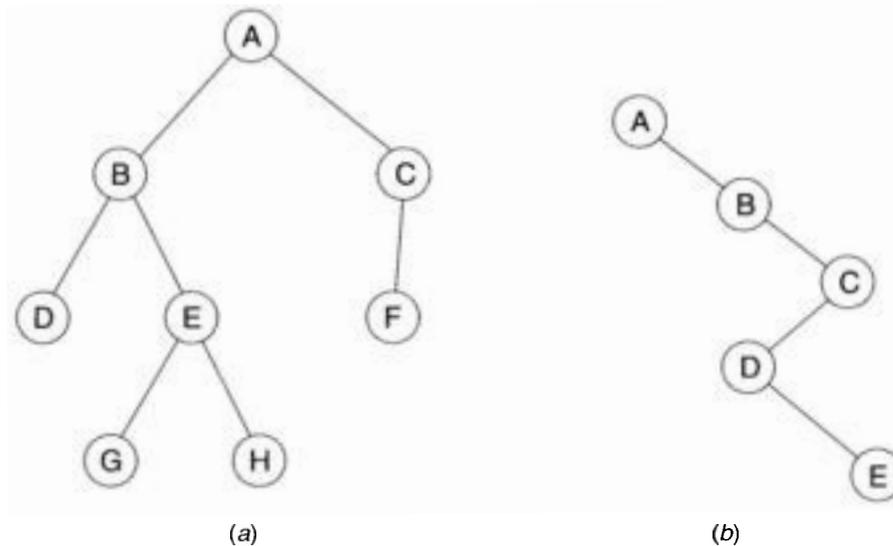


Figura 16.12. Árboles binarios: (a) profundidad 4; (b) profundidad 5.

En la Figura 16.12 (a) el árbol A contiene 8 nodos en una profundidad de 4, mientras que el árbol 16.12 (b) contiene 5 nodos y una profundidad 5. Este último caso es una forma especial, denominado **árbol degenerado**, en el que existe un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

### 16.3.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación

o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio**. Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si definimos la altura del subárbol izquierdo como  $H_L$ , y la altura del subárbol derecho como  $H_R$ , entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula:  $B = H_L - H_R$ .

Utilizando esta fórmula el equilibrio del nodo raíz los ocho árboles de la Figura 16.10 son (a) 0, (b) 0 por definición, (c) 0, (d) -1, (e) 4, (f) -1, (g) 1, (h) 2.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno (su factor de equilibrio es -1, 0, +1) y sus subárboles son también equilibrados.

### 16.3.2. Árboles binarios completos

Un árbol binario **completo** de profundidad  $n$  es un árbol en el que para cada nivel, del 0 al nivel  $n-1$  tiene un conjunto lleno de nodos y todos los nodos hoja a nivel  $n$  ocupan las posiciones más a la izquierda del árbol.

Un árbol binario completo que contiene  $2^n$  nodos a nivel  $n$  es un **árbol lleno**. Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno. La Figura 16.13 muestra un árbol binario completo; el árbol de la Figura 16.14 (b) se corresponde con uno lleno.

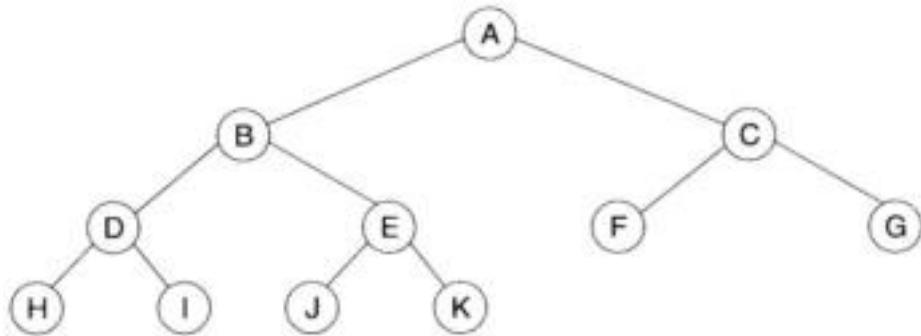


Figura 16.13. Árbol completo (profundidad 4).

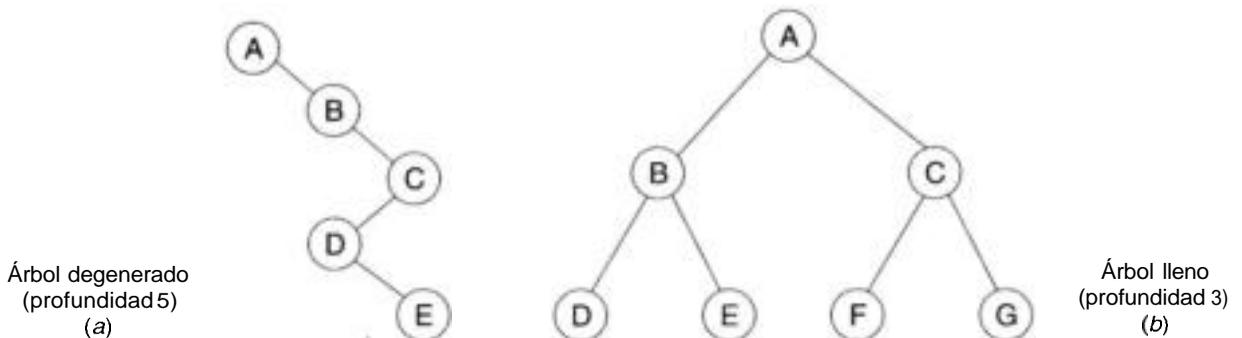


Figura 16.14. Clasificación de árboles binarios: (a) degenerado; (b) lleno.

El último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada. En la Figura 16.15 se muestran árboles llenos y completos.

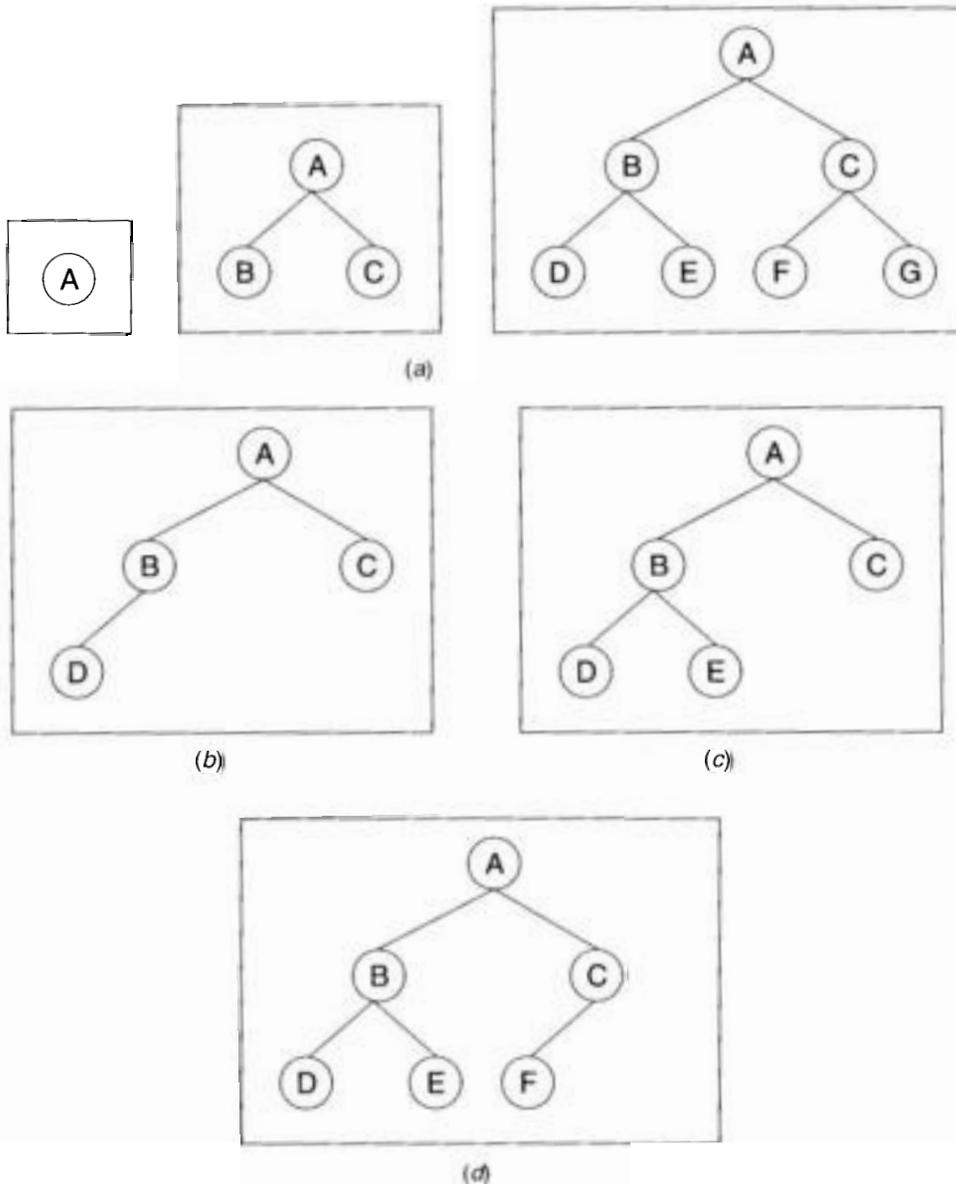


Figura 16.15. (a) Árboles llenos (en niveles 0, 1 y 2); (b),(c) y (d) árboles completos (en nivel 2).

Los árboles binarios y llenos de profundidad  $k+1$  proporcionan algunos datos matemáticos que es necesario comentar. En cada caso, existe un nodo ( $2^0$ ) al nivel 0 (raíz), dos nodos ( $2^1$ ) a nivel 1, cuatro nodos ( $2^2$ ) a nivel 2, etc. A través de los primeros  $k-1$  niveles hay  $2^k-1$  nodos.

$$1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$$

A nivel  $k$ , el número de nodos adicionados para un árbol completo está en el rango de un mínimo de 1 a un máximo de  $2^k$  (lleno). Con un árbol lleno, el número de nodos es

$$1 + 2 + 4 + \dots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

El número de nodos  $n$  en un árbol binario completo de profundidad  $k+1$  (0 a  $k$  niveles) cumple la desigualdad

$$2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$$

Aplicando logaritmos a la ecuación con desigualdad anterior

$$k \leq \log_2(n) < k + 1$$

Se deduce que la altura o profundidad de un árbol binario completo de  $n$  nodos es:

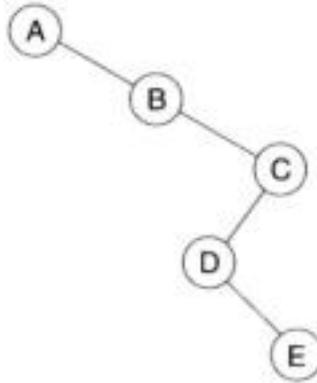
$$h = \lceil \log_2(n + 1) \rceil \quad (\text{parte entera de } \log_2(n + 1))$$

Por ejemplo, un árbol lleno de profundidad 4 (niveles 0 a 3) tiene  $2^4 - 1 = 15$  nodos

### Ejemplo 16.2

Calcular la profundidad máxima y mínima de un árbol con 5 nodos.

La profundidad máxima de un árbol con 5 nodos es 5



La profundidad mínima  $n$  (número de niveles más uno) de un árbol con 5 nodos es

$$k \leq \log_2(5) < k + 1$$

$$\log_2(5) = 2.32 \text{ y la profundidad } n = 3$$

### Ejemplo 16.3

La profundidad de un árbol degenerado con  $n$  nodos es  $n$ , dado que es la longitud del camino más largo (raíz a nodo) más 1.

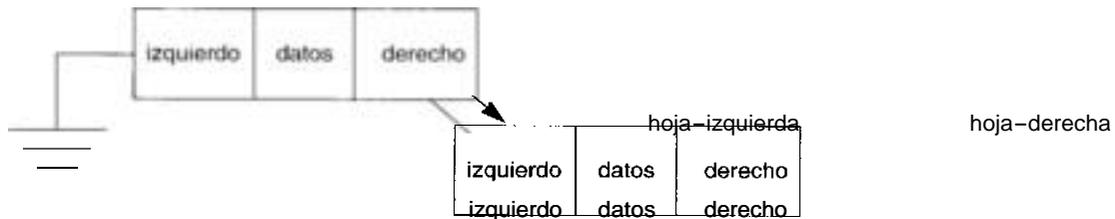
En el árbol binario completo con  $n$  nodos, la profundidad del árbol es el valor entero de  $\log_2(n + 1)$ , que es a su vez la distancia del camino más largo desde el raíz a un nodo más uno.

Suponiendo que el árbol tiene  $n = 10.000$  elementos, el camino más largo es

$$\text{int}(\log 10000) + 1 = \text{int}(13.28) + 1 = 14$$

### 16.4. ESTRUCTURA DE UN ÁRBOL BINARIO

La estructura de un árbol binario se construye con nodos. Cada nodo debe contener el campo dato (datos a almacenar) y dos campos punteros, uno al subárbol izquierdo y otro al subárbol derecho, que se conocen como **puntero izquierdo (izquierdo, izdo)** y **puntero derecho (derecho, dcho)** respectivamente. Un valor NULL indica un árbol vacío.



El algoritmo correspondiente a la estructura de un árbol es el siguiente:

```

Nodo
 subarbolIzquierdo < puntero a Nodo>
 datos < Tipodato >
 subarbolDerecho < puntero a Nodo>
Fin Nodo

```

La Figura 16.16 muestra un árbol binario y su estructura en nodos:

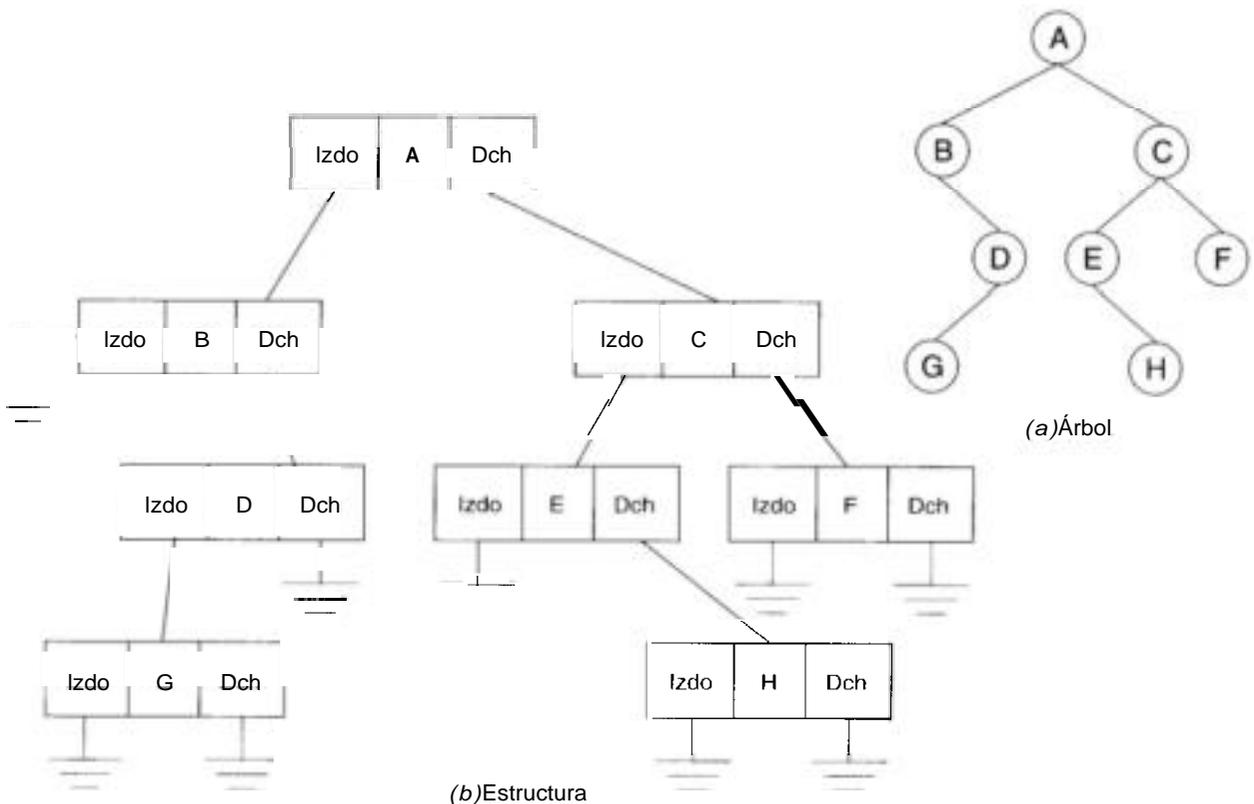
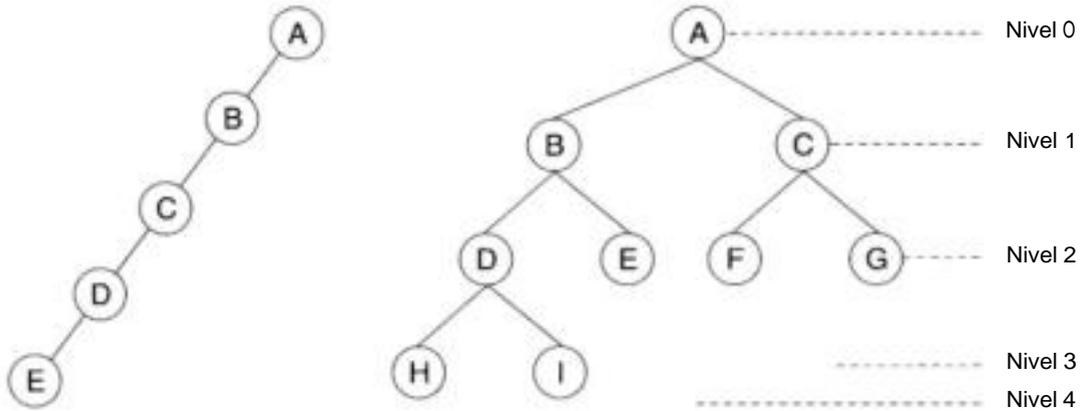


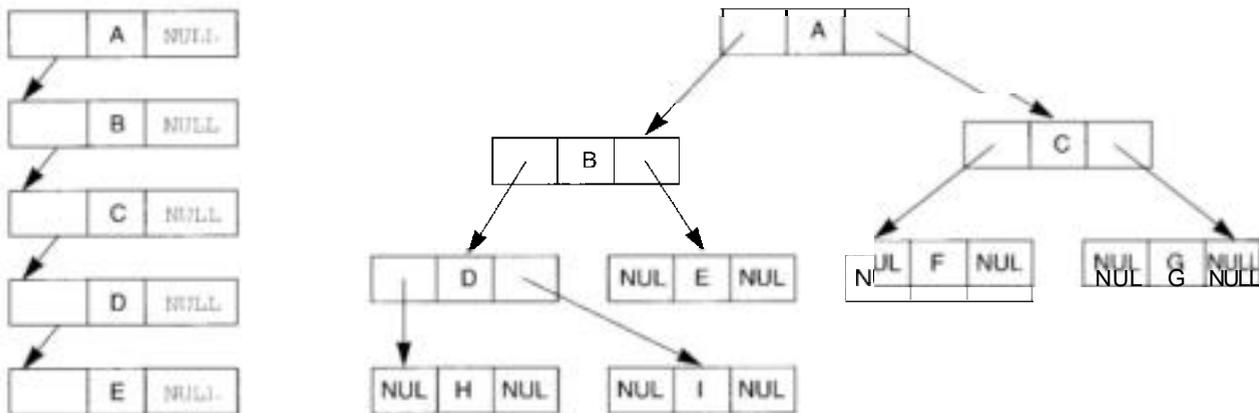
Figura 16.16. Árbol binario y su estructura en nodos

**Ejemplo 16.4**

Representar la estructura en nodos de los dos árboles binarios de raíz A:



La representación enlazada de estos dos árboles binarios es:

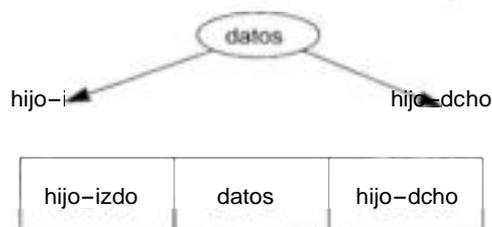


**16.4.1. Diferentes tipos de representaciones en C**

Los nodos pueden ser representados con la estructura `struct`. Suponiendo que el nodo tiene los campos `Datos`, `Izquierdo` y `Derecho`.

*Representación 1*

```
typedef struct nodo "puntero-arbol;
typedef struct nodo {
 int datos;
 puntero-arbol hijo-izdo, hijo-dcho;
};
```



### Representación 2

```
typedef int TipoElemento; /* Puede ser cualquier tipo */
struct Nodo {
 TipoElemento Info;
 struct Nodo *hijo_izdo, *hijo_dcho;
};
typedef struct Nodo ElementoDeArbolBin;
typedef ElementoDeArbolBin *ArbolBinario;
```

Para crear un nodo de un árbol binario, con la representación 2, se reserva memoria para el nodo, se asigna el dato al campo `info` y se inicializa los punteros `hijo_izdo`, `hijo_dcho` a `NULL`.

```
ArbolBinario CrearNodo(TipoElemento x)
{
 ArbolBinario a;
 a = (ArbolBinario) malloc(sizeof(ElementoDeArbolBin));
 a -> Info = x;
 a -> hijo-dcho = a -> hijo-izdo = NULL;
 return a;
}
```

Si por ejemplo se desea crear un árbol binario de raíz 9, rama izquierda 7 y rama derecha 11:

```
ArbolBinario raiz;
raiz = CrearNodo(9);
raiz -> hijo-izdo = CrearNodo(7);
raiz -> hijo-dcho = CrearNodo(11);
```

## 16.5. OPERACIONES EN ÁRBOLES BINARIOS

Una vez que se tiene creado un árbol binario, se pueden realizar diversas operaciones sobre él. El hacer uso de una operación u otra dependerá de la aplicación que se le quiera dar al árbol. Algunas de las operaciones típicas que se realizan en árboles binarios son:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión<sup>2</sup>, evaluar la expresión.
- Si es un árbol de expresión, obtener la forma de paréntesis de la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El

<sup>2</sup> En el apartado siguiente se estudia el importante concepto de *árbol de expresión*.

recorrido de un árbol es la operación de visita al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más tarde.

## 16.6. ÁRBOLES DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresión*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos que siguen unas reglas prescritas). Un *token* puede ser o bien un operando o bien un operador.

La Figura 16.17 representa la expresión infija  $a * (b + c) + d$  y su árbol de expresión. En una primera

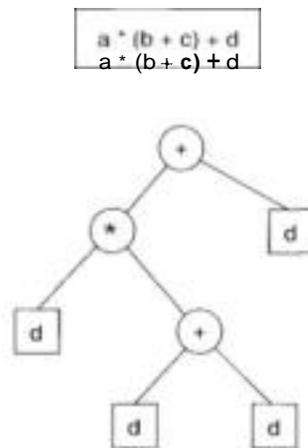


Figura 16.17. Una expresión infija y su árbol de expresión.

observación vemos que los paréntesis no aparecen en el árbol.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz e internos son operadores.
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguaje de programación. La Figura 16.18 muestra un árbol binario de expresiones para la expresión aritmética  $(a + b) * c$ .

Obsérvese que los paréntesis no se almacenan en el árbol pero están implicados en la forma del

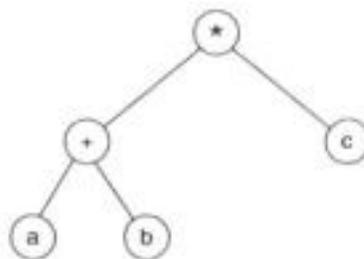


Figura 16.18. Árbol binario de expresiones que representa  $(a + b) * c$ .

árbol. Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión por un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho son los operandos izquierdo y derecho respectivamente. Cada operando puede ser una letra ( $x$ ,  $y$ ,  $a$ ,  $b$ , etc.) o una subexpresión representada como un subárbol. En la Figura 16.19 se puede ver como el operador que está en la raíz es  $*$ , su subárbol izquierdo representa la subexpresión  $(x + y)$  y su subárbol derecho representa la subexpresión  $(a - b)$ . El nodo raíz del subárbol izquierdo contiene el operador  $(+)$  de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador  $(-)$  de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

Utilizando el razonamiento anterior, se puede escribir la expresión almacenada en la Figura 16.20 como

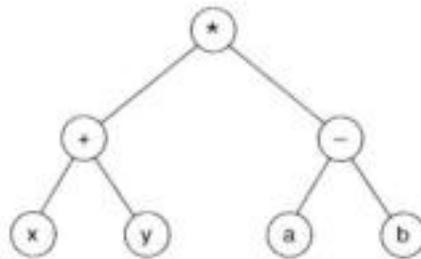


Figura 16.19. Árbol de expresión  $(x+y) * (a-b)$

$$(x * (y - z)) + (a - b)$$

en donde se han insertado paréntesis alrededor de subexpresiones del árbol (la operación  $y - z$ , subexpresión más interna, tiene el nivel de prioridad mayor).

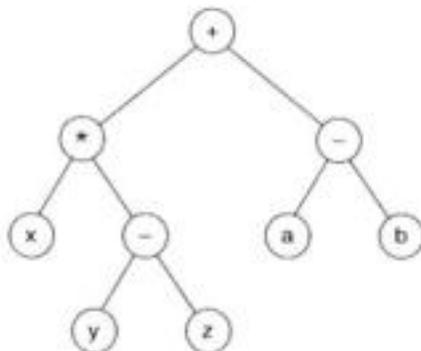
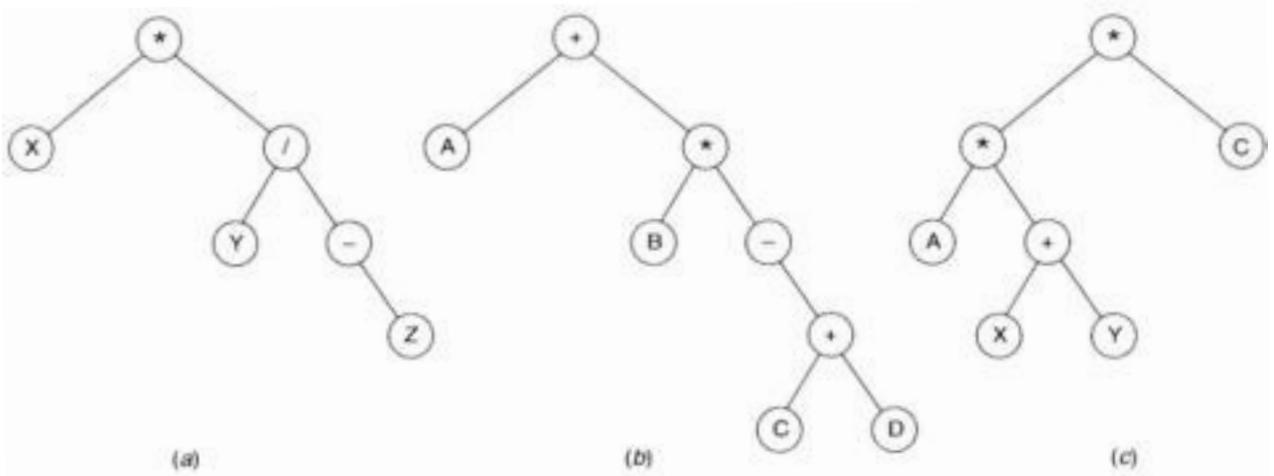


Figura 16.20. Árbol de expresión  $(x * (y - z)) + (a - b)$ .

**Ejemplo 16.5**

Deducir las expresiones que representan los siguientes árboles binarios.



**Soluciones**

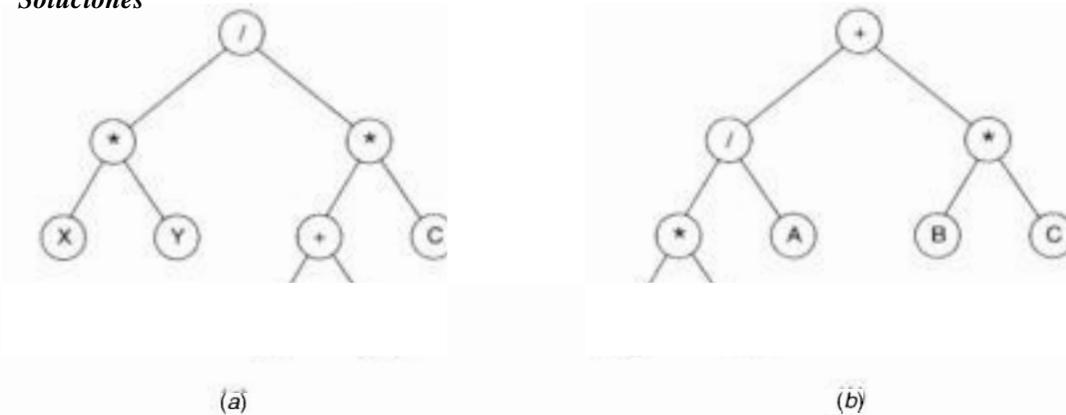
- (a)  $X * (Y / -Z)$
- (b)  $A + (B * - (C + D))$
- (c)  $(A * (X + Y)) * C$

**Ejemplo 16.6**

Dibujar la representación en árbol binario de cada una de las siguientes expresiones.

- (a)  $X * Y / (A + B) * C$
- (b)  $X * Y / A + B * C$

**Soluciones**



**16.6.1. Reglas para la construcción de árboles de expresión**

Los árboles de expresiones se utilizan en las coinputadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es uno que lee una expresión completa que contiene paréitesis en la misma. Una expresión con paréntesis es aquella en que

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

Por consiguiente  $(4 + (5 * 6))$  es un ejemplo de una expresión completa entre paréntesis. Su valor es 34. Si se desean cambiar las prioridades, se escribe  $((4 + 5) * 6)$ , su valor es 54. A fin de ver la prioridad en las expresiones, considérese la expresión

$$(4 * 5) + 6 / 7 - (8 + 9)$$

Los operadores con prioridad más alta son  $*$  y  $/$ ; es decir,

$$(4 * 5) + (6 / 7) - (8 + 9)$$

El orden de los operadores aquí es  $+$  y  $-$ . Por consiguiente, se puede escribir

$$((4 * 5) + (6 / 7)) - (8 + 9)$$

Por último la expresión completa entre paréntesis será

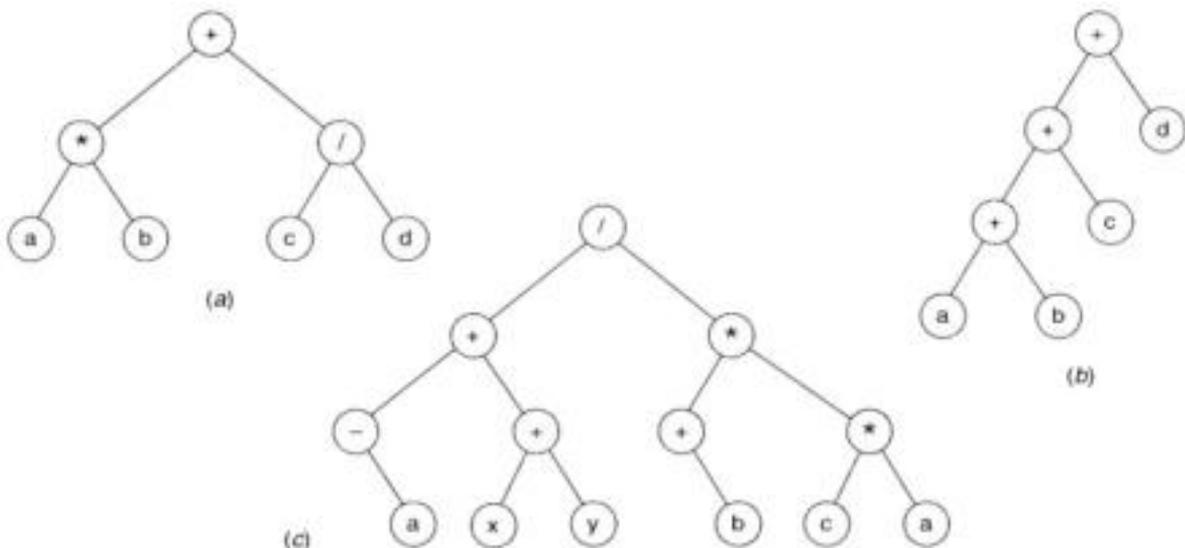
$$(((4 * 5) + (6 / 7)) - (8 + 9))$$

El algoritmo para la construcción de un árbol de expresión es:

1. La primera vez que se encuentra un paréntesis a izquierda, crea un nodo y lo hace en el raíz. Se llama a éste, el *nodo actual* y se sitúa su puntero en una pila.
2. Cada vez que se encuentre un nuevo paréntesis a izquierda, crear un nuevo nodo. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho. Hacer el nuevo nodo el nodo actual y situar su puntero en una pila.
3. Cuando se encuentra un operando, crear un nuevo nodo y asignar el operando a su campo de datos. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho.
4. Cuando se encuentra un operador, sacar un puntero de la pila y situar el operador en el campo datos del nodo del puntero.
5. Ignorar paréntesis derecho y blancos.

### Ejemplo 16.7

Calcular las expresiones correspondientes de los árboles de expresión.



Las soluciones correspondientes son:

**u.**  $(a * b) + (c / d)$

**c.**  $((-a) + (x + y)) / ((+b) * (c * a))$

**b.**  $((a + b) + c) + d$

**Ejercicio 16.1** (a realizar por el lector)

Dibujar los árboles binarios de expresión correspondiente a cada una de las siguientes expresiones:

(v)  $(a + b) / (c - d * e) + e + 9 * h/a$

(h)  $-x -y * z + (a + b + c / d * e)$

(c)  $((a + b) > (c - e)) || a < f \&\& (x < y || y > z)$

### 16.7. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita **recorrer** el árbol o **visitar** los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el raíz viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

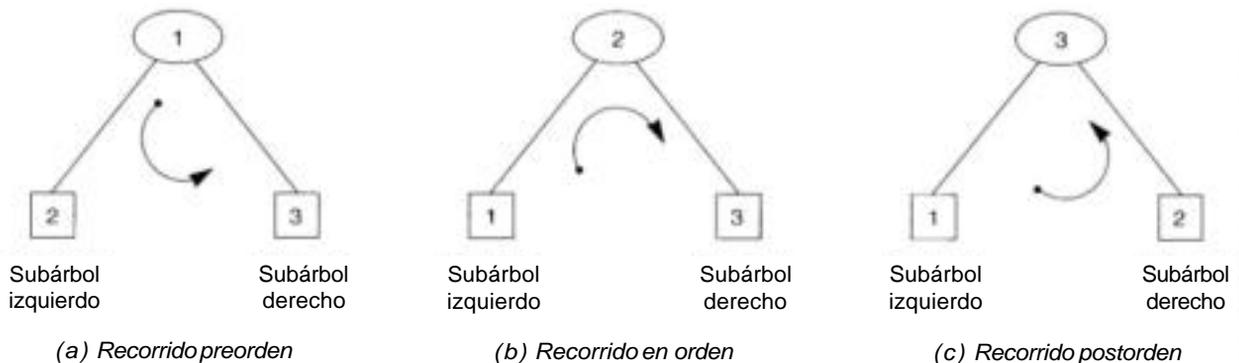
Un **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

En el **recorrido en profundidad**, el proceso exige un camino desde el raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En otras palabras, en el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

El **recorrido** de un árbol supone visitar cada nodo sólo una vez.

Dado un árbol binario que consta de un raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 16.21.



**Figura 16.21.** Recorridos de árboles binarios

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz (**N**), para el subárbol izquierdo (**I**) y para el subárbol derecho (**D**).

Según sea la estrategia a seguir, los recorridos se conocen como **enorden** (*inorder*), **preorden** (*preorder*) y **postorden** (*postorder*)

|                  |                                         |
|------------------|-----------------------------------------|
| <b>Preorden</b>  | (nodo-izquierdo-derecho) ( <b>NID</b> ) |
| <b>Enorden</b>   | (izquierdo-nodo-derecho) ( <b>IND</b> ) |
| <b>Postorden</b> | (izquierdo-derecho-nodo) ( <b>IDN</b> ) |

### 16.7.1. Recorrido *preorden*

El recorrido *preorden*<sup>2</sup> (**NID**) conlleva los siguientes pasos, en los que el raíz va antes que los subárboles:

1. Recorrer el raíz (**N**).
2. Recorrer el subárbol izquierdo (**I**) en *preorden*.
3. Recorrer el subárbol derecho (**D**) en *preorden*.

Dado las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero, se procesa la raíz, a continuación el subárbol izquierdo y a continuación el subárbol derecho. Para procesar el subárbol izquierdo, se hace una llamada recursiva al procedimiento *preorden* y luego se hace lo mismo con el subárbol derecho. El algoritmo recursivo correspondiente para un árbol T es:

```

si T no es vacío entonces
 inicio
 ver los datos en el raíz de T
 Preorden (subarbol izquierdo del raíz de T)
 Preorden (subarbol derecho del raíz de T)
 fin

```

### Regla

En el recorrido *preorden*, el raíz se procesa antes que los subárboles izquierdo y derecho.

Si utilizamos el recorrido *preorden* del árbol de la Figura 16.22 se visita primero el raíz (nodo A). A continuación se visita el subárbol izquierdo de A, que consta de los nodos B, D y E. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el orden **NID**. Por consiguiente, se visita primero el nodo B, después D (izquierdo) y, por último, E (derecho).

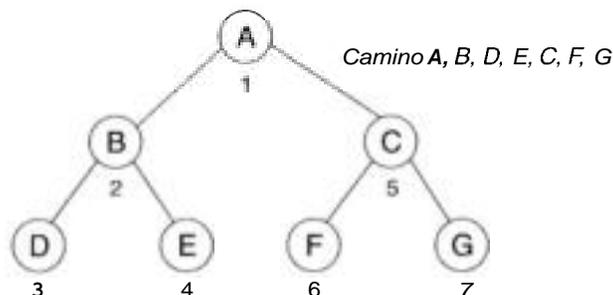


Figura 16.22. Recorrido *preorden* de un árbol binario.

<sup>2</sup> El nombre *preorden*, viene del prefijo latino *pre* que significa «ir antes»

A continuación se visita el subárbol derecho de A, que es un árbol que contiene los nodos C, F y G. De nuevo siguiendo el orden NID, se visita primero el nodo C, a continuación F (izquierdo) y, por último, G (derecho). En consecuencia el orden del recorrido preorden para el árbol de la Figura 16.22 es A-B-D-E-C-F-G.

Un refinamiento del algoritmo es:

```

algoritmo preOrden (val raiz <puntero nodos>)
Recorrer un arbol binario en secuencia nodo-izdo-dcho
Pre raiz es el nodo de entrada del árbol o subárbol
Post cada nodo se procesa en orden
1 si (raiz no es nulo)
 1 procesar (raiz)
 2 preOrden (raiz -> subarbolIzdo)
 3 preOrden (raiz -> subarbolDcho)
2 return

```

La función preorden muestra el código fuente en C del algoritmo ya citado anteriormente. El tipo de los datos es entero.

```

typedef int TipoElemento;
struct nodo {
 TipoElemento datos;
 struct nodo *hijo_izdo, *hijo_dcho;
};
typedef struct nodo Nodo;

void preorden (Nodo *p)
{
 if (p)
 {
 printf("%d ", p -> datos);
 PreOrden(p -> hijo_izdo);
 PreOrden(p -> hijo_dcho);
 }
}

```

### Gráficas de las llamadas recursivas de preorden

El recorrido recursivo de un árbol se puede mostrar gráficamente por dos métodos distintos: 1) *paseo preorden* del árbol; 2) recorrido algorítmico.

Un medio gráfico para visualizar el recorrido de un árbol es imaginar que se está dando un «paseo» alrededor del árbol comenzando por la raíz y siguiendo el sentido contrario a las agujas del reloj, un nodo a continuación de otro sin pasar dos veces por el mismo nodo. El camino señalado por una línea continua que comienza en el nodo 1 (Fig. 16.21) muestra el recorrido preorden completo. En el caso de la Figura 16.22 el recorrido es A B D E C F G.

El otro medio gráfico de mostrar el recorrido algorítmico recursivo es similar a las diferentes etapas del algoritmo. Así la primera llamada procesa la raíz del árbol A. A continuación se llama recursivamente a procesar subárbol izquierdo, procesa el nodo B. La tercera llamada procesa el nodo D, que es un subárbol formado por un único nodo. En ese punto, se llama en preorden, con un puntero nulo, que produce un retorno inmediato al subárbol D para procesar a su subárbol derecho. Debido a que el subárbol derecho de D es también nulo, se vuelve al nodo B de modo que va a procesar (visitar) su subárbol derecho, E. Después de procesar el nodo E, se hacen dos llamadas más, una con el puntero izquierdo *null* de E y otra con su puntero derecho *null*. Como el subárbol B ha sido totalmente procesado, se vuelve a la raíz del árbol y se procesa su subárbol derecho, C. Después de procesar C, llama para procesar su subárbol izquierdo F. Se hacen dos llamadas con *null*, vuelve al nivel donde está el nodo C para procesar su rama derecha G. Aún se realizan dos llamadas más, una al subárbol izquierdo *null* y otra al subárbol derecho. Entonces se retorna en el árbol, se concluye el recorrido del árbol.

### 16.7.2. Recorrido *enorden*

El recorrido **en orden** (*inorden*) procesa primero el subárbol izquierdo, después el raíz y a continuación el subárbol derecho. El significado de *in* es que la raíz se procesa entre los subárboles. Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (I) en *inorden*.
2. Visitar el nodo raíz (N).
3. Recorrer el subárbol derecho (D) en *inorden*.

El algoritmo correspondiente es:

```

Enorden(A)
si el arbol no esta vacio entonces
 inicio
 Recorrer el subarbol izquierdo
 Visitar el nodo raiz
 Recorrer el subarbol derecho
 fin

```

Un refinamiento del algoritmo es:

```

algoritmo enOrden (val raiz <puntero a nodos>)
Recorrer un árbol binario en la secuencia izquierdo-nodo-derecho
 pre raíz en el nodo de entrada de un árbol o subárbol
 post cada nodo se ha de procesar en orden
1 si (raíz no es nulo)
 1 enorden (raíz -> subarbol Izquierdo)
 2 procesar (raíz)
 3 enOrden (raíz->subarbolDerecho)
2 retorno
fin enorden

```

En el árbol de la Figura 16.23, los nodos se han numerado en el orden en que son visitados durante el recorrido *enorden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra B. Este subárbol consta de los nodos B, D y E y es a su vez otro árbol con el nodo B como raíz, por lo que siguiendo el orden IND, se visita primero D, a continuación B (nodo raíz) y, por último, E (derecha). Después de la visita a este subárbol izquierdo se visita el nodo raíz A y, por último, se visita el subárbol derecho que consta de los nodos C, F y G. A continuación, siguiendo el orden IND para el subárbol derecho, se visita primero F, después C (nodo raíz) y, por último, G. Por consiguiente, el orden del recorrido *inorden* de la Figura 16.23 es D-B-E-A-F-C-G.

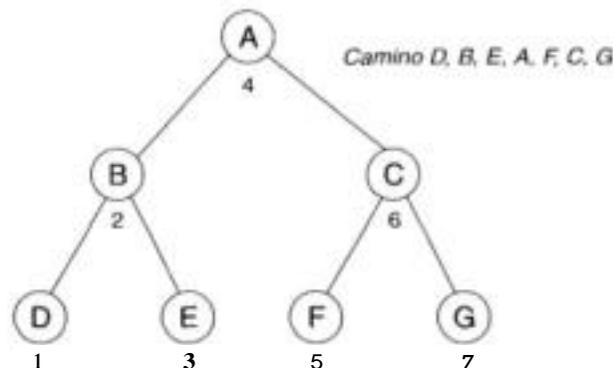


Figura 16.23. Recorrido *enorden* de un árbol binario.

La siguiente función visita y escribe el contenido de los nodos de un árbol binario de acuerdo al recorrido *EnOrden*. La función tiene como parámetro un puntero al nodo raíz del árbol.

```
void enorden (Nodo *p)
{
 if (p)
 {
 enorden(p -> hijo-izqdo); /* recorrer subárbol izquierdo */
 printf("%d ", p -> datos); /* visitar la raíz */
 enorden (p -> hijo-dcho); /* recorrer subárbol derecho */
 }
}
```

### 16.7.3. Recorrido *postorden*

El recorrido *postorden* (**IDN**) procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación se procesa su subárbol derecho. **Por** último se procesa el nodo raíz. Las etapas del algoritmo son:

1. Recorrer el subárbol izquierdo (**I**) en *postorden*.
2. Recorrer el subárbol derecho (**D**) en *postorden*.
3. Visitar el nodo raíz (**N**).

El algoritmo recursivo **para** un árbol A es:

```
si A no esta vacio entonces
 inicio
 Postorden (subarbol izquierdo del raíz de A)
 Postorden (subarbol derecho del raíz de A)
 Visitar la raíz de A
 fin
```

El refinamiento del algoritmo es:

```
algoritmo postorden (val raiz <puntero a nodo>)
```

**Recorrer un árbol binario en secuencia izquierda-derecha-nodo**

pre *raíz es el nodo de entrada de un árbol a un subárbol*  
post *cada nodo ha sido procesado en orden*

```
1 Si (raíz no es nulo)
 1postOrden (raíz -> SubarbolIzdo)
 2postOrden (raíz -> SubarbolDcho)
 3procesar (raiz)
2 retorno
fin postorden
```

Si se utiliza el recorrido *postorden* del árbol de la Figura 16.24, se visita primero el subárbol izquierdo A. Este subárbol consta de los nodos B, D y E y siguiendo el orden **IDN**, se visitará primero D (izquierdo), luego E (derecho) y, por último, B (nodo). A continuación, se visita el subárbol derecho A que consta de los nodos C, F y G. Siguiendo el orden **IDN** para este árbol, se visita primero F (izquierdo), después G (derecho) y, por último, C (nodo). Finalmente se visita el raíz A (nodo). Así el orden del recorrido *postorden* del árbol de la Figura 16.24 es D-E-B-F-G-C-A.

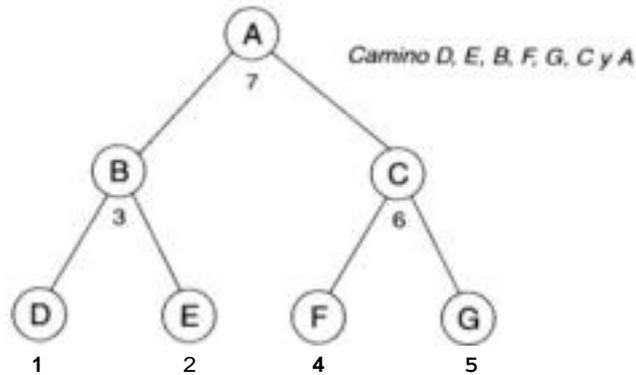


Figura 16.24. Recorrido *postorden* de un árbol binario.

La función *postorden* que implementa en C el código fuente del algoritmo correspondiente

```

void postorden (Nodo *p)
{
 if (p)
 {
 postorden (p -> hijo_izqdo);
 postorden (p -> hijo_dcho);
 printf("%d ", p -> datos);
 }
}

```

### Nota de programación modular

La visita al nodo raíz del árbol que se representa mediante una sentencia `printf()` podría representarse también con una función *visitar*

```

void visitar (Nodo *p)
{
 printf("%d ", p -> datos);
}

```

La función *postorden* quedaría así:

```

void postorden (Nodo *p)
{
 if (p)
 {
 postorden (p -> hijo_izqdo);
 postorden (p -> hijo_dcho);
 visitar (p);
 }
}

```

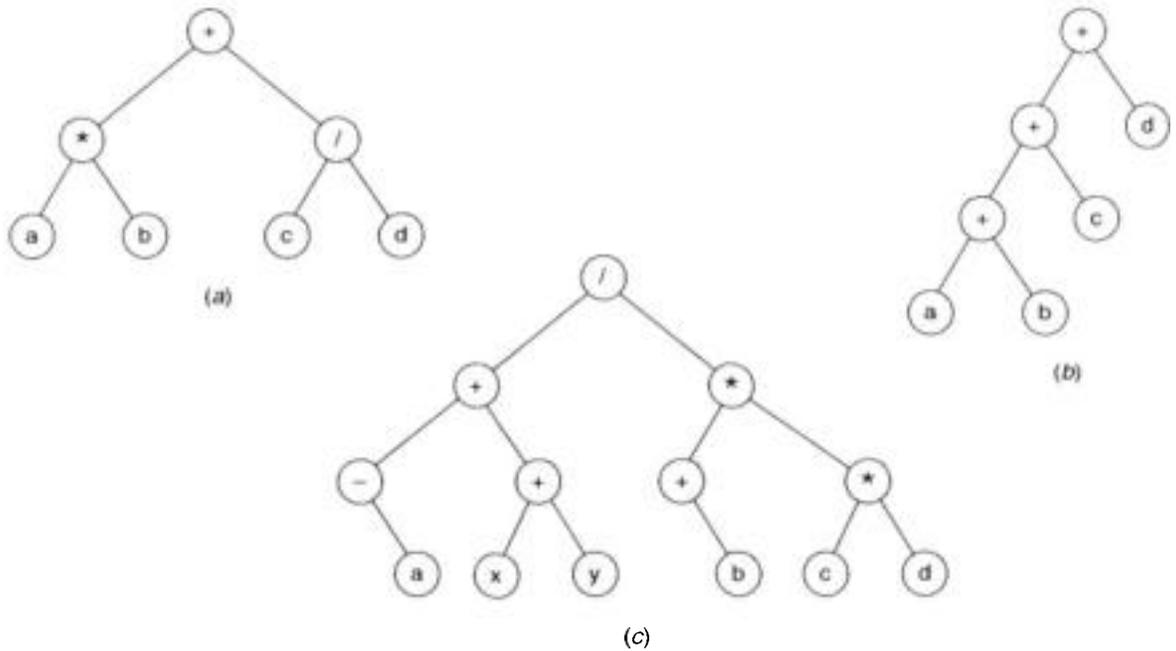


Figura 16.25. Árboles de expresión.

**Ejercicio 16.2**

Si la función `visitar()` se reemplaza por la `.sentencia`.

```
printf("%d ", t -> dato);
```

deducir los elementos de los árboles binarios siguientes en cada uno de los tres recorridos fundamentales.

Los elementos de los árboles binarios listados en preorden, enorden y postorden.

|           | Árbola  | Árbol b | Árbol c                  |
|-----------|---------|---------|--------------------------|
| PreOrden  | +*ab/cd | +++abcd | /+- a + xy * +b * cd     |
| EnOrden   | a*c+c/d | a+b+c+d | -a + x + y / + b * c * d |
| PostOrden | ab*cd/+ | ab+c+d+ | a - xy ++ b + cd ** /    |

**16.7.4. Profundidad de un árbol binario**

La profundidad de un árbol binario es una característica que se necesita conocer con frecuencia durante el desarrollo de una aplicación con árboles. La función `Profundidad` evalúa la *profundidad* de un *árbol* binario. Para ello tiene un parámetro que es un puntero a la raíz del árbol.

El caso más sencillo de cálculo de la profundidad es cuando el árbol está vacío en cuyo caso la profundidad es 0. Si el árbol no está vacío, cada subárbol debe tener su propia profundidad, por lo que se necesita evaluar cada una por separado. Las variables `profundidadI`, `profundidadD` almacenarán las profundidades de los subárboles izquierdo y derecho respectivamente.

El método de cálculo de la profundidad de los subárboles utiliza llamadas recursivas a la función `Profundidad` con punteros a los respectivos subárboles como parámetros de la misma. La fun-

ción Profundidad devuelve como resultado la profundidad del subárbol mas profundo más I (la misma del raíz).

```
int Profundidad (Nodo *p)
{
 if (!p)
 return 0 ;
 else
 {
 int profundidadI = Profundidad (p -> hijo-izqdo);
 int profundidadD = Profundidad (p -> hijo-dcho);
 if (profundidadI > profundidadD)
 return profundidadI + 1;
 else
 return profundidadD + 1;
 }
}
```

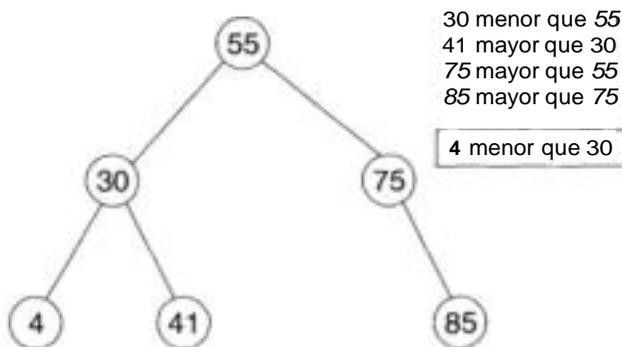
## 16.8. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles vistos hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se pueden buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en arrays.

Un **árbol binario de búsqueda** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El árbol binario del Ejemplo 16.8 es de búsqueda.

### Ejemplo 16.8

*Árbol binario de búsqueda.*

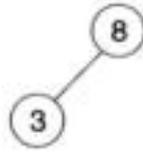


### 16.8.1. Creación de un árbol binario de búsqueda

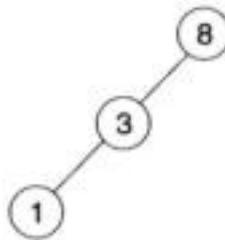
Supongamos que se desea almacenar los números 8 3 1 20 10 5 4 en un árbol binario de búsqueda. Siguiendo la regla, dado un nodo en el árbol todos los datos a su izquierda deben ser menores que todos los datos del nodo actual, mientras que todos los datos a la derecha deben ser mayores que los datos. Inicialmente el árbol está vacío y se desea insertar el 8. La única elección es almacenar el 8 en el raíz:



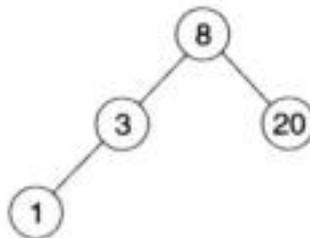
A continuación viene el 3. Ya que 3 es menor que 8, el 3 debe ir en el subárbol izquierdo.



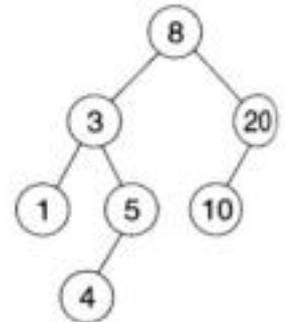
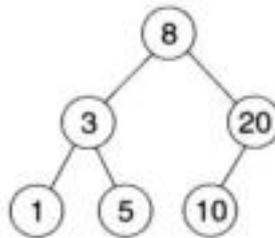
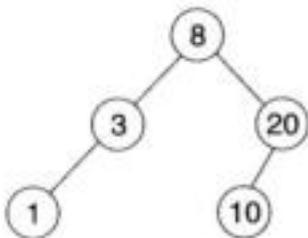
A continuación se ha de insertar 1 que es menor que 8 y que 3, por consiguiente irá a la izquierda y debajo de 3.



El siguiente número es 20, mayor que 8, lo que implica debe ir a la derecha de 8.



Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente.

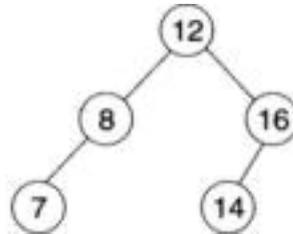


Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

**Ejemplo 16.9**

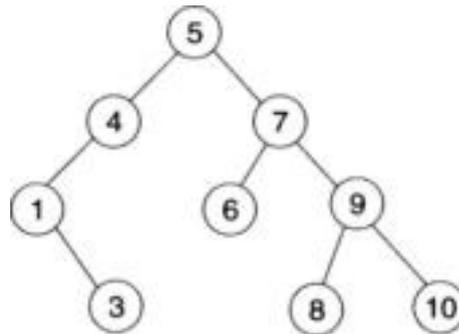
Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.

*Solución*



**Ejemplo 16.10**

Construir un árbol binario de búsqueda que corresponda a un recorrido en orden cuyos elementos son: 1, 3, 4, 5, 6, 7, 8, 9 y 10.



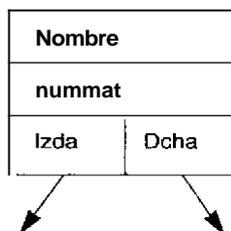
**16.8.2. Implementación de un nodo de un árbol binario de búsqueda**

Un árbol binario de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Estudiemos un ejemplo de árbol binario en el que cada nodo contiene información relativa a una persona. Cada nodo almacena un nombre de una persona y el número de matrícula en su universidad (dato entero).

Declaración de tipos

Nombre  
Matrícula

Tipo de dato cadena (string)  
Tipo entero



```

struct nodo {
 int nummat;
 char nombre[30];
 struct nodo *izda, *dcha;
};
typedef struct nodo Nodo;

```

### Creación de un nodo

La función tiene como entrada un dato entero que representa un número de matrícula y el nombre. Devuelve un puntero al nodo creado.

```
Nodo* CrearNodo(int id, char* n)

 Nodo* t;
 t = (Nodo*)malloc(sizeof(Nodo));
 t->nummat = id;
 strcpy(t->nombre, n);
 t->izda = t->dcha = NULL;
 return t;
I
```

## 16.9. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

De lo expuesto se deduce que los árboles binarios tienen naturaleza recursiva y en consecuencia las operaciones sobre los árboles son recursivas, si bien siempre tenemos la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Kecorriúo* de un árbol.
- *Borrado* de un nodo.

### 16.9.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecha. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

#### *Buscar una información específica*

Si se desea encontrar un nodo en el árbol que contenga la información sobre una persona específica. La función *buscar* tiene dos parámetros, un puntero al árbol y un número de matrícula para la persona requerida. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información sobre esa persona; en el caso de que la información sobre la persona no se encuentra se devuelve el valor 0. El algoritmo de búsqueda es el siguiente:

1. Comprobar si el árbol está vacío.  
En caso afirmativo se devuelve 0.  
Si la raíz contiene la persona, la tarea es fácil: el resultado es, simplemente, un puntero a la raíz.
2. Si el árbol no está vacío, el subárbol específico depende de que el número de matrícula requerido es más pequeño o mayor que el número de matrícula del nodo raíz.
3. La función de búsqueda se consigue llamando recursivamente a la función *buscar* con un puntero al subárbol izquierdo o derecho como paráinetro.

El código C de la función *buscar*. es:

```
Nodo* buscar (Nodo* p, int buscado)
{
```

```

if (!p)
 return 0;
else if (buscado == p -> nummat)
 return p;
else if (buscado < p -> nummat)
 return buscar (p -> izda, buscado);
else
 return buscar (p -> dcha, buscado);

```

### 16.9.2. Insertar un nodo

Una característica fundamental que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (*clave*) buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).

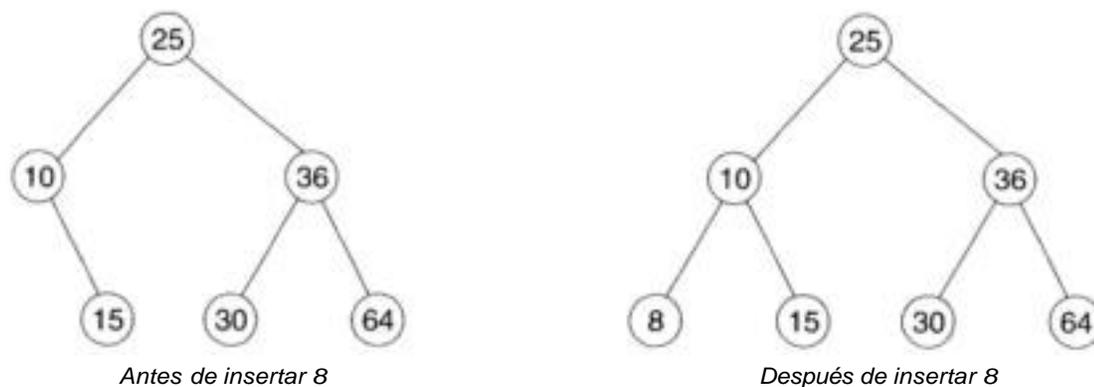
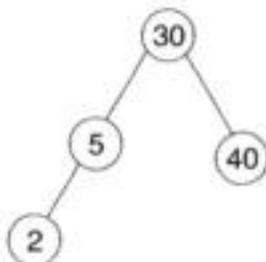


Figura 16.26. Inserción en un árbol binario de búsqueda.

Por ejemplo, considérese el caso de añadir el nodo 8 al árbol de la Figura 16.26. Se comienza el recorrido en el nodo raíz 25; la posición 8 debe estar en el subárbol izquierdo de 25 ( $8 < 25$ ). En el nodo 10, la posición de 8 debe estar en el subárbol izquierdo de 10, que está actualmente vacío. El nodo 8 se introduce como un hijo izquierdo del nodo 10.

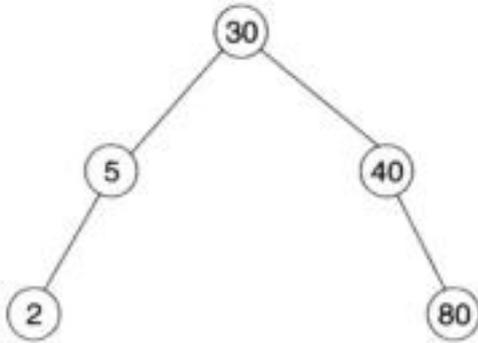
### Ejemplo 16.11

Insertar un elemento con clave 80 en el **árbol** binario de búsqueda siguiente:

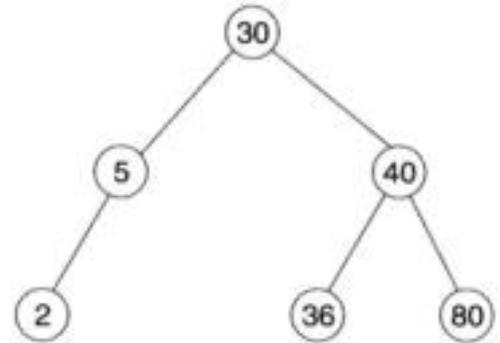


A continuación insertar un elemento con clave **36** en el árbol binario de búsqueda resultante.

### Solución



(a) Inserción de 80



(a) Inserción de 36

### 16.9.3. Función insertar()

La función `insertar` que pone nuevos nodos es sencilla. Se deben declarar tres argumentos: un puntero a la raíz del árbol, el nuevo nombre y número de matrícula de la persona. La función creará un nuevo nodo para la nueva persona y lo inserta en el lugar correcto en el árbol de modo que el árbol permanezca como binario de búsqueda.

La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. Los pasos a seguir son:

1. Asignar memoria para una nueva estructura nodo.
2. Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará como nodo hoja.
3. Enlazar el nuevo nodo al árbol.

El código *C* de la función:

```

void insertar (Nodo** raiz, int nuevomat, char *nuevo-nombre)
{
 if (!(*raiz))
 *raiz = CrearNodo(nuevo_mat, nuevo-nombre);
 else if (nuevomat < (*raiz) -> nummat)
 insertar (&((*raiz) -> izda), nuevomat, nuevo-nombre);
 else
 insertar (&((*raiz) -> dcha), nuevomat, nuevo-nombre);
}

```

Si el árbol está vacío, es fácil insertar la entrada en el lugar correcto. El nuevo nodo es la raíz del árbol y el puntero `raiz` se pone apuntando a ese nodo. El parámetro `raiz` debe ser un parámetro referencia ya que debe ser leído y actualizado, por esa razón se declara puntero a puntero (`Nodo**`). Si el árbol no está vacío, se debe elegir entre insertar el nuevo nodo en el subárbol izquierdo o derecho, dependiendo de que el número de matrícula de la nueva persona sea más pequeño o mayor que el número de matrícula en la raíz del árbol.

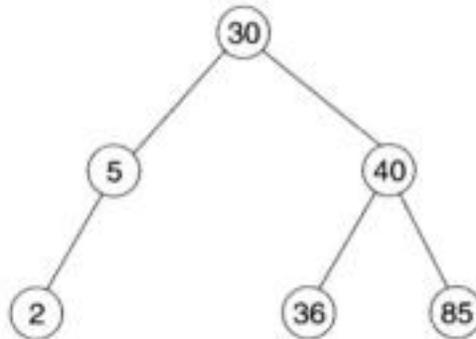
### 16.9.4. Eliminación

La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien **más** compleja que la inserción debido a que el nodo a suprimir puede ser cualquiera y la operación de supresión debe mantener la estructura de árbol binario de búsqueda después de la eliminación de datos. Los pasos a seguir son:

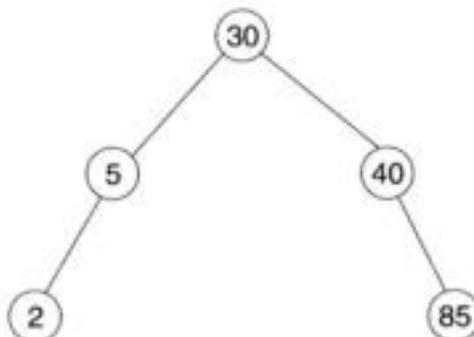
1. Buscar en el árbol para encontrar la posición de nodo a eliminar.
2. Reajustar los punteros de sus antecesores si el nodo a suprimir tiene menos de 2 hijos, o subir a la posición que éste ocupa el nodo más próximo en clave (inmediatamente superior o inmediatamente inferior) con objeto de mantener la estructura de árbol binario.

#### Ejemplo 16.12

Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:

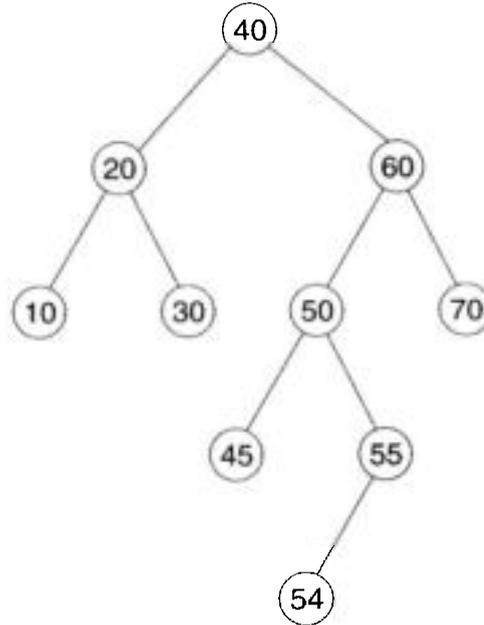


El árbol resultante es:

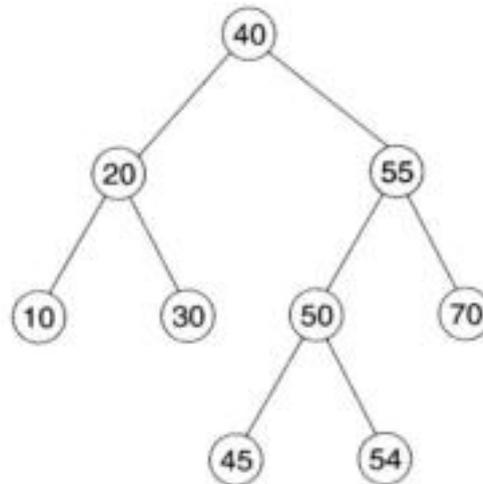


**Ejemplo 16.13**

Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza 60 bien con el elemento mayor (55) en su subárbol izquierdo o el elemento más pequeño (70) en su subárbol derecho. Si se opta por reemplazar con el elemento mayor del subárbol izquierdo. Se mueve el 55 al raíz del subárbol y se reajusta el árbol.

**Ejercicio 16.3**

Con los registros de estudiantes formar un árbol binario de búsqueda, ordenado respecto al campo clave `nummat`. El programa debe de tener las opciones de mostrar los registros ordenados y eliminar un registro dando el número de matrícula.

**Análisis**

Cada registro tiene sólo dos campos de información: nombre y `nummat`. Además los campos de enlace con el subárbol izquierdo y derecho.

Las operaciones que se van a implementar son las de insertar, eliminar, buscar y visualizar el árbol. Los algoritmos de las tres primeras operaciones ya están descritos anteriormente. La operación de visualizar va a consistir en un recorrido en inorden, cada vez que se visite el nodo raíz se escribe los datos del estudiante.

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

struct nodo {
 int nummat;
 char nombre[30];
 struct nodo *izda, *dcha;
};
typedef struct nodo Nodo;
Nodo* CrearNodo(int id, char* n);
Nodo* buscar (Nodo* p, int buscado);
void insertar (Nodo** raiz, int nuevo-mat, char *nuevo-nombre);
void eliminar (Nodo** r, int mat);
void visualizar (Nodo* r);

int main()
{
 int nm;
 char nom[30];
 Nodo* R = 0;

 /* Crea el árbol */
do{
 printf("Numero de matricula (0 -> Fin): "); scanf("%d%c",&nm);
 if (nm)
 {
 printf("Nombre: "); gets(nom);
 insertar(&R,nm,nom);
 }
}while (nm);
/* Opciones de escribir el árbol o borrar una registro */
clrscr();
do{
 puts(" 1. Mostrar el árbol\n");
 puts(" 2. Eliminar un registro\n");
 puts(" 3. Salir\n ");
 do scanf("%d%c", &nm); while(nm<1 || nm>3);
 if (nm == 1) {
 printf("\n\t Registros ordenados por número de matrícula:\n");
 visualizar(R);
 }
 else if (nm == 2){
 int cl;
 printf("Clave: "); scanf("%d",&cl);
 eliminar(&R,cl);
 }
}while (nm != 3);

return 1;

I
Nodo* CrearNodo(int id, char* n)
{
```

```

 Nodo* t;
 t = (Nodo*) malloc(sizeof(Nodo));
 t -> nummat = id;
 strcpy(t->nombre,n);
 t -> izda = t-> dcha = NULL;
 return t;
}

Nodo* buscar (Nodo* p, int buscado)

 if (!p)
 return 0;
 else if (buscado == p -> nummat)
 return p;
 else if (buscado < p -> nummat)
 return buscar (p -> izda, buscado);
 else
 return buscar (p -> dcha, buscado);
}

void insertar (Nodo** raiz, int nuevomat, char *nuevo-nombre)
{
 if (!(*raiz))
 *raiz = CrearNodo(nuevo_mat, nuevo-nombre);
 else if (nuevomat < (*raiz) -> nummat)
 insertar (&((*raiz) -> izda), nuevomat, nuevo-nombre);
 else
 insertar (&((*raiz) -> dcha), nuevo-mat, nuevo-nombre);
}

void visualizar (Nodo* r)
{
 if (r)
 {
 visualizar(r -> izda);
 printf("Matricula %d \t %s \n", r->nummat, r->nombre);
 visualizar(r -> dcha);
 }
}

void eliminar (Nodo** r, int mat)

 if (!(*r))
 printf("!! Registro con clave %d no se encuentra !!. \n", mat);
 else if (mat < (*r)->nummat)
 eliminar(&(*r)->izda, mat);
 else if (mat > (*r)->nummat)
 eliminar(&(*r)->dcha, mat);
 else /* Matricula encontrada */
 {
 Nodo* q; /* puntero al nodo a suprimir */
 q = (*r);
 if (q->izda == NULL)
 (*r) = q->dcha;
 else if (q->dcha == NULL)
 (*r) = q->izda;
 else

```

```

{ /* tiene rama izda y dcha. Se reemplaza por el mayor
 de los menores */
 Nodo* a, *p;
 p = q;
 a = q->izda;
 while (a->dcha){
 p = a;
 a = a->dcha;
 }
 q->nummat = a->nummat;
 strcpy(q->nombre, a->nombre);
 if (p == q)
 p->izda = a->izda;
 else
 p->dcha = a->izda;
 q = a;
}
free(q);
}
}

```

### 16.9.5. Recorridos de un árbol

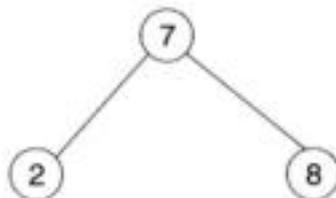
Existen dos tipos de recorrido de los nodos de un árbol: el recorrido en anchura y el recorrido en profundidad. En *el recorrido en anchura* se visitan los nodos por niveles. Para ello se utiliza una estructura auxiliar tipo cola en la que después de mostrar el contenido de un nodo, empezando por el nodo raíz, se almacenan los punteros correspondientes a sus hijos izquierdo y derecho. De esta forma si recorremos los nodos de un nivel, mientras mostramos su contenido, almacenamos en la cola los punteros a todos los nodos del nivel siguiente.

El *recorrido en profundidad* se realiza por uno de tres métodos recursivos: *preorden*, *inorden* y *postorden*. El primer método consiste en visitar el nodo raíz, su árbol izquierdo y su árbol derecho, por este orden. El recorrido *inorden* visita el árbol izquierdo, a continuación el nodo raíz y finalmente el árbol derecho. El recorrido *postorden* consiste en visitar primero el árbol izquierdo, a continuación el derecho y finalmente el raíz.

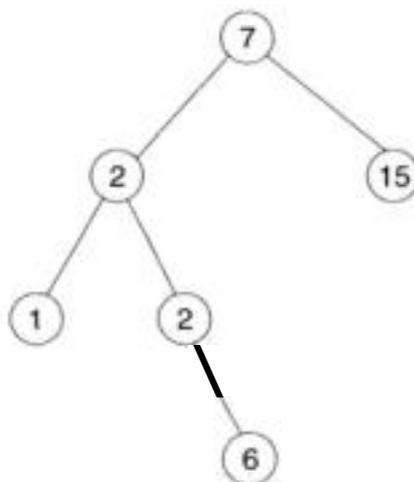
|                  |      |      |      |
|------------------|------|------|------|
| <i>preorden</i>  | Raíz | Izdo | Dcho |
| <i>en orden</i>  | Izdo | Raíz | Dcho |
| <i>postorden</i> | Izdo | Dcho | Raíz |

### 16.9.6. Determinación de la altura de un árbol

La *altura* de un árbol dependerá del criterio que se siga para definir dicho concepto. Así, si en el caso de un árbol que tiene nodo raíz, se considera que su altura es 1, la altura del árbol es 2, y la altura del árbol



es 4. Por último, si la altura de un árbol con un nodo es 1, la altura de un árbol vacío (el puntero es NULL) es 0.



### Nota

**La altura de un árbol es 1 más que la mayor de las alturas de sus subárboles izquierdo y derecho.**

A continuación, en el Ejemplo 16.14 se escribe una función entera que devuelve la altura de un árbol.

### Ejemplo 16.14

*Función que determina la altura de un árbol binario de manera recursiva. Se considera que la altura de un árbol vacío es 0; si no está vacío, la altura es 1 + máximo entre las alturas de rama izquierda y derecha.*

```

int altura(Nodo* r)
{
 if (r == NULL)
 return 0;
 else
 return (1 + max(altura(r->izda), altura(r->dcha)));
}

```

## 16.10. APLICACIONES DE ÁRBOLES EN ALGORITMOS DE EXPLORACIÓN

Los algoritmos recursivos de recorridos de árboles son el fundamento de muchas aplicaciones de árboles. Proporcionan un acceso ordenado y metódico a los nodos y a sus datos. Vamos a considerar en esta sección una serie de algoritmos de recorrido usuales en numerosos problemas de programación, tales como: contar el número de nodos hoja, calcular la profundidad de un árbol, imprimir un árbol copiar y eliminar árboles.

### 16.10.1. Visita a los nodos de un árbol

En muchas aplicaciones se desea explorar (recorrer) los nodos de un árbol pero sin tener en cuenta un orden de recorrido preestablecido. En esos casos, el cliente o usuario es libre para utilizar el algoritmo oportuno.

La función `ContarHojas` recorre el árbol y cuenta el número de nodos hoja. Para realizar esta operación se ha de visitar cada nodo comprobando si es un nodo hoja. El recorrido utilizado será el *postorden*.

```

/* Función ContarHojas
 la función utiliza recorrido postorden
 en cada visita se comprueba si el nodo es un nodo hoja
 (no tiene descendientes)
*/

void contarhojas (Nodo* r, int* nh)
{
 if (r != NULL)
 {
 contarhojas(r -> izda, nh);
 contarhojas(r -> dcha, nh);
 /* procesar raíz: determinar si es hoja */
 if (r->izda==NULL && r->dcha==NULL) (*nh)++;
 }
}

```

La función `eliminarbol` utiliza un recorrido *postorden* para liberar todos los nodos del árbol binario. Este recorrido asegura la liberación de la memoria ocupada por un nodo después de haber liberado su rama izquierda y derecha.

```

/* Función eliminarbol
 Recorre en postorden el árbol. Procesar la raíz, en esta
 función es liberar el nodo con free().
*/

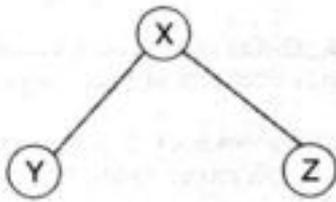
void eliminarbol (Nodo* r)
{
 if (r != NULL)
 {
 eliminarbol(r -> izda);
 eliminarbol(r -> dcha);
 printf("\tNodo borrado: %d ", r->nummat);
 free(r);
 }
}

```

## 16.11. RESUMEN

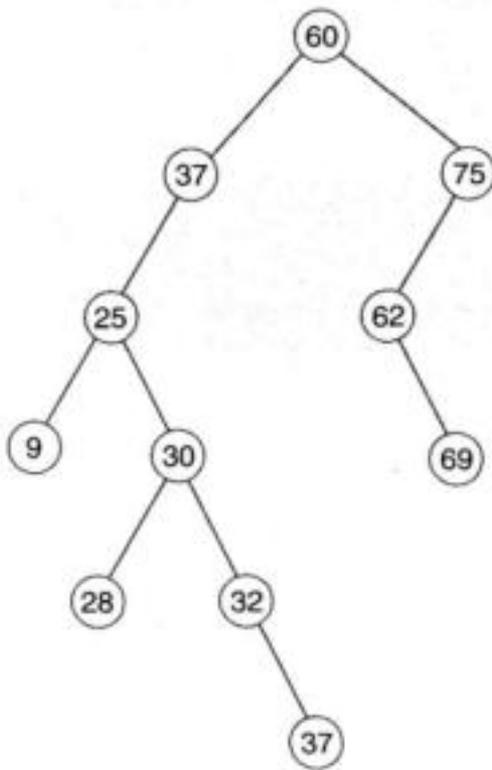
En este capítulo se introdujo y desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

La estructura árbol más utilizada normalmente es el *árbol binario*. Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.



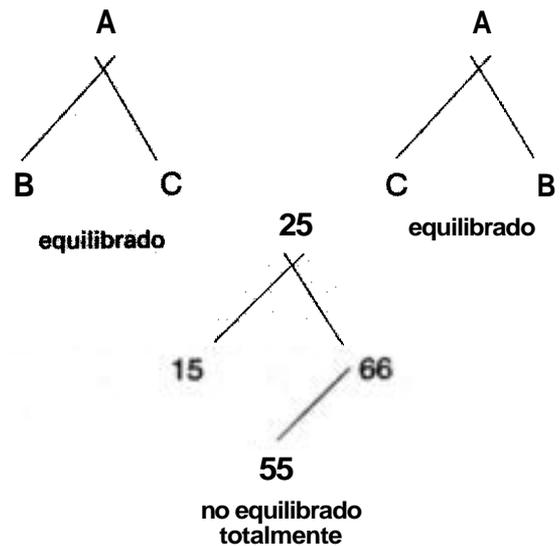
X padre o raíz  
 Y hijo izquierdo de X  
 Z hijo derecho de X

En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre. X es un *antecesor* o *ascendente* del elemento Y.



La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A es vacío, la altura es 0. La altura del árbol anterior es 6. El nivel o *profundidad* de un elemento es un concepto similar al de altura. En el árbol anterior el nivel de 30 es 3 y el nivel de 37 es 5. Un nivel de un elemento se conoce también como *profundidad*.

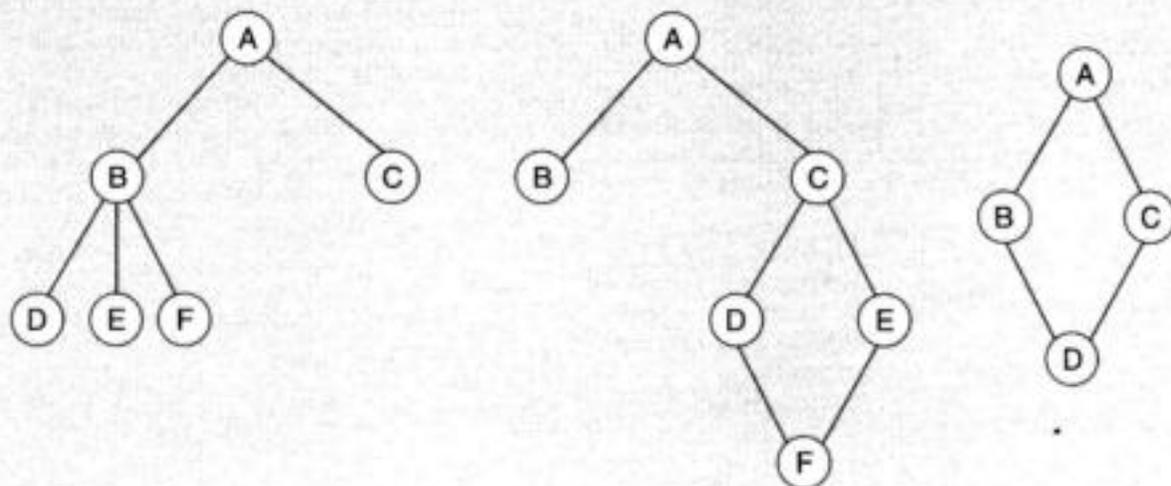
Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.



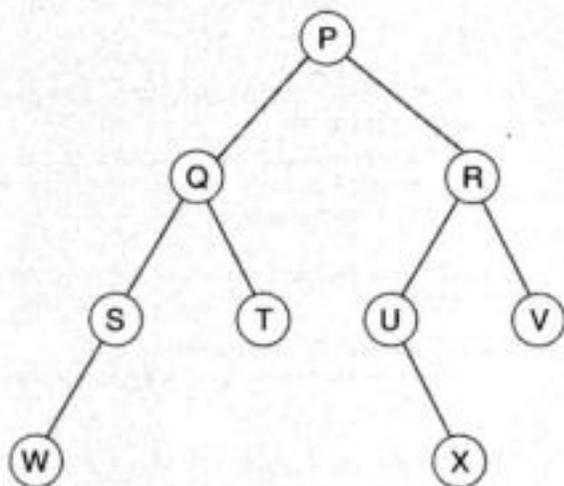
Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

### 16.12. EJERCICIOS

16.1. Explicar porqué cada una de las siguientes estructuras no es un árbol binario.



16.2. Considérese el árbol siguiente:



- ¿Cuál es su altura?
- ¿Está el árbol equilibrado? ¿Por qué?
- Listar todos los nodos hoja.
- ¿Cuál es el predecesor inmediato (padre) del nodo U?
- Listar los hijos del nodo R.
- Listar los sucesores del nodo R.

16.3. Para cada una de las siguientes listas de letras:

- Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.
- Realizar recorridos enorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.
  - M, Y, T, E, R
  - T, Y, M, E, R
  - R, E, M, Y, T
  - C, O, R, N, F, L, A, K, E, S

16.4. Para el árbol del ejercicio 2 hacer recorridos utilizando los órdenes siguientes: NDI, DNI, DIN.

16.5. Dibujar los árboles binarios que representan las siguientes expresiones:

- $(A+B) / (C-D)$
- $A+B+C/D$
- $A - (B - (C-D) / (E+F))$
- $(A+B) * ((C+D) / (E+F))$
- $(A-B) / ((C*D) - (E/F))$

16.6. El recorrido preorden de un cierto árbol binario produce.

ADFGHKLQPQRWZ

y en recorrido *enorden* produce

GFHKDLAWRQPZ

Dibujar el árbol binario.

16.7. Escribir una función no recursiva que cuente las hojas de un árbol binario.

16.8. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

- I (seguido de un carácter): Insertar un carácter
- B (seguido de un carácter): Buscar un carácter
- RE : Recorrido en orden
- RP : Recorrido en preorden
- RT : Recorrido postorden
- SA : Salir

16.9. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.

16.10. Escribir una función *booleana* a la que se le pase un puntero a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso en caso contrario.

16.11. Diseñar una función recursiva de búsqueda, que devuelva un puntero a un elemento en un árbol binario de búsqueda; si no está el elemento, devuelva NULL.

16.12. Diseñar una función iterativa que encuentre el número de nodos hoja en un árbol binario.

### 16.13. PROBLEMAS

16.1. Crear un archivo de datos en el que cada *línea* contenga la siguiente información

|          |       |                               |
|----------|-------|-------------------------------|
| Columnas | 1-20  | Nombre                        |
|          | 21-31 | Número de la Seguridad Social |
|          | 32-78 | Dirección                     |

Escribir un programa que lea *cada* registro de datos de un árbol, de modo que **cuando** el árbol se **recorra** utilizando recorrido en **orden**, los números de la seguridad social se **ordenen** en **orden** ascendente. Imprimir **una** cabecera "DATOS DE EMPLEADOS ORDENADOS POR NUMERO SEGURIDAD SOCIAL". A continuación **se** han de imprimir los **tres** datos utilizando el siguiente formato de salida.

|          |        |                               |
|----------|--------|-------------------------------|
| Columnas | 1-11   | Número de la Seguridad Social |
|          | 25-44  | Nombre                        |
|          | 58-104 | Dirección                     |

16.3. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferen-

tes contenidas **en el** texto, así como su frecuencia de aparición.

Hacer uso **de** la estructura árbol **binario** de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.

16.3. Se dispone **de** un árbol binario de elementos de tipo entero. Escribir funciones que calculen:

- a) La **suma** de sus elementos
- b) La suma de sus elementos que son múltiplos de 3.

16.4. Escribir **una** función *booleana* IDENTICOS que **permita** decir si dos árboles binarios **son** iguales.

16.5. Diseñar un programa interactivo que **permita dar altas, bajas, listar**, etc., en un árbol **binario** de búsqueda

16.6. Construir un procedimiento recursivo para escribir **todos** los nodos de un **árbol** binario de búsqueda cuyo campo clave sea mayor que un valor dado (**el** campo clave es de tipo entero).

- 16.7.** Escribir una función que determine la altura de un nodo. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:
- La altura de cada nodo del árbol.
  - La diferencia de altura entre rama izquierda y derecha de cada nodo.
- 16.8.** Diseñar procedimientos no recursivos que listen los nodos de un árbol en inorden, preorden y postorden.
- 16.9.** Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tienen o no elementos comunes.
- 16.10.** Dado un árbol binario de búsqueda construir su **árbol** espejo. (Árbol espejo es el que se construye a partir de uno dado, convirtiendo el subárbol izquierdo en subárbol derecho y viceversa.)
- 16.11.** Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición **I** del array, su hijo izquierdo se encuentra en la posición  $2 * I$  y su hijo derecho en la posición  $2I + 1$ . Diseñar a partir de esta representación los correspondientes procedimientos y funciones para gestionar interactivamente un árbol de números enteros. (Comente el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse.)
- 16.12.** Una matriz de  $N$  elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordene ascendente la cadena de caracteres.
- 16.13.** Dado un árbol binario de búsqueda diseñe un procedimiento que liste los nodos del árbol ordenados descendente.