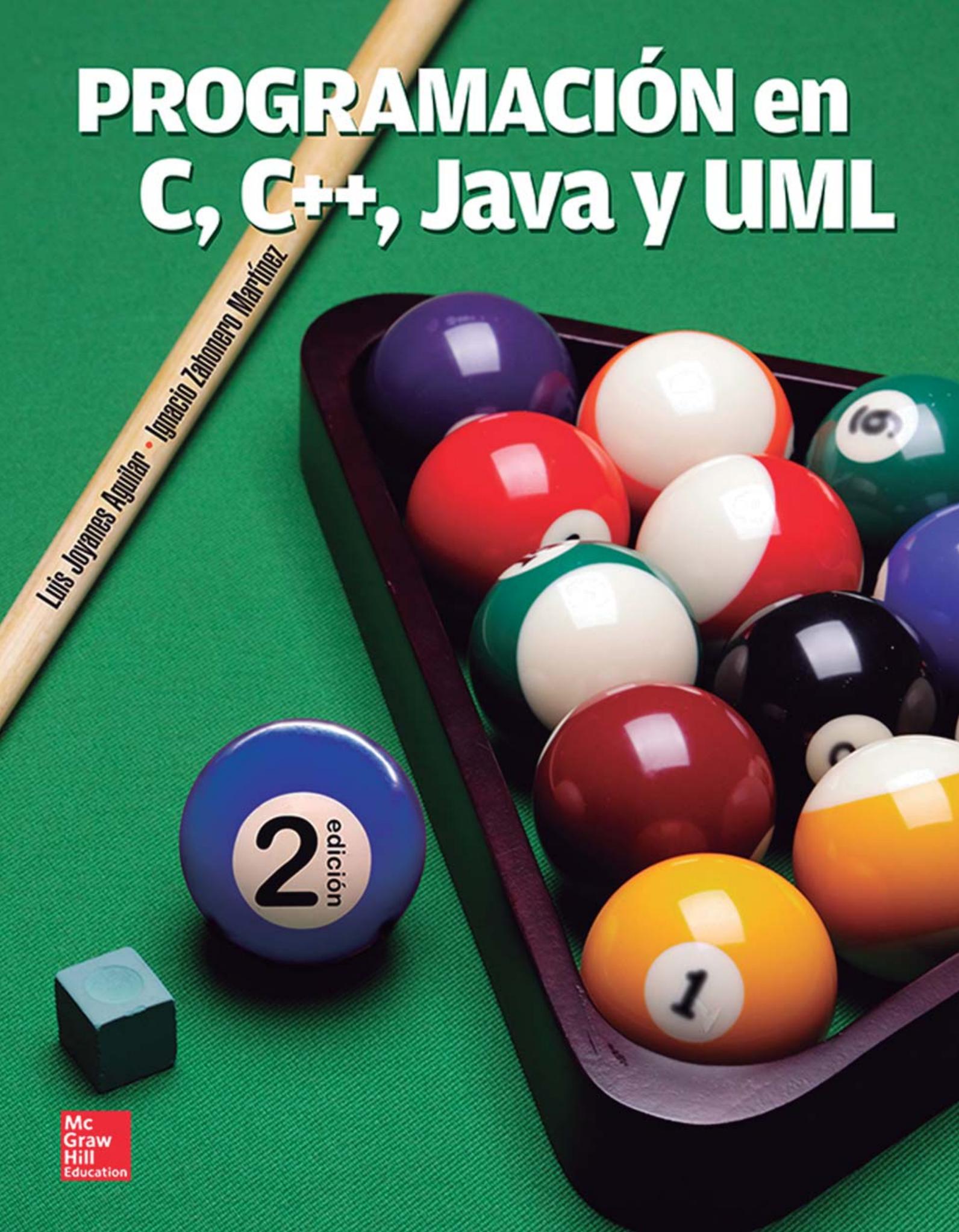


PROGRAMACIÓN en C, C++, Java y UML

Luis Joyanes Aguilan · Ignacio Zahonero Martínez



Programación en C, C++, Java y UML

Programación en C, C++, Java y UML

Segunda edición

Luis Joyanes Aguilar

Catedrático de Lenguajes y Sistemas Informáticos
Universidad Pontificia de Salamanca

Ignacio Zahonero Martínez

Profesor Asociado de Programación y Estructura de Datos
Universidad Pontificia de Salamanca



MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID • NUEVA YORK
SAN JUAN • SANTIAGO • SAO PAULO • AUCKLAND • LONDRES • MILÁN • MONTREAL
NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director general: Miguel Ángel Toledo Castellanos
Coordinador sponsor: Jesús Mares Chacón
Coordinadora editorial: Marcela I. Rocha Martínez
Editora de desarrollo: Ana Laura Delgado Rodríguez
Supervisor de producción: Zeferino García García

PROGRAMACIÓN EN C, C++, JAVA Y UML

Segunda edición

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin la autorización escrita del editor.



DERECHOS RESERVADOS © 2014, 2009 respecto a la segunda edición por
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.

Edificio Punta Santa Fe

Prolongación Paseo de la Reforma 1015, Torre A

Piso 17, Colonia Desarrollo Santa Fe,

Delegación Álvaro Obregón

C.P. 01376, México, D. F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-607-15-1212-3

ISBN(de la edición anterior): 978-970-10-6949-3

ARR 05/14

1234567890

Impreso en México

2356789014

Printed in Mexico

Contenido

PRÓLOGO	xv	
PARTE I		
Resolución de problemas con software	1	
CAPÍTULO 1		
Fundamentos de computadoras y de lenguajes de programación	2	
Introducción	2	
1.1 Las computadoras en perspectiva	3	
Generaciones de computadoras	3	
1.2 Las computadoras modernas: una breve taxonomía	5	
1.3 Estructura de una computadora	6	
1.4 <i>Hardware</i>	7	
Unidad central de proceso (CPU)	8	
Memoria de la computadora	9	
Dispositivos de entrada y salida	12	
Dispositivos de almacenamiento secundario	12	
Dispositivos de comunicación	13	
1.5 Software: conceptos básicos y clasificación	13	
Software de sistema	14	
Software de aplicaciones	14	
1.6 Sistema operativo	15	
1.7 El lenguaje de la computadora	17	
Representación de la información en las computadoras (códigos de caracteres)	18	
Los lenguajes de programación	18	
ensambladores y de alto nivel	18	
El proceso de programación	20	
1.8 Internet y la Web	21	
La revolución Web 2.0	22	
Social Media	22	
Desarrollo de programas web	23	
La Web Semántica y la Web 3.0	23	
1.9 <i>Cloud computing</i> (computación en la nube)	23	
Software como servicio (SaaS)	24	
1.10 Movilidad: tecnologías, redes e internet móvil	24	
Redes inalámbricas	24	
Redes móviles	25	
Sistemas operativos móviles	25	
Aplicaciones móviles	25	
1.11 Geolocalización y realidad aumentada	26	
¿Qué es la realidad aumentada?	26	
1.12 Internet de las cosas	26	
1.13 <i>Big data</i> . Los grandes volúmenes de datos	27	
1.14 Lenguajes de programación	27	
Traductores de lenguaje: el proceso de traducción de un programa	29	
La compilación y sus fases	29	
1.15 Evolución de los lenguajes de programación	31	
1.16 Paradigmas de programación	32	
Lenguajes imperativos (procedimentales)	32	
Lenguajes declarativos	32	
Lenguajes orientados a objetos	33	
Resumen	33	
CAPÍTULO 2		
Algoritmos, programas y metodología de la programación	34	
Introducción	34	
2.1 Resolución de problemas con computadoras: fases	34	
Análisis del problema	35	
Diseño del problema	36	
Herramientas gráficas y alfanuméricas	37	
Codificación de un programa	39	
Compilación y ejecución de un programa	40	
Verificación y depuración de un programa	40	
Mantenimiento y documentación	42	
2.2 Algoritmo: concepto y propiedades	42	
Características de los algoritmos	43	
Diseño de algoritmos	44	
Escritura de algoritmos	45	
Representación gráfica de los algoritmos	47	
Pseudocódigo	47	
Diagramas de flujo	48	
Diagramas de Nassi-Schneiderman (N-S)	57	
2.6 Metodología de la programación	58	
Programación modular	58	
Programación estructurada	58	
Programación orientada a objetos	60	
2.7 Herramientas de programación	61	
Editores de texto	61	
Programa ejecutable	62	
Proceso de compilación/ejecución de un programa	62	
Resumen	64	
Ejercicios	64	
Actividades de aprendizaje	65	
Actividades complementarias	66	
PARTE II		
Programación en C	67	
CAPÍTULO 3		
El lenguaje C: elementos básicos	68	
Introducción	68	
3.1 Estructura general de un programa en C	68	
Directivas del preprocesador	70	
Declaraciones globales	71	

Función main()	72	4.8	Operador coma	116
Funciones definidas por el usuario	72	4.9	Operadores especiales: (), []	116
Comentarios	74		El operador ()	116
3.2 Creación de un programa	75	4.10	El operador []	116
3.3 El proceso de ejecución de un programa en C	76	4.11	Operador sizeof	117
3.4 Depuración de un programa en C	79		Conversiones de tipos	118
Errores de sintaxis	79		Conversión implícita	118
Errores lógicos	79		Reglas	118
Errores de regresión	80		Conversión explícita	118
Mensajes de error	80	4.12	Prioridad y asociatividad	119
Errores en tiempo de ejecución	80		Resumen	120
3.5 Pruebas	81		Ejercicios	120
3.6 Los elementos de un programa en C	81		Problemas	121
Tokens (elementos léxicos de los programas)	82			
Identificadores	82			
Palabras reservadas	82			
Comentarios	83			
Signos de puntuación y separadores	83			
Archivos de cabecera	83			
3.7 Tipos de datos en C	83	5.1	Estructuras de control	123
Enteros (int)	84	5.2	La sentencia if	124
Tipos de coma flotante (float/double)	85	5.3	Sentencia if de dos opciones: if-else	126
Caracteres (char)	86	5.4	Sentencias if-else anidadas	129
3.8 El tipo de dato lógico	87		Sangría en las sentencias if anidadas	130
Escritura de valores lógicos	88		Comparación de sentencias if anidadas	
3.9 Constantes	88		y secuencias de sentencias if	131
Constantes literales	88	5.5	Sentencia de control switch	132
Constantes definidas (simbólicas)	91		Caso particular de case	136
Constantes enumeradas	91		Uso de sentencias switch en menús	137
Constantes declaradas const y volatile	91	5.6	Expresiones condicionales: el operador ?:	137
3.10 Variables	92	5.7	Evaluación en cortocircuito de	
Declaración	93		expresiones lógicas	138
Inicialización de variables	94	5.8	Puesta a punto de programas	139
Declaración o definición	95		Estilo y diseño	139
3.11 Duración de una variable	95	5.9	Errores frecuentes de programación	140
Variables locales	95		Resumen	142
Variables globales	96		Ejercicios	142
3.12 Entradas y salidas	97		Problemas	144
Salida	97			
Entrada	99			
Salida de cadenas de caracteres	100			
Entrada de cadenas de caracteres	100			
Resumen	101			
Ejercicios	102			
CAPÍTULO 4	103			
Operadores y expresiones				
Introducción	103			
4.1 Operadores y expresiones	103	6.1	Sentencia while	145
4.2 Operador de asignación	104		Operadores de incremento y decrecimiento	145
4.3 Operadores aritméticos	105		(++, --)	148
Asociatividad	107		Terminaciones anormales de un bucle	
Uso de paréntesis	107		(ciclo o llave)	149
4.4 Operadores de incrementación	108		Diseño eficiente de bucles	149
y decrementación	108		Bucles while con cero iteraciones	150
4.5 Operadores relacionales	111		Bucles controlados por centinelas	150
4.6 Operadores lógicos	112		Bucles controlados por indicadores (banderas)	151
Asignaciones booleanas (lógicas)	114		La sentencia break en los bucles	152
4.7 Operador condicional ?:	115		Bucles while (true)	153
		6.2	Repetición: el bucle for	154
			Diferentes usos de bucles for	158
		6.3	Precauciones en el uso de for	159
			Bucles infinitos	160
			Los bucles for vacíos	161
			Sentencias nulas en bucles for	161
CAPÍTULO 5	123			
Estructuras de selección: sentencias if y switch				
Introducción	123			
5.1 Estructuras de control	123			
5.2 La sentencia if	124			
5.3 Sentencia if de dos opciones: if-else	126			
5.4 Sentencias if-else anidadas	129			
Sangría en las sentencias if anidadas	130			
Comparación de sentencias if anidadas				
y secuencias de sentencias if	131			
5.5 Sentencia de control switch	132			
Caso particular de case	136			
Uso de sentencias switch en menús	137			
5.6 Expresiones condicionales: el operador ?:	137			
5.7 Evaluación en cortocircuito de				
expresiones lógicas	138			
5.8 Puesta a punto de programas	139			
Estilo y diseño	139			
5.9 Errores frecuentes de programación	140			
Resumen	142			
Ejercicios	142			
Problemas	144			
CAPÍTULO 6	145			
Estructuras de control: bucles				
Introducción	145			
6.1 Sentencia while	145			
Operadores de incremento y decrecimiento				
(++, --)	148			
Terminaciones anormales de un bucle				
(ciclo o llave)	149			
Diseño eficiente de bucles	149			
Bucles while con cero iteraciones	150			
Bucles controlados por centinelas	150			
Bucles controlados por indicadores (banderas)	151			
La sentencia break en los bucles	152			
Bucles while (true)	153			
6.2 Repetición: el bucle for	154			
Diferentes usos de bucles for	158			
6.3 Precauciones en el uso de for	159			
Bucles infinitos	160			
Los bucles for vacíos	161			
Sentencias nulas en bucles for	161			

6.4	Sentencias break y continue	162	7.14	Funciones recursivas	210
	Repetición: el bucle do-while	163		Recursividad indirecta: funciones	
	Diferencias entre while y do-while	164		mutuamente recursivas	211
6.5	Comparación de bucles	165	7.15	Condición de terminación de la recursión	213
	while, for y do-while			Recursión versus iteración	213
6.6	Diseño de bucles	165		Directrices en la toma de decisión	
	Final de un bucle	166		iteración/recursión	214
	Bucles for vacíos	167	7.16	Recursión infinita	215
6.7	Bucles anidados	168		Resumen	216
6.8	Enumeraciones	170		Ejercicios	217
	Resumen	172		Problemas	219
	Ejercicios	172			
	Problemas	174			

CAPÍTULO 7

Funciones y recursividad

Introducción

7.1	Concepto de función	176	8.1	Arreglos (arrays)	221
7.2	Estructura de una función	177		Declaración de un arreglo o array	222
	Nombre de una función	178		Subíndices de un arreglo	223
	Tipo de dato de retorno	179		Almacenamiento en memoria de los arreglos	
	Resultados de una función	180		(arrays)	224
	Llamada a una función	180		El tamaño de los arreglos	225
7.3	Prototipos de las funciones	181		Verificación del rango del índice de un arreglo	225
	Prototipos con un número no especificado	181	8.2	Inicialización de un arreglo (array)	225
	de parámetros	184	8.3	Arreglos multidimensionales	228
7.4	Parámetros de la función	184		Inicialización de arreglos multidimensionales	229
	Paso de parámetros por valor	186		Acceso a los elementos de los arreglos	
	Paso de parámetros por referencia	187		bidimensionales	230
	Diferencias entre paso de variables	188		Lectura y escritura de elementos de arreglos	
	por valor y por referencia	189		bidimensionales	230
	Parámetros const de una función	189		Acceso a elementos mediante bucles	231
7.5	Funciones en línea, macros con argumentos	191		Arreglo de más de dos dimensiones	232
	Creación de macros con argumentos	191	8.4	Proceso de un arreglo de tres dimensiones	233
7.6	Ámbito (alcance) de una variable	193		Utilización de arreglos como parámetros	234
	Ámbito del programa	194		Precauciones	237
	Ámbito del archivo fuente	194	8.5	Paso de cadenas como parámetros	238
	Ámbito de una función	195		Concepto de cadena	238
	Ámbito de bloque	195		Declaración de variables de cadena	239
	Variables locales	195	8.6	Inicialización de variables de cadena	240
7.7	Clases de almacenamiento	196		Lectura de cadenas	241
	Variables automáticas	196		Función gets()	241
	Variables externas	196		Función getchar()	243
	Variables registro	197		Función putchar()	243
	Variables estáticas	197	8.7	Función puts()	244
7.8	Concepto y uso de funciones de biblioteca	198		La biblioteca string.h	245
7.9	Funciones de carácter	199		Utilización del modificador const	
	Comprobación alfabética y de dígitos	199	8.8	con cadenas	245
	Funciones de prueba de	201		Arreglos y cadenas como parámetros	
	caracteres especiales	201		de funciones	245
	Funciones de conversión de caracteres	202	8.9	Asignación de cadenas	247
7.10	Funciones numéricas	202		Función strncpy()	248
	Funciones matemáticas de carácter general	202	8.10	Longitud y concatenación de cadenas	249
	Funciones trigonométricas	204		Función strlen()	249
	Funciones logarítmicas y exponenciales	204	8.11	Funciones strcat() y strncat()	249
	Funciones aleatorias	205		Comparación de cadenas	250
7.11	Funciones de utilidad	206		Función strcmp()	250
7.12	Visibilidad de una función	208		Función strncmp()	251
7.13	Compilación separada	208	8.12	Conversión de cadenas a números	251
				Función atoi()	251
				Función atof()	252

CAPÍTULO 8

Arreglos (arrays), listas y tablas. Cadenas

176	Introducción	221
	8.1 Arreglos (arrays)	222
	Declaración de un arreglo o array	222
	Subíndices de un arreglo	223
	Almacenamiento en memoria de los arreglos	
	(arrays)	224
	El tamaño de los arreglos	225
	Verificación del rango del índice de un arreglo	225
8.2	Inicialización de un arreglo (array)	225
8.3	Arreglos multidimensionales	228
	Inicialización de arreglos multidimensionales	229
	Acceso a los elementos de los arreglos	
	bidimensionales	230
	Lectura y escritura de elementos de arreglos	
	bidimensionales	230
	Acceso a elementos mediante bucles	231
	Arreglo de más de dos dimensiones	232
	Proceso de un arreglo de tres dimensiones	233
8.4	Utilización de arreglos como parámetros	234
	Precauciones	237
8.5	Paso de cadenas como parámetros	238
	Concepto de cadena	238
	Declaración de variables de cadena	239
	Inicialización de variables de cadena	240
8.6	Lectura de cadenas	241
	Función gets()	241
	Función getchar()	243
	Función putchar()	243
	Función puts()	244
8.7	La biblioteca string.h	245
	Utilización del modificador const	
	con cadenas	245
	Arreglos y cadenas como parámetros	
	de funciones	245
8.8	Asignación de cadenas	247
	Función strncpy()	248
	Longitud y concatenación de cadenas	249
	Función strlen()	249
	Funciones strcat() y strncat()	249
8.11	Comparación de cadenas	250
	Función strcmp()	250
	Función strncmp()	251
8.12	Conversión de cadenas a números	251
	Función atoi()	251
	Función atof()	252

Función <code>atol()</code>	252	10.9 Tamaño de estructuras y uniones	295
Función <code>strtol()</code> y <code>strtoul()</code>	253	Resumen	296
Función <code>strtod()</code>	254	Ejercicios	296
Entrada de números y cadenas	255	Problemas	297
Resumen	256		
Ejercicios	256		
Problemas	258		
CAPÍTULO 9			
Algoritmos de ordenación y búsqueda			
Introducción	260	CAPÍTULO 11	
9.1 Ordenación	260	Apuntableadores (punteros)	299
9.2 Ordenación por burbuja	260	Introducción	299
Algoritmo de la burbuja	261	11.1 Direcciones en memoria	300
Codificación del algoritmo de la burbuja	262	11.2 Concepto de apuntador (puntero)	301
Análisis del algoritmo de la burbuja	263	Declaración de apuntadores	302
9.3 Ordenación por selección	264	Inicialización (iniciación) de apuntadores	303
Algoritmo de selección	265	Indirección de apuntadores	304
Codificación del algoritmo de selección	266	Apuntadores y verificación de tipos	305
9.4 Ordenación por inserción	266	11.3 Apuntadores <code>NULL</code> y <code>void</code>	306
Algoritmo de ordenación por inserción	266	11.4 Apuntadores a apuntadores	307
Codificación del algoritmo de inserción	267	11.5 Apuntadores y arreglos	308
9.5 Ordenación rápida (<i>quicksort</i>)	267	Nombres de arreglos como apuntadores	308
Algoritmo <i>quicksort</i>	269	Ventajas de los apuntadores	308
Codificación del algoritmo <i>quicksort</i>	270	11.6 Arreglos (<i>arrays</i>) de apuntadores	309
Análisis del algoritmo <i>quicksort</i>	272	Inicialización de un arreglo de apuntadores	
9.6 Búsqueda en listas: búsqueda secuencial	273	a cadenas	310
y binaria	273	11.7 Apuntadores a cadenas	310
Búsqueda binaria	273	Apuntadores versus arreglos	310
Algoritmo y codificación de la búsqueda	273	11.8 Aritmética de apuntadores	311
binaria	275	Una aplicación de apuntadores:	
Análisis de los algoritmos de búsqueda	275	conversión de caracteres	312
Resumen	277	11.9 Apuntadores constantes frente	
Ejercicios	277	a apuntadores a constantes	313
Problemas	278	Apuntadores constantes	313
CAPÍTULO 10		Apuntadores a constantes	314
Estructuras y uniones		Apuntadores constantes a constantes	315
Introducción	280	11.10 Apuntadores como argumentos	
10.1 Estructuras	280	de funciones	316
Declaración de una estructura	280	11.11 Apuntadores a funciones	317
Definición de variables de estructuras	281	Inicialización de un apuntador a una función	318
Uso de estructuras en asignaciones	282	11.12 Apuntadores a estructuras	321
Inicialización de una declaración de estructuras	283	11.13 Asignación dinámica de la memoria	322
El tamaño de una estructura	284	Almacén libre (<i>free store</i>)	322
10.2 Acceso a estructuras	285	11.14 Función de asignación de memoria	
Almacenamiento de información en	285	<i>malloc()</i>	322
estructuras	285	Asignación de memoria de un tamaño	
Lectura de información de una estructura	286	desconocido	326
Recuperación de información de una	286	Uso de <i>malloc()</i> para arreglos	
estructura	287	multidimensionales	326
10.3 Sinónimo de un tipo de datos: <code>typedef</code>	287	11.15 La función <code>free()</code>	327
10.4 Estructuras anidadas	288	11.16 Funciones de asignación de memoria	
Ejemplo de estructuras anidadas	290	<i>calloc()</i> y <i>realloc()</i>	328
10.5 Arreglos de estructuras	291	Función <i>calloc()</i>	328
10.6 Arreglos como miembros	292	Función <i>realloc()</i>	330
10.7 Utilización de estructuras como parámetros	293	11.17 Reglas de funcionamiento de la asignación	
10.8 Uniones	294	dinámica	332
Resumen		Resumen	333
Ejercicios		Ejercicios	334
Problemas		Problemas	335
CAPÍTULO 12			
Entradas y salidas por archivos			
Introducción			337

12.1	Flujos	337	Elementos de comportamiento	378
12.2	Apuntador (puntero) FILE	338	Elementos de agrupación	379
12.3	Apertura de un archivo	339	13.8 Especificaciones de UML	379
	Modos de apertura de un archivo	340	13.9 Historia de UML	379
	NULL y EOF	341	Resumen	381
	Cierre de archivos	341	Ejercicios	381
12.4	Volcado del buffer: <code>fflush()</code>	341		
	Funciones de entrada/salida para archivos	342		
	Funciones <code>putc()</code> y <code>fputc()</code>	342		
	Funciones <code>getc()</code> y <code>fgetc()</code>	343		
	Funciones <code>fputs()</code> y <code>fgets()</code>	344		
	Funciones <code>fprintf()</code> y <code>fscanf()</code>	345		
	Función <code>feof()</code>	347		
	Función <code>rewind()</code>	347		
12.5	Archivos binarios en C	348		
	Función de salida <code>fwrite()</code>	348		
	Función de lectura <code>fread()</code>	350		
12.6	Funciones para acceso aleatorio	351		
	Función <code>fseek()</code>	352		
	Función <code>ftell()</code>	353		
	Cambio de posición: <code>fgetpos()</code> y <code>fsetpos()</code>	353		
12.7	Datos externos al programa con argumentos de <code>main()</code>	355		
	Resumen	358		
	Ejercicios	358		
	Problemas	359		
PARTE III				
Lenguaje unificado de modelado UML 2.5		361		
CAPÍTULO 13				
Programación orientada a objetos y UML 2.5		362		
	Introducción	362		
13.1	Programación orientada a objetos	363		
	Objetos	364		
	Tipos abstractos de datos: Clases	365		
13.2	Modelado e identificación de objetos	367		
	Estado	367		
	Comportamiento	367		
	Identidad	367		
13.3	Propiedades fundamentales de la orientación a objetos	368		
	Abstracción	368		
	La abstracción en software	368		
	Encapsulamiento y ocultación de datos	369		
	Herencia	369		
	Reutilización o reusabilidad	370		
	Polimorfismo	371		
13.4	Modelado de aplicaciones: UML	372		
	Lenguaje de modelado	372		
13.5	Modelado y modelos	373		
13.6	Diagramas de UML 2.5	374		
13.7	Bloques de construcción (componentes) de UML 2.5	377		
	Elementos estructurales	377		
CAPÍTULO 14				
Diseño de clases y objetos: Representaciones gráficas en UML		383		
	Introducción	383		
14.1	Diseño y representación gráfica de objetos en UML	384		
	Representación gráfica en UML	385		
	Características de los objetos	386		
	Estado	387		
	Múltiples instancias de un objeto	388		
	Evolución de un objeto	389		
	Comportamiento	389		
	Identidad	391		
	Los mensajes	392		
	Responsabilidad y restricciones	393		
14.2	Diseño y representación gráfica de clases en UML	394		
	Representación gráfica de una clase	395		
	Declaración de una clase	397		
	Reglas de visibilidad	398		
14.3	Declaración de objetos de clases	400		
	Acceso a miembros de la clase: encapsulamiento	401		
	Declaración de métodos	403		
	Tipos de métodos	405		
	Resumen	405		
	Ejercicios	406		
CAPÍTULO 15				
Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia		409		
	Introducción	409		
15.1	Relaciones entre clases	409		
15.2	Dependencia	410		
15.3	Asociación	411		
	Multiplicidad	413		
	Restricciones en asociaciones	414		
	Asociación cualificada	415		
	Asociaciones reflexivas	415		
	Diagrama de objetos	415		
	Clases de asociación	416		
	Asociaciones ternarias	417		
	Asociaciones cualificadas	418		
	Asociaciones reflexivas	419		
	Restricciones en asociaciones	419		
15.4	Agregación	421		
	Composición	422		
15.5	Jerarquía de clases: generalización y especialización	422		
	Jerarquías de generalización/ especialización	423		
		425		

15.6	Herencia: clases derivadas	427	Secuencia y sentencias compuestas	461
	Herencia simple	428	Selección y repetición	462
	Herencia múltiple	428	Sentencia <i>if</i>	462
	Niveles de herencia	429	Sentencia <i>Switch</i>	463
	Declaración de una clase derivada	432	Sentencia <i>while</i>	463
	Consideraciones de diseño	432	Sentencia <i>do-while</i>	464
15.7	Accesibilidad y visibilidad en la herencia	433	Sentencia <i>for</i>	464
	Herencia pública	434	Sentencias <i>break</i> y <i>continue</i>	465
	Herencia privada	434	Estructura de <i>break</i> y <i>continue</i>	465
	Herencia protegida	434	16.7 Funciones	466
15.8	Un caso de estudio especial: herencia múltiple	435	Funciones de biblioteca	466
	Características de la herencia múltiple	437	Definición de una función (función definida por el usuario)	467
15.9	Clases abstractas	438	Argumentos de la función: paso por valor y por referencia	468
	Operaciones abstractas	439	La sentencia <i>return</i>	469
	Resumen	441	La sentencia <i>using</i> y el espacio de nombres <i>std</i>	469
	Ejercicios	441	Resumen	470
			Ejercicios	471
			Problemas	472

PARTE IV **Programar en C++**

443

CAPÍTULO 16 **De C a C++**

444

	Introducción	444
16.1	El primer programa C++	444
	Comentarios en C++	446
	El preprocesador de C++ y el archivo <i>iostream</i>	446
	Entrada y salida	446
	Archivos de cabecera	447
16.2	Espacios de nombres	447
16.3	Tipos de datos nativos	448
	Tipos de datos básicos/primitivos	448
	Tipos de coma flotantes (reales)	449
	Constantes literales	450
	Tipos apuntadores	450
	Declaración de apuntadores	451
	Tipos constantes	451
	Referencias	452
	Nombres de tipos definidos: <i>typedef</i>	452
	Tipos enumeración	452
	Arrays (arreglos)	452
	Tipos carácter	453
	Cadenas	453
16.4	Operadores	454
	Operadores aritméticos	454
	Operadores relacionales y lógicos	455
	Operadores de asignación	456
	Operadores incremento y decremento	456
	Operador condicional	457
	Operador <i>sizeof</i>	457
	Operador coma	458
16.5	Conversiones de tipos	458
	Conversión en expresiones	459
	Conversiones en paso de argumentos	459
	Conversiones explícitas	459
	Operador <i>new</i>	459
16.6	Estructuras de control	461

CAPÍTULO 17 **Clases y objetos.** **Sobrecarga de operadores**

473

	Introducción	473
17.1	Clases y objetos	473
	¿Qué son objetos?	474
	¿Qué son clases?	474
17.2	Definición de una clase	475
	Objetos de clases	479
	Acceso a miembros de la clase: encapsulamiento	481
	Datos miembros (miembros dato)	483
	Funciones miembro	485
	Llamadas a funciones miembro	487
	Tipos de funciones miembro	488
	Funciones en línea y fuera de línea	489
	La palabra reservada <i>inline</i>	490
	Nombres de parámetros de funciones miembro	491
	Implementación de clases	491
	Archivos de cabecera y de clases	492
17.3	Constructores	493
	Constructor por defecto	494
	Constructores alternativos	495
	Constructores sobrecargados	495
	Constructor de copia	496
	Inicialización de miembros en constructores	496
17.4	Destructores	498
	Clases compuestas	499
17.5	Sobrecarga de funciones miembro	499
17.6	Funciones amigas	500
17.7	Sobrecarga de operadores	502
17.8	Sobrecarga de operadores unitarios	504
	Sobrecargar un operador unitario como función miembro	505
	Sobrecarga de un operador unitario como una función amiga	506

17.9	Sobrecarga de operadores binarios	507	19.4	Problemas en las funciones plantilla	559
	Sobrecarga de un operador binario como función miembro	507		Plantillas de clases	559
	Sobrecarga de un operador binario como una función amiga	508		Definición de una plantilla de clase	560
17.10	Conversión de datos y operadores de conversión de tipos	509	19.5	Instanciación de una plantilla de clases	562
17.11	Errores de programación frecuentes	510		Utilización de una plantilla de clase	562
	Resumen	514		Argumentos de plantillas	564
	Ejercicios	515		Una plantilla para manejo de pilas de datos	564
	Problemas	518		Definición de las funciones miembro	565
				Utilización de una clase plantilla	565
				Instanciación de una clase plantilla con clases	568
				Uso de las plantillas de funciones con clases	569
			19.6	Modelos de compilación de plantillas	569
				Modelo de compilación separada	570
		520	19.7	Plantillas frente a polimorfismo	571
			Resumen		572
			Ejercicios		573
CAPÍTULO 18					
Clases derivadas: herencia y polimorfismo					
Introducción					
18.1	Clases derivadas	524			
	Declaración de una clase derivada	525			
	Consideraciones de diseño	525			
18.2	Tipos de herencia	525			
	Herencia pública	528			
	Herencia privada	528			
	Herencia protegida	529			
	Operador de resolución de ámbito	529			
	Constructores-inicializadores en herencia	530			
	Sintaxis del constructor	531			
	Sintaxis de la implementación de una función miembro	532			
18.3	Destructores	532			
18.4	Herencia múltiple	533			
	Características de la herencia múltiple	535	20.5		
	Dominación (prioridad)	536			
	Inicialización de la clase base	537			
18.5	Ligadura	539			
18.6	Funciones virtuales	539			
	Ligadura dinámica mediante funciones virtuales	540	20.6		
18.7	Polimorfismo	542			
	El polimorfismo sin ligadura dinámica	543	20.7		
	El polimorfismo con ligadura dinámica	543	20.8		
18.8	Uso del polimorfismo	544			
18.9	Ligadura dinámica frente a ligadura estática	544			
18.10	Ventajas del polimorfismo	545			
	Resumen	545			
	Ejercicios	546			
CAPÍTULO 19					
Genericidad: plantillas (templates)					
Introducción					
19.1	Genericidad	548			
19.2	Conceptos fundamentales de plantillas en C++	548			
19.3	Plantillas de funciones	549			
	Fundamentos teóricos	551			
	Definición de plantilla de funciones	551			
	Un ejemplo de función plantilla	553			
	Plantillas de función ordenar y buscar	557	21.1		
	Una aplicación práctica	557			
		558			
				Historia del lenguaje Java: de Java 1.0 a Java 8	596
				El lenguaje de programación Java	598
				Tipos de programas Java	599
CAPÍTULO 20					
Excepciones					
Introducción					
20.1	Condiciones de error en programas	574			
	¿Por qué considerar las condiciones de error?	574			
20.2	El tratamiento de los códigos de error	575			
20.3	Manejo de excepciones en C++	575			
20.4	El mecanismo de manejo de excepciones	576			
	Clases de excepciones	577			
	Partes de la manipulación de excepciones	578			
20.5	El modelo de manejo de excepciones	578			
	El modelo de manejo de excepciones	579			
	Diseño de excepciones	580			
	Bloques try	581			
	Lanzamiento de excepciones	582			
	Captura de una excepción: catch	582			
20.6	Especificación de excepciones	583			
20.7	Excepciones imprevistas	586			
20.8	Aplicaciones prácticas de manejo de excepciones	588			
	Calcular las raíces de una ecuación de segundo grado	589			
	Control de excepciones en una estructura tipo pila	589			
	Resumen	591			
	Ejercicios	591			
CAPÍTULO 21					
De C/C++ a JAVA 6/7/8					
Introducción					
21.1	Historia del lenguaje Java: de Java 1.0 a Java 8	596			
21.2	El lenguaje de programación Java	598			
	Tipos de programas Java	599			
PARTE V					
Programar en Java					
595					

CAPÍTULO 24**Colecciones**

Introducción	689
24.1 Colecciones en Java	689
Tipos de colecciones	690
24.2 Clases de utilidades: Arrays y Collections	692
Clase Arrays	692
Ordenación de arrays	692
Clase Collections	695
24.3 Comparación de objetos:	697
Comparable y Comparator	697
Comparable	697
Comparator	697
24.4 Vector y Stack	698
Vector	698
Stack	699
24.5 Iteradores de una colección	700
Enumeration	700
Iterator	701
24.6 Listas	703
ArrayList	703
24.7 Colecciones parametrizadas	705
Declaración de un tipo parametrizado	705
Resumen	706
Ejercicios	706
Problemas	707

CAPÍTULO 25**Multitarea y excepciones**

Introducción	708
25.1 Manejo de excepciones en Java	708
25.2 Mecanismo del manejo de excepciones	709
El modelo de manejo de excepciones	710
Diseño de excepciones	711
Bloques <code>try</code>	712
Lanzamiento de excepciones	714
Captura de una excepción: <code>catch</code>	715
Cláusula <code>finally</code>	717
25.3 Clases de excepciones definidas en Java	719
Excepciones comprobadas	720
Métodos que informan de la excepción	721
25.4 Nuevas clases de excepciones	723
25.5 Especificación de excepciones	724
25.6 Multitarea	727
Utilización de la multitarea	727
25.7 Creación de hilos	728
Criterios a seguir para elegir cómo crear un hilo	730
25.8 Estados de un hilo, ciclo de vida de un hilo	730
25.9 Prioridad entre hilos	734
25.10 Hilos <code>daemon</code>	734
25.11 Sincronización	734
Resumen	736
Ejercicios	737

PARTE VI**Estructura de datos en C/C++ y Java**
(en el Centro de recursos en línea del libro
www.mhhe.com/uni/joyanespcju2e)**CAPÍTULO 26****Organización de datos dentro de un archivo en C****CAPÍTULO 27****Listas, pilas y colas en C****CAPÍTULO 28****Flujos y archivos en C++****CAPÍTULO 29****Listas, pilas y colas en C++****CAPÍTULO 30****Archivos y flujos en Java****CAPÍTULO 31****Listas, pilas y colas en Java****APÉNDICE A****Estructura de un programa en Java y C/C++.**
Entornos de desarrollo integrados

(Java 7, Java 8 y C++11)	739
A.1 Estructura general de un programa en C/C++ y Java	739
A.2 Proceso de programación en Java	740
Máquina virtual en Java (JVM)	740
Prácticas de programación	741
Kit de desarrollo de Java: JDK y JDK8	742
Direcciones de Oracle	742
A.3 Entornos de desarrollo integrados (Java, C y C++)	744
Herramientas de desarrollo	744
NetBeans	745
Eclipse	745
Otros entornos de desarrollo	745
A.4 Compilación sin entornos de desarrollo: Java y C/C++	744
Compiladores de Java	745
Compiladores de C++ (versión C++11 y futura C++14)	746

APÉNDICE B**Representación de la información
en las computadoras**

Representación de textos

Representación de valores numéricos

Representación de enteros

Representación de números reales

Representación de caracteres

Representación de imágenes

Representación de sonidos

APÉNDICE C**Códigos ASCII y UNICODE**

C.1 Código ASCII

C.2 Código Unicode

Referencias web

Bibliografía

APÉNDICE D**Palabras reservadas****de Java 5 a 8, C y C++11**

	D.1	Java	757
		Palabras reservadas de Java con significado especial	758
747		Palabras reservadas (keywords) de C/C++	758
		Palabras nuevas de C++ 11	759
		APÉNDICE E	
		Prioridad de operadores C/C++ y Java	760
		Bibliografía	
		(en Centro de recursos en línea: www.mhhe.com/uni/joyanespcju2e)	
752		Recursos de programación	
		(en Centro de recursos en línea: www.mhhe.com/uni/joyanespcju2e)	
		ÍNDICE ANALÍTICO	763

757

Prólogo

Introducción

Bienvenido a *Programación en C, C++, Java y UML*, 2^a edición. ¿Qué ha sucedido en el campo de la informática y de la computación en general desde 2010, año en que se publicó la primera edición? En el área de computación se han desplegado y consolidado numerosas innovaciones tecnológicas, hoy ya una realidad auténtica, tanto social como empresarial y de negocios (*Cloud Computing*, *Big Data*, Internet de las cosas, etc.). Y en el área de programación de computadoras, la Web 2.0 se ha consolidado, la programación web ha pasado a constituir una materia de estudio y de desarrollo profesional en los últimos semestres de las carreras de ciencias e ingeniería. Y en el caso particular de los lenguajes de programación y de la disciplina de programación, espina dorsal de los estudios de ingeniería de sistemas, ingeniería de telecomunicaciones, ingeniería electrónica..., ciencias, etc., cuando escribimos la primera edición, lo hicimos utilizando las versiones de C99 y C ANSI, C++98 y Java 5 y la joven versión de Java 6 y hoy día se han consolidado las nuevas e innovadoras versiones de **C++11**, **Java 7** y **Java 8** unidas a la actualización de la versión de C, **C11**.

En 2010 comenzaban a despegar las tendencias que hoy son ya una realidad: *cloud computing* (computación en la nube), *movilidad y tecnologías móviles* (celulares) con los teléfonos inteligentes y las tabletas (además de las consolas y los lectores de libros electrónicos), *medios sociales (social media)* que han penetrado en todas las áreas del saber y del conocimiento. En estos últimos años ha emergido una nueva tendencia y tecnología conocida como *big data* (macrodatos o grandes volúmenes de datos). Todas estas tendencias y tecnologías de la información han impulsado nuevas aplicaciones, especialmente, *aplicaciones web* y *aplicaciones móviles (apps)*, que han originado una necesidad creciente de nuevos profesionales del mundo de programación. Los nuevos desarrolladores requieren las más novedosas técnicas de programación y sobre todo el conocimiento y aprendizaje profundo de las tecnologías de programación basadas en los lenguajes de programación clásicos estructurados y orientados a objetos: C, C++ y Java, soporte nuclear e imprescindible que permita al lector y alumno poder llegar a realizar y desarrollar aplicaciones para la web o para entornos profesionales y de negocios en lenguajes como XML, JavaScript, Python o en sistemas operativos como Chrome, Android, MacOS, iOS, Firefox OS.

Por las razones anteriores hemos diseñado una segunda edición más ligera, para facilitar el aprendizaje multidisciplinar en lenguajes o secuencial a elección o decisión personal del maestro y profesor, y hemos potenciado la documentación de la web, de modo que el alumno o lector pueda disponer de una buena página web con recursos de todo tipo que le faciliten y ayuden en el aprendizaje para afrontar los retos a los que se enfrentará en los siguientes niveles de su formación. Hemos querido recoger las versiones actuales de los tres lenguajes de programación, espina dorsal de la obra, **C11**, **C++11** y **Java 7/Java 8** junto con la última versión de UML, la versión **UML 2.5**. Para ello hemos introducido en el texto y en los apéndices, así como en la página web las características fundamentales de estas versiones, dejando constancia que todo el libro se puede utilizar libremente por los usuarios de cualquier versión anterior, tanto de los lenguajes como de UML.

Hemos actualizado la primera edición con los conocimientos que hemos considerado imprescindibles para la formación que pretendemos conseguir y hemos aprovechado para revisar y corregir o actualizar los contenidos que habíamos incluido en la primera edición y requerían una revisión y estudio en profundidad. Por esta razón se han reducido el número de capítulos de la primera edición a fin de facilitar el aprendizaje de la programación, manteniendo los contenidos pero adaptándolas a los nuevos planes de estudios y procesos de aprendizaje que se dan en estos años de la segunda década de este siglo xxi.

Para esta segunda edición seguimos pensando en los nuevos estudiantes y autodidactas que se adentran en la formación de programación *a nivel universitario* o de *formación profesional*, con ningún conocimiento de programación de computadoras o con la base de cursos de introducción a algoritmos o a objetos. Desde el punto de vista de España y Latinoamérica, hemos pensado en los nuevos estudiantes de ingeniería de sistemas, ingeniería electrónica, ingeniería de telecomunicaciones o licenciaturas de informática o de ciencias que desean introducirse en el área de programación, bien por la necesidad de su carrera o como complemento en sus estudios de ciencias o de ingeniería. Desde el punto de vista español

o portugués, se ha pensado en los nuevos planes de estudio acogidos al Espacio Europeo de Educación Superior (EEES), más conocido como los nuevos estudios de la Declaración de Bolonia, por el que todos los países de la Unión Europea, y muchos otros que sin ser de la UE son europeos, han decidido tener unos contenidos comunes para intercambio de estudios y de carreras a partir del año 2010.

Todos los estudios de computación y de informática, tanto de Latinoamérica como de Europa, suelen contemplar asignaturas tales como *Introducción*, *Fundamentos* o *Metodología de Programación*, junto con estudios de *Estructuras de Datos* y de *Programación Orientada a Objetos*. Dependiendo de la institución académica y sobre todo del maestro y profesor, los estudios se realizan con base en algoritmos y el uso en su aprendizaje de un lenguaje de programación. C, C++ y Java son los lenguajes de programación por excelencia. Son muchos los maestros que optan por comenzar con C, como lenguaje de programación tradicional e histórico, luego siguen con C++ para terminar en Java. Otros lo hacen en forma simultánea con C++ y Java, y aquellos que incorporan ideas más innovadoras se inician directamente en Java para llevar a sus alumnos por el camino de la web. Nuestra experiencia de más de dos décadas de enseñanza de programación de computadoras y muchas obras escritas en este campo nos dice que todas las experiencias son positivas y que es la decisión del maestro, la mejor elección y, sin duda, la decisión más acertada.

Por estas razones, llevamos a cabo esta experiencia *multidisciplinar* y *multilenguaje*, y esta segunda edición es la consolidación y nueva plasmación de esa idea. Decidimos que estudiar los tres lenguajes de programación, primero independientes, luego interrelacionados y luego en paralelo, con la decisión correcta del maestro o autodidacta, en su caso, que decide la secuencia, el lenguaje, los lenguajes, en la forma que mejor se adapte a sus clases, era una propuesta razonable. La obra que tiene usted en su poder, recoge estas ideas fruto de nuestros muchos años de docencia y de investigación junto con la experiencia emanada de la primera edición. Naturalmente, necesitábamos también otras herramientas: libros de problemas y un sitio web de referencia, donde pudíramos añadir cursos, tutoriales, proyectos, recursos web, etc... Los libros de problemas complementarios son los que hemos publicado en la prestigiosa Serie Schaum de nuestra editorial McGraw-Hill y el sitio web es el **Centro de Recursos en Línea**, que puede consultar en www.mhhe.com/uni/joyanespcju2e).

Todas las carreras universitarias de ciencias e ingeniería, así como los estudios de formación profesional, requieren un curso básico de algoritmos y de programación con un lenguaje potente y profesional pero que sea simple y fácil de utilizar. C es idóneo para aprender a programar directamente las técnicas algorítmicas y de programación o bien en paralelo con asignaturas tales como *Introducción*, *Fundamentos* o *Metodología de la programación* cuando se utiliza un lenguaje algorítmico o un lenguaje de programación estructurada. C sigue siendo el lenguaje universal más utilizado y recomendado en planes de estudio de universidades y centros de formación de todo el mundo. Organizaciones como ACM, IEEE, colegios profesionales, siguen recomendando la necesidad del conocimiento en profundidad de técnicas y de lenguajes de programación estructurada con el objetivo de “acomodar” la formación del estudiante a la concepción, diseño y construcción de algoritmos y de estructuras de datos. El conocimiento profundo de algoritmos unido a técnicas fiables, rigurosas y eficientes de programación preparan al estudiante o al autodidacta para un alto rendimiento en programación y la preparación para asumir los retos de la programación orientada a objetos en una primera fase y las técnicas y métodos inherentes a ingeniería de software en otra fase más avanzada. Por estas razones, las partes I y II constituyen con sus doce capítulos, un primer curso de *Fundamentos de programación en C*.

Hoy día, no se entiende el estudio de ingeniería de sistemas, ingeniería informática o cualquier otra ingeniería o carrera de ciencias, sin estudiar objetos y modelado de sistemas orientado a objetos. Por esta razón, la parte III está constituida por un *primer curso de introducción a UML, el Lenguaje Unificado de Modelado*, por excelencia, y sin duda, el más utilizado no sólo en universidades e institutos tecnológicos y de formación profesional, sino cada vez más en escuelas de negocios, como herramienta de modelado en ambientes de negocios y de empresas.

Si el maestro ha decidido tomar como base C, se encontrará con que la próxima decisión será elegir, seguir sus clases ya en siguientes semestres, en C++ y/o en Java; por esta razón las partes IV y V son *cursos de programación orientada a objetos y clases con C++ y Java*, que se pueden estudiar en paralelo o de modo secuencial.

Por último y una vez que nuestros alumnos o autodidactas ya conocen los tres lenguajes de programación mejores para el aprendizaje y para el campo profesional, es el momento de iniciarse en *Estructuras de datos*¹ y por esta razón la parte VI consta de *cursos de introducción a estructuras de datos en C, en*

¹ Los autores tienen publicados en McGraw-Hill sendas obras de *Estructuras de datos en C++* y *Estructuras de datos en Java*.

C++ y en Java, organizados secuencialmente de modo que sea otra vez el profesor y maestro quien decide su estudio en paralelo o de modo secuencial. Esta opción se la ofrecemos a todos los lectores en el Centro de Recursos en Línea libro: <http://www.mhhe.com/uni/joyanespclju2e>.

El contenido del libro se ha pensado para dos semestres de modo muy intensivo, e incluso tres semestres, en este caso, de un modo más liviano y espaciado. El libro pretende servir, en primer lugar para asignaturas típicas de *Algoritmos, Introducción, Fundamentos o Metodología de la Programación*; en segundo lugar para asignaturas de *Programación Orientada a Objetos* y por último, pero no necesariamente de modo secuencial, para asignaturas de *Estructura de Datos*. Todo ello utilizando C, C++ y/o Java.

¿Qué necesita para utilizar este libro?

Esta obra se ha diseñado para enseñar métodos de escritura de programas útiles tan rápido y fácil como sea posible, aprendiendo tanto la sintaxis y funcionamiento del lenguaje de programación como las técnicas de programación y los fundamentos de construcción de algoritmos básicos. El contenido se ha escrito pensando en la posibilidad de que el lector sea:

- Una persona novata en la programación que desea aprender acerca de la programación y escritura de programas en C/C++ y Java desde el principio.
- Una persona con conocimientos básicos de programación que ha seguido cursos de iniciación en algoritmos como pueden ser nuestras obras de *Metodología de la programación* o *Fundamentos de programación*
- Una persona con conocimientos básicos de lenguajes de programación tales como C, C++ o Java pero que necesita interrelacionar los tres lenguajes con el objeto de llegar a adquirir un conocimiento profundo de los tres con objeto de aplicar eficientemente las características fundamentales de cada uno de ellos y utilizar de modo profesional el lenguaje que considera más idóneo para el desarrollo de su aplicación.
- El libro es eminentemente didáctico para la enseñanza sistematizada de la programación de computadoras, pero *no presupone ningún conocimiento previo de programación*, por lo que puede ser también utilizado por lectores autodidactas con o sin formación en informática o en ciencias computacionales.

Para utilizar este libro y obtener el máximo rendimiento, usted necesitará una computadora con un compilador de C/C++ y posteriormente de Java. Es deseable que tenga instalada una biblioteca de funciones de modo que se puedan ejecutar los ejemplos del libro y un editor de texto para preparar sus archivos de código fuente. Existen numerosos compiladores de C/C++ y Java en el mercado y también abundantes versiones *shareware* (libres de costos) disponibles en Internet. Idealmente, se debe elegir un compilador que sea compatible con la versión estándar de C/C++ del American National Standards Institute (ANSI), que es la versión empleada en la escritura de este libro. La mayoría de los actuales compiladores disponibles de C++, comerciales o de dominio público, soportan C, por lo que tal vez esta pueda ser una opción muy recomendable. En el caso de Java, las últimas versiones de compiladores puede descargarlas del sitio de Oracle, por lo que siempre tendrá la seguridad de utilizar un estándar. En cualquier forma, más adelante le recomendaremos los compiladores y fabricantes más populares, así como la mejor forma de descargar versiones gratuitas de la web.

Usted puede utilizar cualquier editor de textos, tales como *Notepad* o *Vi*, para crear sus archivos de programas fuente, aunque será mucho mejor utilizar un editor específico para editar código, como los que suelen venir con los entornos integrados de desarrollo, bien para Windows o para Linux. Sin embargo, no deberá utilizar un procesador de textos, tipo Microsoft Word, ya que normalmente los procesadores de texto o de tratamiento de textos comerciales, incrustan o “embeben” códigos de formatos en el texto que no entenderá su compilador.

En cualquier forma si usted es alumno, de cualquier nivel de enseñanza y sigue un curso sistematizado, el mejor método para estudiar este libro es seguir los consejos de su profesor tanto para su formación teórica como para su formación práctica. Si usted es autodidacta o estudia de modo autónomo, la recomendación entonces será que compile, ejecute y depure (limpie) de errores sus programas, tanto los propuestos en el libro, como los que diseñe, a medida que vaya leyendo el libro, tratando de entender la lógica del algoritmo y la sintaxis del lenguaje en cada ejercicio que realice.

El objetivo final que buscamos es, no sólo describir la sintaxis de los tres lenguajes de programación, sino, y sobre todo, mostrar las características más sobresalientes de ellos a la vez que se enseñan técnicas de programación estructurada y orientada a objetos y posteriormente las técnicas básicas de estructura de datos. Por consiguiente, las características fundamentales de esta obra son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Un medio de resolución de problemas mediante técnicas de programación.
- Actualización de contenidos al último estándar **ANSI/ISO C/C++, C11 y C++11, Java 7 y Java 8**, incluyendo las novedades más sobresalientes en los tres lenguajes de programación.
- Tutorial enfocado a los tres lenguajes, incluyendo numerosos ejemplos, ejercicios y herramientas de ayuda al aprendizaje.
- Descripción detallada del lenguaje respectivo, con un énfasis especial en técnicas de programación actuales y eficientes. Dado que hemos optado por iniciar el aprendizaje con un curso completo de C, hemos dedicados dos capítulos específicos para explicar las diferencias clave de C++ y Java comparadas con C, capítulos que hemos nombrado “De C a C++” (capítulo 16) y “De C/C++ a Java 6/7/8” (capítulo 21) antes de iniciar el aprendizaje ya más en profundidad de ambos lenguajes.
- El contenido se ha estructurado en diferentes partes siguiendo nuestra experiencia docente en el mundo de la programación y en nuestras obras similares, y sobre todo en una secuencialidad que consideramos beneficiará al alumno en su formación progresiva.
- Una introducción a la informática, a las ciencias de la computación y a los algoritmos y metodología de la programación.

En resumen, éste es un libro diseñado para enseñar a programar utilizando un lenguaje de programación y no un libro específico diseñado para enseñar C/C++ o Java, aunque también pretende conseguirlo. No obstante, confiamos que los estudiantes y autodidactas que utilicen la obra puedan conocer los tres lenguajes de programación y los conocimientos clave de UML, de modo que puedan aprender y conocer profesionalmente, tanto las técnicas clásicas y avanzadas de programación estructurada, como las técnicas orientadas a objetos y el diseño y construcción de estructura de datos. La programación orientada a objetos no es la panacea universal de programador del siglo XXI, pero le ayudará a realizar tareas que, de otra manera, serían complejas y tediosas y le facilitará el tránsito a los caminos que le conducirán a la programación de lenguajes de programación para la web, más específicos, como C# y los nuevos utilizados en las nuevas tecnologías de la **Web 2.0** y **AJAX**, tales como **JavaScript, XML, Phyton o Ruby**.

Objetivos

El libro pretende enseñar a programar una computadora utilizando varios lenguajes de programación estándares (C, C++ y Java), introducir al lector en el lenguaje de modelado de sistemas UML, enseñar a programar con el paradigma estructurado y el orientado a objetos, así como introducir al lector en el fundamental campo de las estructuras de datos. El contenido del libro sigue los objetivos iniciales de la primera edición: cubrir los contenidos de asignaturas, cuyos títulos dependen de los planes de estudio de la universidad y país correspondiente, tales como Fundamentos de programación I y II (dos semestres), Algoritmos y Estructuras de datos, Programación I (un semestre, en lenguaje C), Programación Orientada a Objetos y a UML, tanto en C++ como en Java (1/2 semestres dependiendo del empleo de C++ y/o Java) y desde el punto de vista práctico de lenguajes o para programación de profesionales, tecnológicos, formación profesional..., materias de un semestre utilizando lenguajes C, C++ o Java, es decir Programación en C, Programación en C++, Programación en Java.

Para ello se apoyará en los siguientes conceptos fundamentales:

1. Una revisión actual y básica de los *conceptos fundamentales de las computadoras, los lenguajes de programación y las herramientas y tecnologías de programación y metodología de la programación*.
2. *Algoritmos* (conjunto de instrucciones programadas para resolver una tarea específica).
3. *Sintaxis de los lenguajes de programación C, C++ y Java*, en sus versiones estándares en el caso de C/C++ (**C11** y **C++11**) y en las últimas versiones de Java; 6, 7 y 8.
4. *Orientación a Objetos* (datos y algoritmos que manipulan esos datos, encapsulados en un tipo de dato conocido como objeto) con la ayuda de los lenguajes C++ y Java.
5. *UML 2.5, notaciones y relaciones gráficas de clases y objetos* (iniciación al modelado de aplicaciones con una herramienta ya universal como es UML, lenguaje unificado de modelado).

6. Técnicas de programación en C, C++ y Java (aprendizaje fiable y eficiente del desarrollo de programas de aplicaciones).

El libro dispone de una página web (<http://www.mhhe.com/uni/joyanespcju2e>) y una complementaria “El portal tecnológico y de conocimiento” (www.mhe.es/joyanes), donde no sólo podrá consultar y descargarse códigos fuente de los programas del libro, sino también apéndices complementarios y cursos o tutoriales de programación en C, C++ y Java, especialmente pensados para los lectores sin conocimiento de este lenguaje, junto con tutoriales y ejercicios propuestos y resueltos de algoritmos utilizando herramientas tales como pseudocódigos y los clásicos diagramas de flujo.

1. Actividades de programación (ejercicios, test, programas complementario, entre otros) de modo que el alumno pueda utilizar durante su aprendizaje y de modo complementario con los ejemplos, ejercicios, programas etc., contenidos en el texto.
2. Resolución de ejercicios y problemas propuestos en el libro.
3. Propuesta y resolución de otros ejercicios y problemas complementarios no propuestos en el libro.
4. Cursos de programación en transparencias (*slides*) para facilitar el aprendizaje.
5. Recursos web de programación.
6. Libros, revistas, artículos de programación y de lenguajes de programación.
7. Test y ejercicios de autocomprobación.
8. Referencias y ejercicios de programación propuestos y resueltos, de otros libros de programación de los autores.
9. La parte VI: 6 capítulos que pueden constituir, por sí solos, un curso de introducción a *estructuras de datos* en C, C++ y Java.

Los 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 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. Por ejemplo, la nómina de una empresa o la gestión de ventas de un almacén, se representa 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 1980, 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 se modela un conjunto de abstracciones de datos. En C++ y Java, 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.

C++ y Java son lenguajes *multiparadigma*. Dan soporte a ambos tipos de paradigmas aunque su uso habitual y su gran potencia residen en la programación orientada a objetos. Así pues, y en resumen, los objetivos fundamentales de esta obra son: enseñanza y aprendizaje de *introducción a la programación estructurada y programación orientada a objetos* empleando los lenguajes de programación C, C++ y Java, junto con el uso de un lenguaje unificado de modelado como UML. Un objetivo complementario será el aprendizaje de las estructuras de datos, que esperamos cumplir a nivel medio con el contenido del Centro de recursos en línea del libro en donde se publicará una parte completa y extensa sobre el conocimiento avanzado de estructuras de datos.

La evolución de C

C fue una evolución de los lenguajes **BCPL** y **B**, desarrollados en la década de 1960, por Martin Richards como lenguajes para escribir sistemas operativos y compiladores. Posteriormente Ken Thompson utilizó B para crear las primeras versiones del sistema operativo **UNIX** en Bell Laboratories, sobre una compu-

tadora DEC PDP-7 de Digital. El lenguaje C, como tal, fue una evolución directa del lenguaje B y fue creado en 1972 por Dennis Ritchie, con la colaboración de Ken Thompson, también en Bell Laboratories y se implementó originalmente la mítica computadora **DEC PDP-11**.

En 1978, Ritchie y Brian Kernighan, publicaron la primera edición de *El lenguaje de programación C*, primera especificación no formal del lenguaje C y que se ha conocido tradicionalmente como el “**K&R C**” (el C de *Kernighan y Ritchie*). En 1983, el American National Standards Institute (ANSI) creó el comité X3J11 cuyo objetivo era definir una especificación estándar de C. En 1989, fue aprobado el estándar ANSI X3159:1989. Esta versión se conoce como **ANSI C** o bien **C89**.

En 1990, la International Standard Organization (ISO) adoptó como estándar el ANSI C, con la denominación de estándar ISO/IEC 9899: 1990, que se le conoce como **C90** o **ANSI/ISO C**, aunque ambas versiones, ANSI C y ANSI/ISO C, son formalmente las mismas.

El lenguaje C siguió evolucionando y en 1999, ante el avance de C++, se revisó y actualizó y se publicó un nuevo estándar INCITS/ISO/IEC 9899:1999, conocido también como **C99**.

El lenguaje C ha ido evolucionando y su última especificación aprobada por ISO/IEC ha sido la versión **ISO/IEC 9899:2011**, conocida popularmente como **C11** (antes C1X), que fue aprobada en 2011. Esta versión se ha implementado ya en casi todos los compiladores y plataformas de compilación tanto gratuitas como de pago.

C sigue siendo un lenguaje de programación muy utilizado en la formación de programadores e ingenieros, y su fortaleza reside en que ha sido el referente y modelo de lenguajes como **Objective-C** (utilizado hoy día en plataformas de Apple, tales como el sistema operativo MacOS y el iOS 6 e iOS 7 para dispositivos móviles), **C++, Java y C#**.

La evolución de C++

El aprendizaje de C++ es una aventura de descubrimientos, en especial porque el lenguaje se adapta muy bien a diferentes paradigmas de programación, incluyendo entre ellos, *programación orientada a objetos*, *programación genérica* y la tradicional *programación procedimental o estructurada*. C++ ha ido evolucionando desde la publicación del libro de Stroustrup: *C++. Manual de referencia con anotaciones* (conocido como ARM y traducido al español por el coautor de este libro, profesor Joyanes y por el profesor Miguel Katrib de la Universidad de La Habana) hasta llegar a la actual versión de su obra clave: *The C++ Programming Language*, cuarta edición, que incluye C++11, y que fue publicada por Addison-Wesley en mayo de 2013.

Esta edición sigue el estándar ANSI/ISO, aunque en algunos ejemplos y ejercicios, se ha optado por mantener el antiguo estándar, a efectos de compatibilidad, con otras versiones antiguas que pueda utilizar el lector y para aquellos casos en que utilice un compilador no compatible con el estándar, antes mencionado. Aprenderá muchas características de C++, e incluso las derivadas de C, a destacar:

- Clases y objetos
- Herencia
- Polimorfismo y funciones virtuales
- Sobrecarga de funciones y de operadores
- Variables referencia
- Programación genérica, utilizando plantillas (templates) y la biblioteca de plantillas estándar (STL, Standard Template Library)
- Mecanismo de excepciones para manejar condiciones de error
- Espacio de nombres para gestionar nombres de funciones, clases y variables

C++ se comenzó a utilizar como un «*C con clases*» y fue a principios de la década de 1980 cuando se inició la revolución C++, aunque su primer uso comercial, por parte de una organización de investigación, fue en julio de 1983. Como Stroustrup cuenta en el prólogo de la 3^a edición de la 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 (véase Pascal, BASIC, FORTRAN...). La explosión del lenguaje en la comunidad informática hizo inevitable la estandarización, proceso que comenzó en 1987 [Stroustrup 94]. Así nació una primera fuente de estandarización: *The Annotated C++ Reference Manual* [Ellis 89].² En diciembre de

² Existe versión española de Addison-Wesley Díaz de Santos y traducida por los profesores Manuel Katrib y Luis Joyanes.

1989 se reunió el comité X3J16 de ANSI, bajo el auspicio de Hewlett-Packard, y en junio de 1991 se realizó el primer esfuerzo de estandarización internacional de la mano de ISO, y así comenzó a nacer el estándar ANSI/ISO 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 fue en 1998 cuando el proceso se pudo dar por terminado (*ANSI/ISO C++ Draft Standard*) conocido como **ISO/IEC 14882: 1998** o simplemente **C++ Estándar** o **ANSI C++**. Stroustrup publicó en 1997 la tercera edición de su libro *The C++ Programming Language* [Stroustrup 97] y en 2000, una actualización que se publicó como *edición especial*.³

C++ sigue evolucionando gracias a su creador Stroustrup y el comité de estandarización de C++. Las especificaciones del nuevo estándar que fueron aprobadas el 12 de agosto de 2011, norma ISO/IEC 14882: 2011, y se conoce popularmente como **C++11**, ha supuesto una gran revolución dado que si bien soporta las versiones anteriores, es más genérico, más uniforme y con mejores mecanismos de abstracción. Incluye mejoras significativas pensadas en programación profesional y en la fuerte competencia de Java y C#.

La evolución de Java: Java 2, Java 5, Java 6, Java 7 y, "en camino", Java 9

James Gosling y otros desarrolladores de Sun Microsystems (empresa creadora de Java, vendida posteriormente a Oracle), trabajaban, a principios de la década de 1990, en un proyecto que deseaba diseñar un lenguaje de computadoras que pudiera ser utilizado para dispositivos de consumo tales como aparatos de televisión y otros componentes electrónicos. Los requisitos eran simples: un lenguaje portable que pudiera ser utilizado en cualquier máquina que tuviera el intérprete adecuado. El proyecto se llamaba "Green" y el lenguaje pretendía generar código intermedio para una máquina hipotética. Este código intermedio debía poder utilizarse en cualquier máquina que tuviera el intérprete correcto. El proyecto utilizaba una máquina virtual, pero los ingenieros procedían de trabajar en entornos UNIX con el lenguaje C++ y por esta razón era un lenguaje orientado a objetos y no a procedimientos. En paralelo a los desarrollos de Sun, la World Wide Web, desde 1993, estaba creciendo a marchas agigantadas. La clave para la web es el navegador que traducía la página del hipertexto a la pantalla. En 1994, la mayoría de los usuarios de Internet utilizaban Mosaic, un navegador web no comercial.

El proyecto Green se transformó en "First Person, Inc", el cual presentó algunos resultados de éxito, pero se disolvió en 1994; sin embargo, comenzó a utilizarse el lenguaje que primero se llamó "Oak" y posteriormente usó Java en el diseño de navegadores, en concreto en un navegador llamado HotJava. El lenguaje fue escrito en Java para mostrar toda la potencia de la que eran capaces. Ellos construyeron aplicaciones que llamaron "applets" e hicieron el navegador capaz de ejecutar código en el interior de páginas web. Esta experiencia se mostró en el SunWorld '95 y fue el comienzo de la historia de Java y el inicio de la leyenda de este lenguaje para desarrollo en Internet.

Sun lanzó la primera versión de Java a principios de 1996 y se le denominó Java 1.0 y desde entonces ha lanzado siete importantes revisiones del Java Development Kit (JDK) y la Interfaz de Programación de Aplicaciones (API) ha crecido desde alrededor de 200 clases a más de 3 000 clases. La API ahora genera áreas tan diversas como construcción de interfaces de usuario, gestión de bases de datos, internacionalización, seguridad y procesamiento XML. En 1997, se lanzó la primera revisión, 1.1, que ya contenía clases internas. En diciembre de 1998 con ocasión de la conferencia JavaOne se presentó la versión 1.2 y se cambió el nombre comercial a Java 2 Standard Edition Software Development Kit Versión 1.2 (se le conoció como Java 2 y hoy sigue siendo muy utilizada).

Las versiones 1.3 (año 2000) y 2.4 (año 2004) de la Standard Edition fueron mejoras incrementales sobre la versión inicial de Java 2. Con la versión 2 Sun comenzó a renombrar las versiones de Java con las siglas J2SE (Java 2 Platform Standard Edition) y las siguientes versiones fueron llamadas J2SE 1.3 y J2SE 1.4. La siguiente versión de Java fue J2SE 5 y fue revolucionaria todavía fue conducida por su empresa creadora, Sun Microsystems. A finales de 2004 se presentó la versión Java 5.0 (originalmente se le llamó versión 1.5), que es la primera versión que actualiza el lenguaje Java desde la versión 1.1 de modo significativo (añadió entre otras cosas "clases genéricas", el bucle "for each", varargs, autoboxin, metadatos, enumeración e importación estática). Las clases genéricas eran similares a las plantillas (*templates*)

³ Esta obra fue traducida por un equipo de profesores universitarios de la Facultad de Informática de la Universidad Pontificia de Salamanca en Madrid que dirigió y coordinó el profesor Luis Joyanes, coautor de esta obra.

de C++ y las características “*bucle for-each*”, “*autoboxing*” y *metadatos* las incorporó C#, el lenguaje creado por Microsoft a principios de la década de 2000 para competir con Java.

A finales de 2006, se lanzó la versión 6 (sin el sufijo .0) que no trajo avances en el lenguaje sino mejoras adicionales en prestaciones y en bibliotecas. Esta versión ya se llamaba JSE 6 y se construyó añadiendo mejoras incrementales a J2SE 5.

La versión más extendida de Java se llama Java SE 7 con el JDK (Java Developer’s Kit), llamado JDK 7. Java SE 7 (o simplemente Java 7) es la primera versión importante desde que Oracle compró Sun Microsystems en abril de 2009 (se terminó la operación de compra en enero de 2010). La versión Java SE 7 se lanzó en 2011 e incorporó muchas nuevas características, incluyendo adiciones significativas al lenguaje y a las librerías API. El 14 de enero de 2014 se lanzó la actualización número 51 (Java SE 7 update 51).

El 25 de marzo de 2014, Oracle anunció oficialmente en una nota de prensa del lanzamiento de **Java 8** y la disponibilidad de descargas de los paquetes Java SE8 y Java ME8 para el desarrollo de aplicaciones en Java.

Compiladores y compilación de programas: Entornos de Desarrollo Integrados (IDE) C/C++

Existen numerosos compiladores de C/C++ en el mercado de código propietario o de código abierto, gratuitos o de pago. En el caso de C++ vamos a considerar en primer lugar el consejo que Bjarne Stroustrup creador de C++ da en su propia página web para aquellos que se inician o ya programan profesionalmente en C++.

NOTA PRÁCTICA DE C++ ESTÁNDAR (Bjarne Stroustrup): www.stroustrup.com

En el caso de C/C++, su inventor, Stroustrup, recomienda en su página web una regla de compatibilidad de compiladores y que nosotros lógicamente también aconsejamos, por venir de la mano del genial inventor. Stroustrup aconseja compilar con su compilador, el sencillo programa fuente C++ siguiente. Si usted compila bien este programa no tendrá problemas con C++ estándar, en caso contrario aconseja buscar otro compilador que sea compatible.

```
#include<iostream>
#include<string>

using namespace std;

int main( )
{
    string s;
    cout << "Por favor introduzca su nombre seguido por Intro \n";
    // en el original "Please enter your first name followed by a // newline\n";
    cin >> s;
    cout << "Hola, " << s << '\n';
    // en el original "Hello, " y "this return statement isn't
    // necessary"
    return 0; // esta sentencia return no es necesaria
}
```

Java

El lector debe comenzar por aprender a instalar el Kit de Desarrollo de Java JDK (Java Development Kit) que le permitirá compilar y ejecutar diferentes tipos de programas: programas de consola, aplicacio-

nes gráficas y *applets*. Se pueden ejecutar las herramientas JDK tecleando órdenes en una ventana *shell*. Sin embargo, la mayoría de los programadores, y eso es lo que recomendamos, prefieren la comodidad del entorno integrado de desarrollo. A continuación, le indicamos cómo puede instalar y utilizar gratuitamente el entorno integrado de desarrollo.

Entornos integrados de desarrollo

Existen numerosos entornos integrados de desarrollo (IDE, *Integrated Development Environment*) de diferentes fabricantes en software propietario y en software libre, que soportan todo el proceso de desarrollo de software incluyendo editores para escribir y editar programas, compiladores, intérpretes y depuradores para localizar y corregir errores lógicos. Algunos de los IDE más populares son los siguientes:

C/C++

DevC++ (www.bloodshed.net)
Borland C++ Builder (www.borland.com)
Visual C++ Studio (www.microsoft.com)
Code::Block C++ (www.codeblocs.org)
Jcreator (www.jcreator.com)

Java

NetBeans (EID de Sun)	www.netbeans.org
Eclipse	www.eclipse.org
JBuilder, del fabricante Borland	www.borland.com
BlueJ (EID, gratuito y muy popular)	www.bluej.org

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++* (primera edición), *Fundamentos de programación* (en su cuarta edición), *Programación en C* (en su segunda edición), *Programación en Pascal* (en su cuarta edición), y *Programación en BASIC* (que alcanzó tres ediciones y numerosísimas reimpresiones en la década de 1980) nos ha llevado a mantener la estructura de esta obra, actualizándola a los contenidos que se prevén para los estudiantes del actual 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 grado en ingeniería informática, matemáticas, físicas, ..., sino también de grados en ingenierías tales como industriales, telecomunicaciones, agrónomos o geodesia. Asimismo, nuestro conocimiento del mundo educativo latinoamericano nos ha llevado a tener en cuenta también las carreras de ingeniería de sistemas computacionales y las licenciaturas en informática y en sistemas de información, como se las conoce en el continente americano.

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 para currículas actuales y las vigentes en universidades latinoamericanas, muchas de las cuales conocemos y con las que tenemos relaciones profesionales.

El contenido del libro abarca los citados programas y comienza con la introducción a la computación y a la programación, para llegar a estructuras de datos y objetos. 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 impreso se ha organizado en cinco partes con numerosos apéndices incluidos en el centro de recursos en línea, con objeto de que el lector seleccione y descargue aquellos apéndices que considere de su interés, y de este modo no incrementar el número de páginas de la obra impresa.

Se trata de describir los dos paradigmas más populares en el mundo de la programación: el *procedimental* y el *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, han introducido a sus alumnos en la programación orientada a objetos, inmediatamente después del conocimiento de la programación estructurada, e incluso, en ocasiones, antes o en paralelo. 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 y programación orientada a objetos que constituyen las cinco partes del libro impreso.

Características importantes del libro

Programación en C, C++, Java y UML, 2^a edición utiliza los siguientes elementos clave para 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 en orden alfabético los términos informáticos y de programación más notables que se tratarán en el capítulo.

Descripción del capítulo. Explicación 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, en su caso, las explicaciones del profesor relativas al capítulo.

Problemas. Despues del apartado Ejercicios, se añade 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.

Recuadros. En ellos aparecen conceptos importantes que el lector debe considerar durante el desarrollo del capítulo.

Consejo. Son ideas, sugerencias y 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.

Recursos. En esta edición y en los capítulos adecuados se han incluido listados de recursos bibliográficos y de web.

¿Cómo está organizado el libro?

Aprovechando la experiencia de la primera edición y con base en las realimentaciones recibidas de diferentes personas en distintos lugares, se ha decidido mantener las cinco partes originales de la 1^a edición pero estructuradas de un modo **más ágil y dinámico de modo** que los 30 capítulos originales se han convertido en 25. La reducción del número de capítulos pretende facilitar la lectura, el aprendizaje y la asimilación de contenidos para los lectores y profesores/maestros que recomiendan el libro como texto académico, aunque conservando el mismo contenido teórico-práctico de la primera edición. Se han potenciado las actividades de programación en el centro de recursos en línea de modo que el alumno pueda progresar más rápida y eficazmente en su lectura con la ayuda de las citadas actividades que pretenden una evaluación más continua y sistemática del alumno a medida que avance en los sucesivos capítulos, tanto en el aspecto teórico como en la propuesta y resolución de ejercicios de comprobación del progreso del estudio de los sucesivos capítulos.

Con esta reestructuración, pretendemos hacer más homogéneo el libro para facilitar la distribución temporal en bimestres/trimestres/semestres en el función del trabajo personal del alumno o las directrices pedagógicas del maestro/profesor que imparte la asignatura apoyándose en el libro.

Los capítulos 1 y 2 se han actualizado totalmente con los recursos de *hardware* y *software* utilizados hoy día (año 2014 y siguientes), en organizaciones y empresas así como con las innovaciones tecnológicas de mayor impacto.

Asimismo, se han incluido las característica fundamentales de las últimas versiones de los lenguajes C, C++ y Java, o sea: C11, C++ 11, Java 7 y Java 8.

Los apéndices recogen las herramientas y documentación necesaria para complementar el aprendizaje del lector en los diferentes lenguajes de programación, así como guías, tablas, etc., de utilidad.

El libro se divide en seis partes (incluida la parte VI que incluye capítulos en el centro de recursos en línea) que unidas constituyen un curso completo ,*nivel de iniciación y medio*, de programación en C/C++ y Java (*algoritmos, programación estructurada y orientada a objetos, y estructuras de datos*) dividido en cuatro partes: Introducción a la computación y a la programación, curso de programación en C, curso de programación en C++ con énfasis en programación orientada a objetos, programación en Java 6/7/8, y cursos de introducción a estructuras de datos en C++ y Java, junto con un curso de introducción al modelado usando UML 2. 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, tanto en el paradigma procedimental como en el orientado a objetos. Los apéndices contienen un conjunto de temas importantes que incluyen desde guías de sintaxis de ANSI/ISO C++ hasta un glosario 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 (URL) donde el lector podrá complementar, ampliar y profundizar en el mundo de la programación y en la introducción a la ingeniería de software.

Todos los capítulos siguen una estructura similar que respeta el formato de la primera edición. Así, cada capítulo comienza con un extracto del contenido, una breve introducción al capítulo y una lista de los conceptos y términos más importantes del capítulo. La descripción teórica y práctica se acompaña de numerosos ejemplos y ejercicios prácticos con el objetivo fundamental de que el lector/alumno vaya aprendiendo a la par que sigue y estudia su contenido. A continuación se incluye un resumen con un recordatorio de los conceptos teóricos y/o prácticos más importantes del capítulo. Por último se incluyen una serie de descripciones de ejercicios de nivel de iniciación y medio, junto con otra lista de enunciados de problemas propuestos, con el objetivo fundamental que el lector/alumno practique y vea la progresión realizada con el seguimiento del libro así como de sus clases teórico-prácticas en su centro de enseñanza.

El libro, como ya se ha comentado, pretende facilitar la lectura y estudio por parte del lector/alumno o bien por la organización del profesor para la impartición de sus clases, de modo que pueda organizarlas en la manera que mejor considere para conseguir la máxima eficacia y el objetivo final de un aprendizaje correcto y el dominio de la programación en C, C++ y Java, tanto a nivel de iniciación como a nivel medio, que le faculten para iniciar su profesión de “programador de computadoras” o bien pasar a “cursos de nivel avanzado de programación en C, C++ o Java” o inclusive de programación de sistemas. Así mismo una de las partes trata de introducir al lector en UML (Lenguaje Unificado de Modelado) de modo que el lector se inicie en el modelado de objetos para su aplicación en programación orientada a objetos y en estructuras de datos y que le sirva de base para cursos completos de UML y avanzados de programación orientada a objetos.

Parte I Resolución de problemas con programación de computadoras

Capítulo 1. Fundamentos de computadoras y lenguajes de programación. Explica y describe los conceptos fundamentales de la computación y de los lenguajes de programación en la década del siglo xxi. El conocimiento completo del contenido del capítulo no es requisito imprescindible para el conocimiento y aprendizaje de los restantes capítulos aunque sí es fundamental en su formación

de programación. Por estas circunstancias el lector/alumno puede optar por su estudio en una sola vez o bien graduar el aprendizaje a lo largo de todo su curso de programación. Siempre se cuenta también con el profesor y maestro que podrá orientar al alumno en su progresión docente. Para los lectores que no hayan recibido ningún curso de introducción a las computadoras o a la informática les recomendamos su lectura antes de pasar al siguiente capítulo y luego realizar una relectura en el momento y forma que ellos mismos consideren.

Este capítulo, todos los conceptos, componentes y software básicos han sido actualizados a los existentes en el mercado y en los laboratorios de programación de los años 2013 y 2014, así como las últimas innovaciones tecnológicas de mayor impacto que más afectan al mundo de la programación. Se describen tecnologías tan implantadas en la actualidad como *cloud computing* y *big data*.

Capítulo 2. Algoritmos, programas y metodología de la programación. Se introduce al lector en los conceptos fundamentales de algoritmos y sus herramientas de representación. Asimismo se describen los tipos clásicos de programación con especial énfasis en la programación y también en la programación orientada a objetos. Esta parte está pensada esencialmente para alumnos principiantes en asignaturas de algoritmos e introducción a la programación o en su defecto como recordatorio para alumnos que hayan cursado dicha asignatura.

Parte II Programación en C

Capítulo 3. El lenguaje C, elementos básicos. Se describe el lenguaje C y sus elementos básicos. Introduce a la estructura y los componentes principales de un programa en C. Comprende los significados de los elementos fundamentales de todo programa, tales como datos, constantes, variables y las operaciones básicas de entrada/salida.

Capítulo 4. Operadores y expresiones. Se aprende el uso de los operadores aritméticos, relacionales y lógicos para la manipulación de operaciones y expresiones en C. Se estudian también operadores especiales y conversiones de tipos, junto con reglas de prioridad y asociatividad de los operadores en las expresiones y operaciones matemáticas.

Capítulo 5. Estructuras de selección: sentencias if y switch. Introduce a las sentencias de selecciones básicas y fundamentales en cualquier programa. Se examina el uso de sentencias compuestas o bloques así como el uso de operadores condicionales y evaluación de expresiones lógicas.

Capítulo 6. Estructuras de control: bucles. Se aprende el concepto de *bucle* o *lazo* y el modo de controlar la ejecución de un programa mediante las sentencias *for*, *while* y *do-while*. También se explica el concepto de anidamiento de bucles y bucles vacíos; se proporcionan ejemplos útiles para el diseño eficiente de bucles.

Capítulo 7. Funciones y recursividad. Examina las funciones en C, una parte importante de la programación. Aprende programación estructurada, un método de diseño de programas que enfatiza en el enfoque descendente para la resolución de problemas mediante la descomposición

del problema grande en problemas de menor nivel que se implementan a su vez con funciones. También se describen los fundamentos de la recursividad, técnicas clave en el diseño y construcción de algoritmos y programas.

Capítulo 8. Arreglos (arrays), listas y tablas. **Cadenas.** Explica un método sencillo pero potente de almacenamiento de datos. Se aprende cómo agrupar datos similares en *arrays* o “*arreglos*” (listas y tablas) numéricas. Se describe el concepto de cadena (*string*) así como las relaciones entre punteros (apuntadores), *arrays* y cadenas en C.

Capítulo 9. Algoritmos de ordenación y búsqueda. Enseña los métodos para ordenar listas y tablas, así como búsqueda de datos en éstas. Se estudian los algoritmos clásicos más sencillos y eficientes tanto de ordenación como de búsqueda.

Capítulo 10. Estructuras y uniones. Se describen conceptos básicos de estructuras, uniones y enumeraciones: declaración, definición, iniciación, uso y tamaño. Las operaciones fundamentales de acceso a estructuras, *arrays* de estructuras y estructuras anidadas se analizan también en este capítulo. En el capítulo se muestra de modo práctico como usar estructuras y uniones para conseguir las necesidades del programa; se explican las diferencias entre estructuras y uniones, así como el uso de la palabra reservada *typedef*.

Capítulo 11. Apuntadores (punteros). Presenta una de las características más potentes y eficientes del lenguaje C, los apuntadores (punteros). Este capítulo proporciona explicación detallada de los apuntadores, *arrays* (arreglos) de apuntadores, apuntadores de cadena, aritmética de apuntadores, apuntadores constantes, apuntadores como argumentos de funciones, apuntadores a funciones y a estructuras. Se describe la gestión dinámica de la memoria y las funciones asociadas para esas tareas: *malloc()*, *free()*, *calloc()* y *realloc()*. Se proporcionan reglas de funcionamiento de esas funciones y reglas para asignación de memoria.

Capítulo 12. Entradas y salidas por archivos. Se estudia el concepto de flujo (*stream*) y los diferentes métodos de apertura de archivos, junto con los conceptos de archivos binarios y funciones para el acceso aleatorio. Muestra de un modo práctico cómo C utiliza los flujos, examina los flujos predefinidos y el modo práctico de trabajar con la pantalla, la impresora y el teclado.

Parte III Lenguaje unificado de modelado: UML 2.5

Capítulo 13. Programación orientada a objetos y UML 2.5. Con este capítulo se inicia el estudio del

Lenguaje Unificado de Modelado (UML). Se describen las características fundamentales de la programación orientada a objetos y cómo se modelan e identifican los objetos, así como las propiedades fundamentales de la orientación a objetos.

Capítulo 14. Diseño de clases y objetos. Representaciones gráficas en UML. Los conceptos fundamentales de clases y objetos, junto con constructores y destructores y cómo se representan gráficamente las clases y objetos en UML 2.5.

Capítulo 15. Relaciones entre clases: delegaciones, asociaciones, agregación y herencia. Uno de los conceptos clave en el modelado orientado a objetos es la relación. Las relaciones entre clases y sus diferentes tipos (generalización, especialización, delegación, agregación, asociación, etc.) se estudian con ejemplos prácticos y sus respectivas notaciones gráficas en UML 2.5.

Parte IV Programar en C++

Capítulo 16. Fundamentos de C a C++. Enseña los fundamentos de C++, organización y estructura general de un programa, función `main()`, ejecución, depuración y prueba de un programa, elementos de un programa, tipos de datos (el tipo de dato `bool`), constantes, variables y entradas/salidas de datos (`cin` y `cout`). Se describen los conceptos fundamentales del lenguaje C++, haciendo énfasis en las diferencias clave con los conceptos aprendidos de C. En esencia, el capítulo se ha escrito como un breve curso de introducción a la programación en C++, y que debe servir de fundamentos para el aprendizaje de los siguientes capítulos que se centran esencialmente en programación orientada a objetos y en propiedades específicas de C++ como *excepciones* y *sobrecarga de operadores*.

Capítulo 17. Clases y Objetos. Sobrecarga de operadores. Este capítulo muestra la forma de implementar la abstracción de datos, mediante tipos abstractos de datos, clases. Se describen los conceptos de clase y objeto, así como el sistema para definición de una clase. El capítulo examina el método de inicialización de objetos, junto con el concepto de constructores y destructores de una clase.

Capítulo 18. Clases derivadas: herencia y polimorfismo. Dos de las propiedades más importantes de la programación orientada a objetos son: *herencia* y *polimorfismo*. La herencia es un sistema de *reusabilidad* de software en el que nuevas clases se desarrollan rápida y fácilmente a partir de las clases existentes y añadiendo nuevas propiedades a las mismas. Este capítulo examina las nociones de clases base y clases derivadas, herencia *protegida, pública y privada* (`protected`, `public`, `private`), constructores y destructores en clases base

y derivadas. Se describen los diferentes tipos de herencia: simple y múltiple. El *polimorfismo* y la *ligadura dinámica* se describen también a lo largo del capítulo.

Capítulo 19. Genericidad: plantillas (templates). Examina una de las incorporaciones más importantes de la evolución del lenguaje C++: las plantillas (*templates*). Se describen el concepto y la definición de las plantillas de funciones. Las plantillas de clases denominadas tipos *parametrizados* permiten definir tipos genéricos tales como una cola de enteros, una cola de reales (`float`), una cola de cadenas, etc. Se presenta una aplicación práctica y se realiza una comparativa de las plantillas y el polimorfismo.

Capítulo 20. Excepciones. Se examina en este capítulo una de las mejoras más sobresalientes del lenguaje C++. El manejo de excepciones permite al programador escribir programas que sean más robustos, más tolerantes a fallos y más adecuados a entornos de misión y negocios críticos. El capítulo introduce a los fundamentos básicos del manejo de excepciones con bloques `try`, sentencias `throw` y bloques `catch`; indica cómo y cuándo se vuelve a lanzar una excepción y se incluyen aplicaciones prácticas de manejo de excepciones.

Parte V. Programar en Java

Capítulo 21. De C/C++ a Java 6/7/8. Este capítulo constituye, al igual que el capítulo 19 para C++, un breve curso de introducción al lenguaje Java describiendo los elementos fundamentales de un programa en Java.

Capítulo 22. Programación orientada a objetos en Java. Clases y objetos. En este capítulo se describen los conceptos ya conocidos de clases y objetos, y cómo se implementan en Java, junto con los conceptos nuevos de paquetes y recolección de objetos.

Capítulo 23. Programación orientada a objetos en Java. Herencia y polimorfismo. Las ya conocidas clases derivadas se vuelven a estudiar en este capítulo desde el enfoque de Java. Asimismo se describen las propiedades clásicas de herencia y polimorfismo.

Capítulo 24. Colecciones. Colección es una concepto importante en Java junto con iteradores, conjuntos, mapas y diccionarios. Los iteradores e interfaces son otros conceptos importantes en Java y que se estudian en el capítulo.

Capítulo 25. Multitarea y excepciones. Dos propiedades muy importantes se describen en este capítulo. El ya conocido concepto de excepción y la propiedad que posibilita la multitarea mediante hilos de ejecución.

Apéndices

Se incluyen componentes de sintaxis y programación de los lenguajes fundamentales del libro:

- A. Estructura de un programa en C/C++ y Java
- B. Representación gráfica de la información en las computadoras Códigos ASCII y UNICODE
- C. Palabras reservadas de C, C++ y Java
- D. Prioridad de operadores C/C++ y Java
- E. Guía rápida de UML 2.1

Parte VI. Estructuras de datos en C/C++ y Java (Centro de recursos en línea del libro)

Capítulo 26. Organización de datos dentro de un archivo en C. La organización clásica de los datos en archivos es el tema central de este capítulo. Se hace una breve introducción a la ordenación y clasificación de archivos.

Capítulo 27. Listas, pilas y colas en C. El estudio de las estructuras básicas de datos: listas, pilas y colas, son los conceptos clave explicados en este capítulo. Se realiza una introducción básica a dichas estructuras y se estudia su codificación en C.

Capítulo 28. Flujos y archivos en C++. El diseño e implementación de los flujos (*stream*) y archivos en C++ es el motivo central de este capítulo.

Capítulo 29. Listas, pilas y colas en C++. La implementación de las estructuras de datos básicas en C++ se describen en el capítulo.

Capítulo 30. Archivos y flujos en Java. La codificación de los archivos y flujos en Java se describen en el capítulo.

Capítulo 31. Listas, pilas y colas en Java.

Los importantes conceptos de listas, pilas y colas, ya estudiados anteriormente, se estudian y sus algoritmos y clases se codifican en Java.

Bibliografía de C/C++, Java y UML
Recursos de programación

**Centro de recursos en línea
(en página web del libro:
www.mhhe.com/uni/joyanespcj2e)**

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 de su contenido. Estos apéndices sirven de guía y referencia de elementos importantes del lenguaje y de la programación de computadoras. En la página del libro en la web, se han incluido numerosos apéndices, además de los incorporados en la edición en papel, de modo que el lector pueda “bajarlos” de ella cuando lo considere oportuno y en función de su lectura y aprendizaje. Se incluyen esencialmente:

*Guías de sintaxis de C, C++ (v.11) y Java 7/8
Guía rápida de referencia de UML 2.1*

Biblioteca de funciones de C, C++ y Java 7/8

Biblioteca de clases de C++ y Java

Biblioteca estándar STL de C++

Glosario

Cursos de introducción a la computación y a la programación

Cursos de algoritmos y estructuras de datos

Tutoriales

Guía de buenas prácticas de programación en C/C++ y Java 7/8

Bibliografía y Recursos web actualizados

Hemeroteca

Agradecimientos

A nuestros compañeros de la Facultad de Informática y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el *campus* de Madrid, que revisaron las primeras pruebas de esta obra y nos dieron consejos sobre las mismas:

- **Dr. Lucas Sánchez García**
- **Dra. Matilde Fernández Azuela**

Gracias, compañeros, y sin embargo, amigos.

En esta ocasión hemos de destacar un agradecimiento especial a nuestros editores mexicanos: **Ana Delgado** (editora de desarrollo), **Marcela Rocha** (coordinadora editorial) y **Jesús Mares** (mi nuevo editor *sponsor*). Gracias amigos. Sin embargo, no podemos dejar de acordarnos y agradecer, como siempre, el apoyo de nuestra editora en Madrid, **Cristina Sánchez**, quien nos ha prestado todo tipo de ayuda para que este proceso editorial tuviera éxito; como en otras ocasiones, los editores de ambos lados del Atlántico, nos han apoyado y sobre todo nos han facilitado el *camino virtual Madrid-Ciudad de México*, y sus consejos también nos han ayudado considerablemente.

De un modo muy especial y con nuestro agradecimiento eterno, a nuestros lectores, a los estudiantes de Hispanoamérica, Brasil, Portugal y otros países de habla portuguesa, y España, que han estudiado o consultado otras obras nuestras, y como no podía ser menos, nuestro agradecimiento más sincero a profesores y maestros de Latinoamérica y España, quienes han tenido la amabilidad de consultar o seguir esta obra en sus clases y a todos aquellos que nos han dado consejos, sugerencias, propuestas para que escribíramos una obra nueva con un enfoque de multilenguaje. Nuestro reconocimiento más sincero y de nuevo “nuestro agradecimiento eterno”; son el aliento diario que nos permite continuar con esta hermosa tarea que es la comunicación con todos ellos. Y como no podía ser menos, nuestro sincero y eterno agradecimiento a nuestros alumnos y lectores que han confiado en nuestras obras. Simplemente, gracias a todos.

En Carchejo (Sierra Mágina, Andalucía),
en Lupiana (Guadalajara, Castilla La Mancha) y en Ciudad de México
Mayo de 2014

PARTE

I

Resolución de problemas con software





Fundamentos de computadoras y de lenguajes de programación

Contenido

- 1.1 Las computadoras en perspectiva
- 1.2 Las computadoras modernas: una breve taxonomía
- 1.3 Estructura de una computadora
- 1.4 Hardware
- 1.5 Software: conceptos básicos y clasificación
- 1.6 Sistema operativo
- 1.7 El lenguaje de la computadora
- 1.8 Internet y la Web
- 1.9 *Cloud Computing* (computación en la nube)
- 1.10 Movilidad: tecnologías, redes e internet móvil
- 1.11 Geolocalización y realidad aumentada
- 1.12 Internet de las cosas
- 1.13 *Big Data*: los grandes volúmenes de datos
- 1.14 Lenguajes de programación
- 1.15 Evolución de los lenguajes de programación
- 1.16 Paradigmas de programación
 - › Resumen

Introducción

Actualmente numerosas personas utilizan procesadores de texto para escribir documentos, hojas de cálculo, realizar presupuestos, nóminas, aplicaciones estadísticas, navegadores (*browsers*) para navegar y explorar la red internet, y programas de correo electrónico para enviar correos electrónicos (*e-mail*) también por la Red. Los procesadores de texto, las hojas de cálculo, los administradores (gestores) de correo electrónico son ejemplos de programas de software que se ejecutan en la computadora. El software se desarrolla utilizando, fundamentalmente, lenguajes de programación como C, Java, C++ o XML, aunque en el campo profesional, de la Web y de internet, se utilizan con gran profusión lenguajes como PHP, JavaScript, Python o Ruby on Rails.

Internet, que a finales del siglo XX todavía no estaba muy extendido, hoy en día es muy popular, y estudiantes, profesionales y personas de todo tipo y condición utilizan la red para consultar y buscar información, comunicarse entre ellas y como herramientas de trabajo. Asimismo, los estudiantes navegan por internet y utilizan las computadoras para realizar sus trabajos académicos. Todas estas actividades, y muchísimas otras, son posibles gracias a la disponibilidad de diferentes aplicaciones de software conocidas como programas de computadora.

Los lenguajes de programación más idóneos para la enseñanza de la programación son estructurados como C y orientados a objetos como C++ y Java. Por esta razón serán los lenguajes de programación que utilizaremos como referencia en nuestra obra, aunque en algunos casos se harán breves referencias al lenguaje Pascal, el lenguaje clásico de aprendizaje de la programación y C#, lenguaje creado por Microsoft para competir con Java creado por Sun Microsystems, hoy propiedad de Oracle.

El objetivo principal de este libro es enseñarle cómo escribir programas en un lenguaje de programación basándonos en competencias que deberá adquirir el alumno que se introduce en los estudios de programación de sistemas computacionales o ingeniería informática.

Antes de comenzar a enseñarle las técnicas de programación, es muy útil que usted entienda la terminología básica y los diferentes componentes de una computadora, así como los conceptos fundamentales de los lenguajes de programación, de internet y de la Web.

Puede saltar la lectura de aquellos apartados con los que se sienta familiarizado, aunque al menos le sugerimos que revise o recuerde, en su casa, los apartados específicos de lenguajes de programación, software y sistemas de numeración.

Es muy posible que muchos de los términos del capítulo ya los conozca usted, sobre todo si es un “nativo digital” que comienza sus estudios universitarios o profesionales; por esta razón se han recogido los términos más usuales en la vida diaria y, naturalmente, en su campo académico o profesional.

1.1 Las computadoras en perspectiva

Todas las historias publicadas sobre el nacimiento de las computadoras¹ comienzan remontándose a la antigüedad con la creación del ábaco como primera máquina de cálculo. Este instrumento, inventado en Asia, se utilizó en la antigua Babilonia, en China y, por supuesto, en Europa. Ha llegado a nuestros días donde todavía se utiliza con fines educativos y de ocio.

En 1642, el filósofo y matemático francés Blas Pascal inventó la primera calculadora mecánica conocida como **pascalina**. La calculadora tenía una serie de engranajes (ruedas dentadas) que permitían realizar sumas y restas, con un método, entonces ingenioso y revolucionario: cuando se giraban los dientes de la primera rueda, avanzaba un diente la segunda rueda, al girar los dientes de la segunda rueda, avanzaba un diente de la tercera, y así sucesivamente. Tanto el ábaco como la pascalina, solo podían realizar sumas y restas. Asimismo, a finales del siglo XVII, el científico alemán **Gottfried Leibniz**, en 1694, inventó una máquina que podía sumar, restar, multiplicar y dividir. En 1819, el francés **Joseph Jacquard** creó las bases de las tarjetas perforadas como soporte de información.

Generaciones de computadoras

Sin embargo, casi todos los historiadores coinciden en que fue a principios del siglo XIX la era de partida de la computación moderna. El físico y matemático inglés, **Charles Babbage** construyó dos máquinas calculadoras: la máquina de diferencias y la máquina analítica. La primera podía realizar automáticamente operaciones complejas como elevar números al cuadrado. Babbage construyó un prototipo de la máquina diferencial, aunque nunca fue fabricado para producción. Entre 1833 y 1835 diseñó la máquina analítica, una calculadora que incluía un dispositivo de entrada, dispositivo de almacenamiento de memoria, una unidad de control que permitía instrucciones de proceso en secuencias y dispositivos de salida. Esta máquina sentó las bases de la computación moderna, aunque los trabajos de Babbage fueron publicados por su colega **Ada Augusta, condesa de Lovelace**, considerada como la primera programadora de computadoras del mundo. A finales del siglo XIX se utilizaron por primera vez con éxito las tarjetas perforadas que ayudaron a realizar el censo de Estados Unidos en 1890. **Herman Hollerith** inventó una máquina calculadora que funcionaba con electricidad y utilizaba tarjetas perforadas para almacenar datos. La máquina de Hollerith tuvo un gran éxito y apoyándose en ella creó la empresa Tabulating Machine Company, que más tarde se convirtió en la reconocida **IBM**.

¹ En España está muy extendido el término **ordenador** para referirse a la traducción de la palabra inglesa *computer*. El DRAE (*Diccionario de la lengua española*, de la Real Academia Española y todas las Academias de la Lengua de Latinoamérica, África y Asia) acepta, indistintamente, los términos sinónimos: **computador**, **computadora** y **ordenador**. Entre las diferentes acepciones define la computadora electrónica como: “máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos”. En el *Diccionario panhispánico de dudas* (Madrid: RAE, 2005, p. 157), editado también por la Real Academia Española y la Asociación de Academias de la Lengua Española, se señala que el término **computadora** (del término inglés *computer*) se utiliza en la mayoría de los países de América, mientras que el masculino **computador** es de uso mayoritario en Chile y Colombia; en España se usa preferentemente el término **ordenador**, tomado del francés *ordinateur*. En este joven diccionario la definición de computador es “Máquina electrónica capaz de realizar un tratamiento automático de la información y de resolver con gran rapidez problemas matemáticos y lógicos mediante programas informáticos”.

Se considera que Mark I fue la primera computadora digital de la historia que aprovechó el éxito de la máquina de tarjetas perforadas de Hollerith. IBM construyó **Mark I** en 1944 junto con la Universidad de Harvard y con la dirección de **Howard Aiken**. Las entradas y salidas de datos se realizaban mediante tarjetas y cintas perforadas. En 1939, **John V. Atanasoff**, profesor de la Universidad de Iowa y el estudiante de doctorado **Clifford E. Berty**, construyeron un prototipo de la ABC, una computadora digital que utilizaba tubos de vacío (válvulas) y el sistema de numeración digital de base 2, además de que disponía de unidad de memoria. La computadora no se comercializó y el proyecto fue abandonado, pero en 1973, un tribunal federal de Estados Unidos reconoció de modo oficial a Atanasoff los derechos sobre la invención de la computadora digital electrónica automática. Entre 1942 y 1946, **John P. Eckert**, **John W. Mauchly** y su equipo de la Universidad de Pensilvania, construyeron la **ENIAC**, considerada la primera computadora digital de la historia y que fue utilizada desde 1946 hasta 1955 (contenía 18 000 tubos de vacío y pesaba 30 toneladas).

Las computadoras como se les conoce hoy día siguen el **modelo Von Newmann**. El matemático **John von Newmann** realizó un estudio teórico a finales de la década de 1940 en los que sentó las bases de la organización y reglas de funcionamiento de la computadora moderna. Su diseño incluía componentes como unidad lógica y aritmética, unidad de control, unidad de memoria y dispositivos de entrada/salida. Estos componentes se describen en el apartado siguiente. Desde el punto de vista comercial, las computadoras más conocidas y que aparecieron en las décadas de 1950 y 1960 son: Univac I, IBM 650, Honeywell 400 y las populares IBM 360.

El transistor fue inventado en 1948 como sustituto de la válvula o tubo de vacío, y los transistores comenzaron a aparecer en las computadoras ocho años más tarde. Las computadoras que utilizaban transistores fueron radicalmente más pequeñas, más fiables y más económicas que las válvulas de vacío para almacenar y manipular datos. A finales de la década de 1950, los investigadores desarrollaron el **circuito integrado**, un pequeño **chip de silicio** que contenía centenares y miles de transistores y otros dispositivos electrónicos. A mediados de la década de 1960 las computadoras basadas en transistores fueron remplazadas por más pequeñas pero más potentes construidas alrededor de estos nuevos circuitos integrados. Los circuitos integrados remplazaron a los transistores por las mismas razones que estos remplazaron a las válvulas de vacío: *fiabilidad, tamaño, velocidad, eficiencia y costo*. Las invenciones anteriores produjeron un gran impacto en la sociedad, pero fue el desarrollado por Intel, en 1971, del primer microprocesador: un único chip de silicio que contenía todos los componentes de una computadora.

Esta era se destacó también por la aparición de la industria de desarrollo del software con la introducción de los dos lenguajes de programación de alto nivel: **FORTRAN** (1954, aplicaciones científicas) y **COBOL** (1959, aplicaciones de negocios). Estos dos lenguajes de programación de alto nivel fueron remplazando en el desarrollo al lenguaje ensamblador que a su vez sustituyó al lenguaje máquina (basado en ceros y unos) que es el lenguaje con el que funcionan las computadoras.

La revolución de las computadoras personales comenzó a finales de la década de 1970 cuando Apple, Commodore y Tandy, y otras compañías, introdujeron computadoras de bajo costo, basadas en microprocesadores que ocupaban muy poco espacio físico. En 1981, IBM presentó su **computadora personal (PC, Personal Computer)** y junto con Microsoft, creador del sistema operativo MS-DOS que sustentaba a esta computadora, fueron las compañías que iniciaron la nueva era de la computación que todavía hoy vivimos y disfrutamos.

Las computadoras modernas de hoy en día son muy potentes, fiables y fáciles de utilizar. Se han convertido en herramientas indispensables de la vida diaria lo mismo para los niños en las escuelas que para los profesionales en las empresas. Ellas pueden aceptar instrucciones orales (de voz) e imitar el razonamiento humano mediante técnicas de inteligencia artificial. En la actualidad, las computadoras se utilizan para ayudar a los médicos en el diagnóstico de enfermedades, a los empresarios en la toma de decisiones y a los militares a posicionarse geográficamente en el terreno mediante satélites y sistemas GPS. Las computadoras portátiles crecen en números exponenciales y las *netbooks* y tabletas (computadoras de 7 a 11 pulgadas de tamaño de pantalla) y los teléfonos inteligentes (*smartphones*) se están convirtiendo en el elemento de cálculo y de acceso a internet por antonomasia.

Los teléfonos celulares (móviles) y las redes de comunicaciones móviles y fijas, redes inalámbricas, sin cables, están configurando un nuevo modelo de computación, computación móvil o computación celular, que está trayendo infinidad de aplicaciones de computación móvil. Los dispositivos portátiles (*handheld*) permiten conexiones a internet, envíos de correos electrónicos (*e-mail*), conexión a redes sociales, navegar con sistemas GPS de posicionamiento global, visualizar mapas de ciudades, de carreteras, etcétera.

En lo relativo a los lenguajes de programación utilizados por las computadoras hay que destacar que cada computadora procesa instrucciones en un lenguaje nativo denominado **lenguaje máquina**. El len-

guaje máquina utiliza códigos numéricos para representar las operaciones básicas de la computadora: sumas, restas, productos... Este lenguaje era tedioso, aunque imprescindible para el funcionamiento de la máquina y poco a poco fueron apareciendo otros lenguajes, primero el **ensamblador**, que ya utilizaba palabras cortas para identificar a las instrucciones y por último los **lenguajes de alto nivel**, ya más similares a los humanos. Estos lenguajes, a los cuales se dedica fundamentalmente nuestra obra, se siguen utilizando con gran profusión en la actualidad: **C, C++, Java, C#, Objective-C** (recuperado por Apple para sus dispositivos)... o los modernos de la Web, **HTML, XML, JavaScript**... o los innovadores **Ruby on Rail, Python...**

1.2 Las computadoras modernas: una breve taxonomía

Hoy en día la computación se está volviendo cada vez más ubicua de modo que se puede leer un correo electrónico, escuchar una canción, hacer una reservación para el avión o el tren, acceder a las cuentas de su banco o reproducir un video o ver una película, desde cualquier lugar, en cualquier dispositivo y en cualquier momento. Desde el punto de vista técnico aunque existen diferentes categorías de computadoras, por ahora, todas ellas tienen la misma organización, comparten los mismos elementos y siguen el modelo de Von Newmann, aunque la computadora cuántica cada vez se encuentra en un estado más avanzado de desarrollo. Aunque todas ellas comparten los mismos elementos básicos, existen diferentes categorías de computadoras dependiendo del tamaño y potencia de las mismas. La clasificación más usual es:

- **Grandes computadoras (mainframes).**
- **Supercomputadoras.** Grandes computadoras dedicadas especialmente a la investigación y construidas con decenas y cientos de miles de microprocesadores o múltiples núcleos. Son las computadoras utilizadas por excelencia para investigación y desarrollos avanzados en I+D+i.
- **Servidores.** Un servidor es una computadora que proporciona a otras computadoras conectadas en red, el acceso a datos, programas u otros recursos. Existen numerosos tipos de servidores, por ejemplo, los servidores web responden a las peticiones en páginas web, los servidores de bases de datos manejan las consultas en las bases de datos, servidores de impresión proporcionan acceso a impresoras, servidores de correo dan acceso al correo electrónico, etc. Aunque cualquier computadora de escritorio podría actuar como servidor, las computadoras servidor se fabrican expresamente con estas finalidades. Los servidores pueden tener procesadores más rápidos, más memoria y conexiones más veloces que los sistemas de escritorio típicos. Los servidores se pueden agrupar en *clusters* con el objeto de aumentar su potencia de procesamiento.
- **Computadoras personales y estaciones de trabajo.** Una computadora personal (PC) está, generalmente hablando, diseñada para ser utilizada por una persona en un momento dado y es una herramienta de productividad, creatividad o comunicación. Las PC pueden clasificarse en computadoras de escritorio, estaciones de trabajo y *laptops* o portátiles (*notebooks, netbooks, ultrabooks, integradas “all-in-one”*).
- **Computadoras o dispositivos de mano (handheld).** Computadoras o dispositivos que se pueden transportar fácilmente en el bolsillo, en pequeñas bolsas y que sirven para cubrir las necesidades de los usuarios, normalmente, que están en movimiento. Están dotadas de teclados (físicos o virtuales) y pantallas pequeñas (3 a 10 pulgadas). Se clasifican en: teléfonos inteligentes (*smartphones*), tabletas (*tablets*), consolas o videoconsolas, lectores de libros electrónicos (*eReaders*), PDA (los primitivos asistentes digitales, hoy casi desaparecidos en beneficio de los restantes dispositivos móviles en los que se han integrado). Las estadísticas actuales más reputadas realizadas por consultoras, fabricantes, UIT (Unión Internacional de Telecomunicaciones), gobiernos, etc., muestran que los teléfonos inteligentes y tabletas han superado en ventas al conjunto de computadoras personales de escritorio y portátiles (*laptops*).
- **Sistemas embebidos.** Más de 90% de los microprocesadores existentes en el mundo se ocultan o “embeben” (empotran) en dispositivos electrónicos, como sensores y en los más variados aparatos, como automóviles, aviones, trenes, barcos... Un microprocesador utilizado como componente de un sistema físico mayor se llama sistema embebido. Se pueden encontrar este tipo de sistemas, además de en los aparatos citados, en luces de tráfico, juguetes, máquinas de juegos, aparatos de TV, tarjetas inteligentes y sensores más diversos. Los microprocesadores en sistemas embebidos actúan de modo similar a como lo hacen en computadoras personales. Cuando un programa se incrusta en un chip



Figura 1.1 Diferentes tipos de computadoras: a) personales; b) supercomputadoras; c) servidores; d) computadoras de mano (tabletas).

de silicio se suele conocer como *firmware*, un híbrido entre *hardware* y *software*. Los sensores y restantes dispositivos dotados de microprocesadores constituyen el núcleo fundamental del internet de las cosas u objetos (*IoT, Internet of Things*), que es la base hoy día de la generación de los grandes volúmenes de datos existentes en la actualidad y conocidos como *Big Data* y que posteriormente comentaremos.

1.3 Estructura de una computadora

Una computadora es un dispositivo electrónico que almacena y procesa datos, y que es capaz de ejecutar órdenes o comandos, denominadas instrucciones o sentencias. Las computadoras procesan datos bajo el control de un conjunto de instrucciones denominadas programas de computadoras.

Una computadora consta de Unidad Central de Proceso (UCP/CPU), memoria y almacenamiento, dispositivos periféricos (incluyendo algún tipo de conexión de red). Una computadora ejecuta cuatro operaciones que se realizan en un ciclo conocido por **EPSA** (o **IPSO**, *input, processing, output, storage*):

1. Entrada.
2. Proceso (procesamiento).
3. Salida.
4. Almacenamiento.

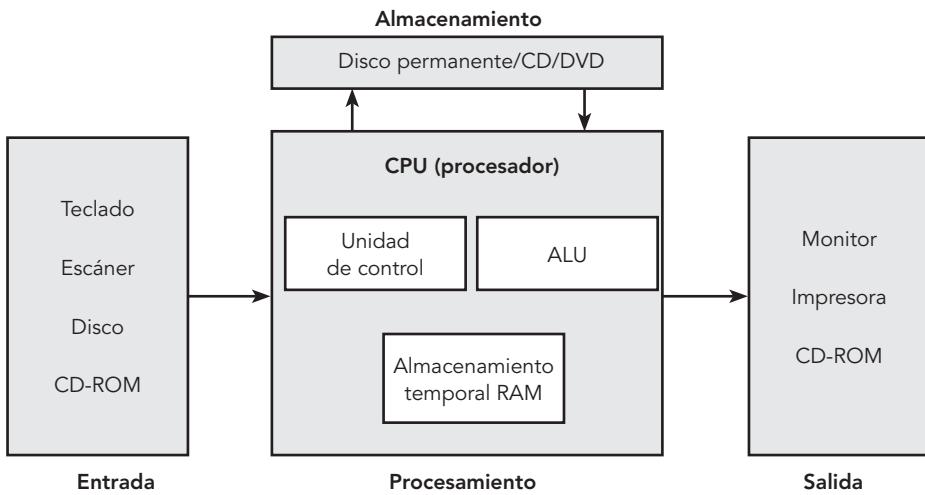


Figura 1.2 Ciclo de computadora EPSA (IPSO).

El usuario introduce los datos e informa a la computadora de las instrucciones a realizar mediante un programa (*Entrada*); se procesan los datos y se convierten en información (*Procesamiento*); se extrae la información de salida de modo que el usuario pueda ver los resultados (*Salida*); y se almacenan los resultados que pueden ser guardados de modo permanente (*Almacenamiento*).

Las computadoras incluyen dos grandes categorías de componentes: *hardware* (componentes físicos) y *software* (componentes lógicos). El *hardware* es el aspecto físico de las computadoras y constan de varios componentes o dispositivos: monitor (pantalla), teclado, ratón (*mouse*), discos duros, memoria, discos CD y DVD, memoria externa (*flash*), cámaras de video, etc. El *software* es el conjunto de instrucciones que controlan el *hardware* y realizan tareas específicas. La programación de computadoras consiste en escribir instrucciones que la computadora ha de ejecutar. Al igual que sucede en numerosas máquinas existentes en la vida diaria, se puede aprender un lenguaje de programación que permitirá escribir los programas sin necesidad de conocer el *hardware* de la computadora, aunque será mejor y más eficiente que el programador (persona que escribe el programa) comprenda bien el efecto de las instrucciones que compone.

Las instrucciones básicas que realiza una computadora son: entrada (lectura o introducción de datos), salida (visualizar o escribir resultados), almacenamiento (guardar datos) y realización de operaciones básicas (aritméticas y lógicas) y complejas. En el mercado actual, y sobre todo en el mercado de la formación, proliferan las computadoras personales de escritorio y las computadoras portátiles (*laptops*, *notebooks* y *netbooks*). Los grandes fabricantes como Hewlett Packard (HP), Dell, Acer, Asus, Oracle (gracias a la compra de Sun Microsystems), Toshiba, Samsung, Sony, LG, etc., ofrecen una extensa gama de equipos y de costos asequibles² que facilitan el aprendizaje tanto a las generaciones más jóvenes como a los profesionales.

1.4 Hardware

Los componentes (unidades) más importantes del *hardware* de una computadora son: unidad central de proceso o procesador (CPU, *Central Processing Unit*), memoria central o principal del equipo (memoria RAM); dispositivos o periféricos de entrada y salida (monitor, impresora, teclado, *mouse* [ratón], video-cámara); dispositivos de almacenamiento secundario (discos, memorias *flash*, memorias SD) y dispositivos de comunicaciones. Los diferentes componentes se conectan a través de un subsistema denominado *bus* que transfiere datos entre componentes. En la figura 1.3 se muestran los componentes principales de una computadora.

² A título meramente comparativo resaltar que la primera PC que tuvo el autor de esta obra, comprado en la segunda mitad de la década de los 80, costó unas 500 000-600 000 pesetas (\$5 000 aprox.) y solo contemplaba una unidad central de 512 KB, disco duro de 10 MB y una impresora de matriz.



Figura 1.3 Componentes principales de una computadora.

La figura 1.4 muestra la estructura global de una computadora considerando los buses de comunicación. Los componentes principales de la computadora: la CPU (procesador), memoria, entrada/salida (I/O) unidos por el bus del sistema. La CPU es el dispositivo que no solo ejecuta las instrucciones del programa, sino también controla (comanda) los diferentes componentes de la computadora. La memoria almacena el (los) programa(s) a ejecutar y los datos que cada programa utiliza. El componente E/S incluye todos los dispositivos periféricos (entrada, salida, almacenamiento, redes) y de almacenamiento donde se almacenan de modo permanente los datos y programas (unidades de disco, cintas, unidades USB “pendrives”...). El bus es el dispositivo que permite mover la información de un dispositivo a otro.

Unidad central de proceso (CPU)

La unidad central de proceso es el “cerebro” y la parte más importante y cara del hardware de la computadora. La CPU ejecuta las operaciones aritméticas (como suma, resta, multiplicación y división) y lógicas en la unidad aritmética y lógica (ALU, *Arithmetic and Logic Unit*) y enlaza con la memoria principal y los dispositivos de entrada y salida, así como los dispositivos de almacenamiento externo o secundario.

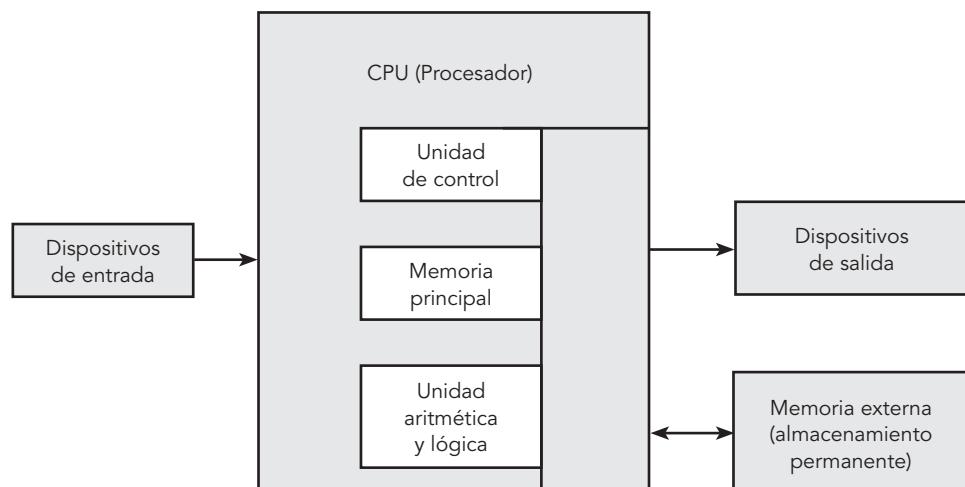


Figura 1.4 Estructura global de una computadora.

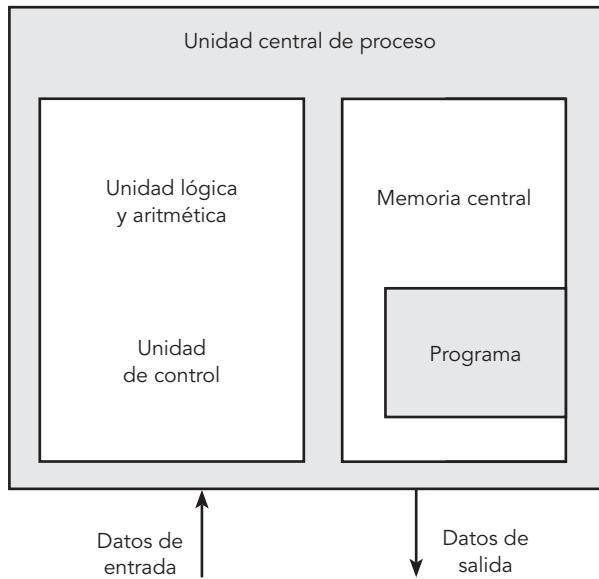


Figura 1.5 Unidad central de proceso.

Normalmente, una computadora personal (*PC*), una tableta (*tablet*) o un teléfono inteligente, tienen una sola CPU o procesador. Sin embargo, cada día comienza a ser más frecuente que al igual que sucede con las supercomputadoras (que tienen numerosos procesadores), las computadoras de escritorio, portátiles, teléfonos inteligentes y tabletas, incluyan múltiples procesadores, lo que permite la realización de muchas operaciones simultáneamente. En la figura 1.5 se muestra la organización de la unidad central de proceso.

Un procesador multinúcleo (*multicore*) incluye múltiples procesadores en un único chip de circuito integrado. En la actualidad, tanto en las PC de escritorio, como *laptops*, teléfonos inteligentes y tabletas, ya comienza a ser usual la comercialización de computadoras con doble núcleo (*dual core*) y de cuatro núcleos (*quad core*).

Intel comercializa para toda su gama de PC y computadoras portátiles (*laptops*) los procesadores Intel Core cuyos modelos más populares son: i3 de dos núcleos, i5 de cuatro núcleos e i7 con dos modelos a su vez de 4 y 6 núcleos.

En dispositivos móviles, la arquitectura ARM es la más implantada en teléfonos inteligentes y tabletas, y los fabricantes con mayor penetración son: Samsung, NVIDIA y Qualcomm, y Apple (que fabrica sus propios procesadores para iPhone y iPad). Todos estos fabricantes comercializan procesadores de dos y cuatro núcleos para sus dispositivos móviles. Así Samsung comercializa el modelo Galaxy SIII con un procesador de 4 núcleos y el Galaxy S IV presentado en 2013 trae de serie un procesador Exynos 5 Octa de 8 núcleos.

Procesadores multinúcleo de Intel para PC/Laptops

- i3, dos núcleos
- i5, cuatro núcleos
- i7, cuatro y seis núcleos

Procesadores para dispositivos móviles

- ARM:** uno, dos y cuatro núcleos
- Samsung Exynos:** de 1, 2, 4 y 8 núcleos
- Apple:** uno, dos y cuatro núcleos

Memoria de la computadora

El trabajo principal de la CPU es seguir las instrucciones codificadas de los programas, pero solo puede manejar una instrucción (aunque ya existen hoy día como veremos más adelante las computadoras multiprocesadoras que pueden efectuar varios procesos simultáneamente) y unos pocos datos en cada momento. La computadora necesita un espacio para almacenar el resto del programa y los datos hasta que el procesador está listo para su uso. Esta es la tarea que realiza la memoria RAM.

La **memoria central RAM** (*Random Access Memory*) o simplemente **memoria** es el tipo más importante de almacenamiento principal de la computadora. Los chips de RAM contienen circuitos que almacenan las instrucciones de los programas, los datos de entrada y los resultados temporalmente.

En la mayoría de las computadoras existen dos tipos de memoria principal: **memoria de acceso aleatorio, RAM**, que soporta almacenamiento temporal de programas y datos y **memoria de solo lectura, ROM**, que almacena datos o programas de modo permanente.

En general, la información almacenada en la memoria puede ser de dos tipos: las *instrucciones* de un programa y los *datos* con los que operan las instrucciones. Para que un programa se pueda *ejecutar* (correr, rodar, funcionar..., en inglés *run*), debe estar 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.

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. Esta memoria RAM puede ser **estática (SRAM)** o **dinámica (DRAM)** según sea el proceso de fabricación. Las memorias RAM actuales más utilizadas son: **SDRAM** en sus tres tipos: **DDR** (*Double Data Rate*), **DDR2** y **DDR3**.

¿Sabía que..?

Ejecución

Cuando un programa se ejecuta (realiza, funciona) en una computadora, se dice que se ejecuta.³

¿Sabía que..?

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.

La memoria ROM es una memoria que almacena información de modo permanente en la que no se puede escribir (viene pregrabada, “grabada,” por el fabricante) ya que es una memoria de solo lectura. Los programas almacenados en ROM no se pierden al apagar la computadora y cuando se enciende (se prende), se lee la información almacenada en esta memoria. Al ser esta memoria de solo 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.

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.

Los programas y los datos se almacenan en RAM. Las memorias de una computadora personal se miden en unidades de memoria (se describen en el siguiente apartado) y suelen ser actualmente de 1 a 8 GB (gigabyte), aunque ya es frecuente encontrar memorias centrales de 16 y 32 GB en computadoras personales y en cantidad mayor en computadoras profesionales y en servidores.

Normalmente una computadora contiene mucha más memoria RAM que memoria ROM interna; también la cantidad de memoria se puede aumentar hasta un máximo especificado, mientras que la cantidad de memoria ROM normalmente es fija. Cuando en la jerga informática y en este texto se menciona la palabra memoria se suele referir a memoria RAM que normalmente es la memoria accesible al programador.

La memoria RAM es una memoria muy rápida y limitada en tamaño, sin embargo la computadora tiene otro tipo de memoria denominada memoria secundaria o almacenamiento secundario que puede

³ En la jerga informática también se conoce esta operación como “correr un programa”.

crecer comparativamente en términos mucho mayor. La memoria secundaria es realmente un dispositivo de almacenamiento masivo de información y por ello, a veces se le conoce como memoria auxiliar, almacenamiento auxiliar, almacenamiento externo y memoria externa.

Unidades de medida de memoria

La unidad básica de almacenamiento es el *byte*. Cada byte contiene una dirección única. La dirección se utiliza para localizar el byte de almacenamiento y recuperación de datos. A los bytes se puede acceder en cualquier orden, de modo aleatorio, y por esta razón la memoria principal se denomina RAM. Los múltiplos de la unidad de memoria básica son 1 **kilobyte (K o Kb)** equivalente a 1.024 bytes,⁴ aunque desde el punto de vista práctico 1 Kb se suele considerar equivalente a 1.000 bytes; 1 **megabyte (M o Mb)** equivale a 1.024×1.024 bytes o de modo práctico, 1 millón de bytes; 1 **gigabyte (G o Gb)** equivale a $1.024 \times 1.024 \times 1.024$ bytes, equivalente a 1.000 millones de bytes; 1 **terabyte (T o Tb)**, equivalente a 1.024 gigabytes; 1 **petabyte (P o Pb)** equivale a 1.024 terabytes y 1 **exabyte (E o Eb)**, equivalente a 1.024 petabytes. La tabla 1.1 muestra las unidades básicas de medida de almacenamiento en memoria.

En la actualidad se vive en la *era del exabyte* de modo que el usuario ya está acostumbrado a los terabytes que almacenan sus discos duros de sus PC o incluso de sus laptops y las empresas ya almacenan en sus bases de datos cantidades de información del orden de petabytes y exabytes.

La mayoría de los equipos actuales tienen al menos 4 a 8 Gb de memoria RAM (en el caso de las laptops y PC de escritorio), aunque ya es muy frecuente encontrar computadoras con 10, 12 y 16 Gb e incluso con memoria superior. La computadora requiere la mayor capacidad de memoria para permitir que el equipo ejecute el mayor número de programas del modo más rápido posible. En caso de necesidad de ampliación de memoria se puede incorporar memoria adicional instalando chips o tarjetas de memoria en las ranuras libres que suelen incluir las placas base del sistema. Si la computadora no tiene suficiente memoria, algunos programas se ejecutarán muy lentamente y otros no lo harán de ningún modo.

Tabla 1.1 Unidades de medida de almacenamiento.

Unidad	Símbolo	Tamaño en bits/bytes
Byte	B	8 bits
Kilobyte	KB	2^{10} bytes = 1.024 bytes
Megabyte	MB	2^{20} bytes = 1 048 576 bytes; 1.024 Kb = 2^{10} Kb
Gigabyte	GB	2^{30} bytes = 1 073 741 824 bytes; 1.024 Mb = 2^{10} Mb = 2^{20} Kb
Terabyte	TB	2^{40} bytes = 1 099 511 627 776 bytes 1 TB = 1.024 GB = 2^{10} GB = 2^{20} Kb
Petabyte	PB	2^{50} bytes = 1 125 899 906 842 624 bytes 1 PB = 1.024 TB = 2^{10}
Exabyte	EB	2^{60} bytes; 1.024 Pb = 2^{10} Pb = 2^{20} TB
Zettabyte	ZB	2^{70} bytes; 1.024 Eb
Yotabyte	YB	2^{80} bytes; 1.024 Zb

Bit: un dígito binario 0 o 1.

Byte: una secuencia de ocho bits.

⁴ Se adoptó el término **kilo** en computadoras debido a que 1 024 es muy próximo a 1 000, y por eso en términos familiares y para que los cálculos se puedan hacer fáciles mentalmente se asocia 1 KB a 1 000 bytes y 1 MB a 1 000 000 de bytes y 1 GB a 1 000 000 000 de bytes. Así, cuando se habla en jerga diaria de 5 KB estamos hablando, en rigor, de $5 \times 1.024 = 5 120$ bytes, pero en cálculos consideramos 5 000 bytes. De este modo se guarda correspondencia con las restantes representaciones de las palabras *kilo*, *mega*, *giga*... Usted debe considerar siempre los valores reales para 1 KB, 1 MB o 1 GB, mientras esté en su fase de formación y posteriormente en el campo profesional desde el punto de vista de programación, para evitar errores técnicos en el diseño de sus programas, y solo recurrir a las cifras mil, millón, etc., para la jerga diaria.

Caso práctico (un ejemplo comercial)

Samsung comercializaba en el verano de 2013, en España, una *laptop*, al precio de 575€ con las siguientes características técnicas: Procesador Intel Core i5, 6 GB de memoria RAM DDR3, un disco duro de 500 GB (0.5 terabytes), una tarjeta gráfica NVIDIA de 1 GB, DDR3 y los dispositivos siguientes: pantalla LED HD de 15.6", red inalámbrica con protocolos abg/n, Bluetooth v4.0, 3 USB 3.0, lector de tarjetas, etcétera.

Dispositivos de entrada y salida

Para realizar tareas útiles en las computadoras se requiere “tomar” o “capturar” datos y programas, y visualizar los resultados de la manipulación de datos. Los dispositivos (periféricos) que envían (alimentan) datos y programas a las computadoras se denominan dispositivos o periféricos de entrada. Se obtiene información (datos y programas) desde los dispositivos de entrada y ponen esta información a disposición de otras unidades de procesamiento. Los periféricos de entrada más usuales son el teclado y el *mouse* (ratón)⁵ y unidades de almacenamiento secundario o escáner para digitalizar información. La información se puede introducir también mediante micrófonos, cámaras de video que permiten “subir” fotografías y videos, o recibir información de redes de comunicaciones, especialmente la red de internet, o dispositivos de escaneado de texto o imágenes.

Los dispositivos que utiliza la computadora para visualizar y almacenar resultados se denominan **periféricos de salida**. La mayoría de la información que sale de la computadora se visualiza en pantallas (monitores), se imprime en papel utilizando impresoras o se almacena en dispositivos de almacenamiento secundario. De igual forma, las computadoras pueden enviar su información a redes como internet.

En la actualidad son muy populares los periféricos multimedia que se utilizan en tareas multimedia y pueden ser de entrada o de salida: altavoces (suelen estar incorporados en los portátiles, pero también pueden conectarse externamente a través de puertos USB); micrófono (permiten capturar sonidos); webcam (cámara digital web incorporada o conectada externamente también mediante puertos USB; auriculares (permite escuchar sonidos) e incluso auriculares con micrófono incorporado.

Una computadora, normalmente, viene dotada de una gran variedad de puertos de comunicación para cumplir necesidades diversas y atender las necesidades de los periféricos citados anteriormente. Así es usual que integren los siguientes puertos (*ports*) de entrada/salida: uno o más puertos de video para conexión de monitores; jacks (conectores) de audio para conexión de altavoces y(o) micrófonos; puertos USB para conexión de teclados, dispositivos apuntadores (como un *mouse* o ratón), impresoras, cámaras, unidades de disco (duro, discos SSD, memorias *flash*...), dispositivos portátiles de almacenamiento (*pen-drives* o “lápices”, memorias SIM...), etc. Otros puertos permiten conexión a tarjetas de expansión, como ampliación de memoria, conexión a aparatos de TV, video, audio, etcétera.

Dispositivos de almacenamiento secundario

Los programas y los datos se deben recopilar en la memoria principal antes de su procesamiento y como esta memoria es volátil toda la información almacenada en ella, antes de su procesamiento, se pierde cuando se apaga la computadora. La información reunida en ella se debe transferir a otros dispositivos de almacenamiento donde se pueda guardar de modo permanente y por largos períodos (con la excepción de que se inutilice o se rompa por alguna causa física o se cambie intencionadamente la información que acumula).

Los datos y los programas se depositan de modo permanente en dispositivos de almacenamiento y se moverán a la memoria cuando la computadora los necesite. Un dispositivo que recopila la información de modo permanente y por largos períodos se denomina almacenamiento secundario o externo. Para poder realizar la transferencia de información desde la memoria principal al almacenamiento

⁵ Todas las acciones a realizar por el usuario se harán con el ratón con la excepción de las que requieren de la escritura de datos por teclado. El nombre de ratón parece que proviene de la similitud del cable de conexión con la cola de un ratón. Hoy día, sin embargo, este razonamiento carece de sentido ya que existen ratones inalámbricos que no usan cable y se comunican entre sí a través de rayos infrarrojos.

secundario o viceversa, estos dispositivos deben ser conectados directamente a la computadora o ellos mismos entre sí. Existen diferentes tipos de dispositivos o periféricos de almacenamiento:

- Unidades de disco (discos duros internos y externos).
- Unidades de discos compactos (CD, DVD, Blu-ray).
- Unidades de cinta (ya en desuso).
- Unidades de memoria flash, USB (*pendrives* o lápices) y tarjetas de memoria (SD).
- Unidades de memoria SD (tarjetas de cámaras fotográficas, videocámaras).
- Unidades de disco duro SSD (memorias de estado sólido).
- Unidades de memoria ZIP.

Todos los soportes de almacenamiento funcionan sobre unidades lectoras de CD, DVD y discos duros.

Dispositivos de comunicación

Las computadoras se conectan entre sí por medio de redes informáticas y dispositivos de comunicación. La conexión física a una red se puede realizar mediante “*conexión cableada*” o aparecen las redes cableadas y las redes inalámbricas (*wireless*). Aunque en las empresas existen todavía redes corporativas LAN e Intranet, hoy día lo más usual es que tanto las corporativas como las domésticas se conecten entre sí a través de la red internet.

Los dispositivos de comunicación más utilizados son: módem y tarjeta de interfaz de red (NIC, *network interface card*). En el caso de las redes inalámbricas se requiere un *router* inalámbrico que permite configurar las redes como si fuesen cableadas.

La conexión a internet de su PC requerirá la contratación de un proveedor de servicios de internet (ISP, *Internet Service Provider*) que facilitará la conexión desde el hogar o desde la empresa a internet. Las conexiones a internet se realizarán a través de tecnología ADSL (que utiliza la línea telefónica tradicional), cable de fibra óptica, por satélite o mediante tecnologías inalámbricas WiFi o WiMax. Hoy día comienza a ser muy usual encontrar desplegadas redes WiFi en ciudades, aeropuertos, campus de universidades, etcétera.

El otro modelo de conexión a internet se realiza mediante las redes móviles (celulares) y los dispositivos móviles (teléfonos inteligentes y tabletas). Las redes móviles actuales como se verá posteriormente vienen soportadas por redes 3G y cada día, más frecuentes, las redes 4G que alcanzan velocidades de conexión superiores en muchos casos a las redes domésticas ADSL e incluso de fibra óptica.

1.5 Software: conceptos básicos y clasificación

Los programas, conocidos como software, son instrucciones a la computadora. Sin programas, una computadora es una máquina vacía. Las computadoras no entienden los lenguajes humanos, de modo que se necesita utilizar lenguajes de computadoras para comunicarse con ellas.

El software de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan los componentes hardware de una computadora y controlan las operaciones de un sistema informático. El auge de las computadoras en el siglo pasado y en el actual siglo XXI, se debe esencialmente al desarrollo de sucesivas generaciones de software potentes y cada vez más amistosas (“fáciles de utilizar”).

Las operaciones que debe realizar el hardware son especificadas por una lista de instrucciones, llamadas programas, o software. Un programa de software es un conjunto de sentencias o instrucciones que se da a la computadora. El proceso de escritura o codificación de un programa se denomina programación y las personas que se especializan en esta actividad se denominan programadores. Existen dos tipos importantes de software: de sistema y de aplicaciones. Cada tipo realiza una función diferente. Los dos tipos de software están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente de la computadora.

El *software de sistema* es un conjunto generalizado de programas que gestiona los recursos de la computadora, como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Los programadores que escriben software del sistema se llaman *programadores de sistemas*. El *software de aplicaciones* es el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que

instruyen a la computadora para que ejecute una tarea específica. Los programadores que escriben software de aplicaciones se llaman *programadores de aplicaciones*.

Los dos tipos de software están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente de la computadora. En la figura 1.6 se muestra una vista organizacional de una computadora donde se ven los diferentes tipos de software a modo de capas de la computadora desde su interior (el hardware) hasta su exterior (usuario). Las diferentes capas funcionan gracias a las instrucciones específicas (instrucciones máquina) que forman parte del software de sistema y llegan al software de aplicaciones, desarrollado por los programadores de aplicaciones, que es utilizado por el usuario que no requiere ser un especialista.

Software de sistema

El software de sistema coordina las diferentes partes de un sistema de computadora y conecta e interacciona entre el software de aplicación y el hardware de la computadora. Otro tipo de software de sistema que gestiona, controla las actividades de la computadora y realiza tareas de proceso comunes, se denomina *utility* o **utilidades** (en algunas partes de Latinoamérica se conoce como **utilerías**). El software de sistema que gestiona y controla las actividades de la computadora se denomina *sistema operativo*. Otro software de sistema son los *programas traductores* o de traducción de lenguajes de computadora que convierten los lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras.

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

A lo largo de este capítulo y del siguiente, se describirán los conceptos fundamentales de los programas de sistemas más utilizados y sobre todo aquellos que afectarán al lector en el aprendizaje de la programación, como sistemas operativos, editores de texto y traductores de lenguajes (compiladores e intérpretes), así como los entornos integrados de desarrollo, herramienta más frecuentemente empleada por los programadores para el desarrollo y ejecución de los programas en los diferentes lenguajes de programación.

Software de aplicaciones

El software de aplicación tiene como función principal asistir y ayudar a un usuario de una computadora para ejecutar tareas específicas. Los programas de aplicación se pueden desarrollar con diferentes lenguajes y herramientas de software. Por ejemplo, una aplicación de procesamiento de textos (*word processing*) como Word o Word Perfect que ayuda a crear documentos, una hoja de cálculo como Lotus 123 o Excel que ayudan a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísti-

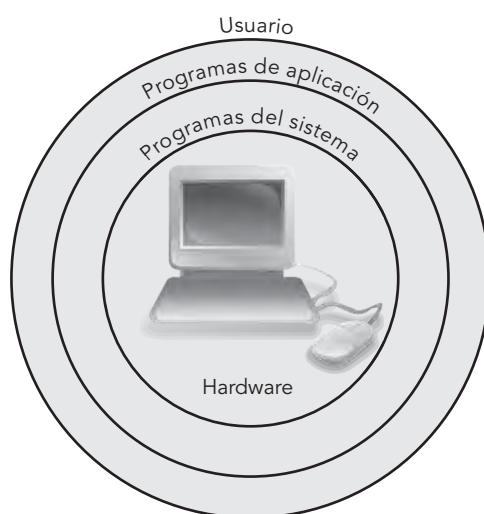


Figura 1.6 Relación entre programas de aplicación y programas de sistema.

cos, a generar diagramas o gráficos, presentaciones visuales como PowerPoint, o a crear bases de datos como Access u Oracle que ayudan a crear archivos y registros de datos.

Los usuarios, normalmente, compran el software de aplicaciones en discos CD o DVD (antiguamente en disquetes) o los descargan (bajan) de la red (internet) e instalan el software copiando los programas correspondientes de los discos en el disco duro de la computadora. Cuando compre estos programas asegúrese de que son compatibles con su computadora y con su sistema operativo. Existe una gran diversidad de programas de aplicación para todo tipo de actividades tanto de modo personal, como de negocios, navegación y manipulación en internet, gráficos y presentaciones visuales, etcétera.

Los lenguajes de programación sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados traductores (compiladores o intérpretes) convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1, bits) que esta pueda entender.

Los programas de utilidad 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 processor*).

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etcétera, 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. La ejecución de un programa requiere una aplicación de una entrada (datos) al programa y la obtención de una salida (resultados). La entrada puede tener una variedad de formas, como números o caracteres alfabéticos. La salida puede también tener formas, como datos numéricos o caracteres, señales para controlar equipos o robots, etcétera.

1.6 Sistema operativo

Un sistema operativo o SO (*OS, Operating System*) es tal vez la parte más importante del software de sistema y es el software que controla y gestiona los recursos de la computadora. En la práctica el sistema operativo es la colección de programas de computadora que controla la interacción del usuario y el hardware de la computadora. El sistema operativo es el administrador principal de la computadora, y por ello a veces se le compara con el director de una orquesta ya que este software es el responsable de dirigir todas las operaciones de la computadora y gestionar todos sus recursos. El núcleo (*kernel*) de un sistema operativo es el software que contiene los componentes fundamentales (*core*) de dicho sistema operativo.

El sistema operativo asigna recursos, planifica el uso de recursos y tareas de la computadora, y monitorea las actividades del sistema informático. Estos recursos incluyen memoria, dispositivos de E/S (entrada/salida), y la UCP o CPU (unidad central de proceso). El sistema operativo proporciona servicios como asignar memoria a un programa y manipulación del control de los dispositivos de E/S como el monitor, el teclado o las unidades de disco. La tabla 1.2 muestra algunos de los sistemas operativos más populares utilizados en enseñanza y en informática profesional. Cuando un usuario interactúa con una computadora, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario, *IGU (Graphical User Interface, GUI)* que hace uso masivo de íconos, botones, barras y cuadros de diálogo para realizar tareas que controla el teclado o el ratón (*mouse*), entre otros dispositivos.

El sistema operativo, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas, 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), discos y otros dispositivos. 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 proporcionan servicios que permiten que cada aplicación se ejecute de modo seguro, eficiente y en concurrencia o en paralelo con otras aplicaciones.

La evolución de los sistemas operativos ha sido considerable desde el nacimiento de las computadoras y han sido numerosos los que han tenido repercusión en las empresas, los negocios, la educación, la



Figura 1.7 Sistemas operativos: Windows 8.1 (izquierda) y Mac OS X Mavericks.

investigación, etc. Los sistemas operativos de mayor impacto han sido desde los primitivos sistemas operativos apoyados por los grandes fabricantes de computación como IBM, Hewlett Packard, Digital, Sun Microsystems, etc., hasta los sistemas operativos que nacieron a la vez que las computadoras personales, PC, como CP/M (muy popular entre las primeras computadoras personales de 8 bits), MS/DOS (diseñado por Microsoft para las computadoras personales PC), UNIX (el primer sistema operativo abierto) que poco a poco ha ido cediendo su presencia en favor de Linux, el sistema operativo abierto por excelencia en la actualidad y con seguridad en el futuro.

Los sistemas operativos más populares hoy día para servidores y computadoras personales (escritorio y portátiles o laptops) son: **Linux**, el sistema operativo de código abierto, **Windows** de Microsoft (en sus versiones 7, XP, Vista y 8); y **Mac OS** para computadoras Apple.

En los últimos años han proliferado los sistemas operativos para teléfonos celulares o móviles conocidos como teléfonos inteligentes (smartphones) que tienen la característica de ejecutarse y conectarse a internet, de modo inmediato tan pronto se “prendan” (“se encienden”), y se les denomina sistemas operativos móviles. Existen diferentes sistemas operativos asociados a los fabricantes de teléfonos inteligentes. Entre ellos destacan **Android** de Google, **iOS** de Apple y menos extendidos, **Windows Phone** de Microsoft, **BBM** de Blackberry, **Symbian** de Nokia, **Web Os** (creado y diseñado por Palm, adquirido en agosto de 2011 por Hewlett Packard y posteriormente en 2013 vendido a LG que parece tiene previsto instalar en sus televisiones inteligentes, SmartTV).

A lo largo de 2013 se inició la comercialización de sistemas operativos móviles de código abierto que se comenzaron a comercializar por diferentes fabricantes de teléfonos móviles. Los sistemas operativos presentados en 2013 y que ya han alcanzado buena penetración en el mercado son:

- Firefox OS de la Fundación Mozilla (apoyado por Telefónica).
- Tizen, apoyado por la Fundación Linux, Intel y Samsung.
- Ubuntu de la empresa Canonical.

Los sistemas operativos en función del número de usuarios se clasifican en: *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios) y atendiendo al número de las tareas (procesos) que pueden realizar simultáneamente, o hilos de proceso:

- Monotarea (una sola tarea).
- Por lotes (*batch*).
- Multitarea (múltiples tareas).
- Multiprogramación.
- Multiproceso.
- Multihilo.

Desde el punto de vista práctico con enfoque al usuario y a la instalación del sistema existen dos grandes categorías: sistema operativo para usuario y sistema operativo para servidores (las computadoras que gestionan y controlan las redes de computadoras de las organizaciones y empresas). Los sistemas operativos con mayor penetración en los servidores de las redes de computadoras son: Linux, Windows Server, Mac OS, etc. Los servidores son una parte fundamental en los centros de procesos de datos y los

Tabla 1.2 Sistemas operativos más utilizados en educación, consumo y en la empresa.

Sistema operativo	Características
Windows 8, Windows 7	Últimas versiones del sistema operativo Windows de Microsoft, que viene a sustituir a Windows Vista, de poco éxito. Ha tenido una gran repercusión e impacto tanto en computadoras grandes, como en personales, portátiles (<i>laptops, notebooks, netbooks</i> , etc.). La versión 8.0 se presentó en octubre de 2012 y la última versión 8.1 en junio de 2013.
Linux	Sistema operativo de software abierto, gratuito y de libre distribución, similar a UNIX, y una gran alternativa a Windows. Muy utilizado actualmente en servidores de aplicaciones para internet.
Mac OS	Sistema operativo de las computadoras Apple Macintosh. Su versión más popular es Mac OS X.
iPhone OS (iOS)	Sistema operativo utilizado por los dispositivos de Apple (iPad, iPod, iPhone). En la actualidad, la última versión es la 7.0 y 7.1, se espera la versión 8.0 a fin de 2014.
Android	Sistema operativo abierto creado por Google y de gran aceptación en teléfonos inteligentes (<i>smartphones</i>) y tabletas (<i>tablets</i>). Su última versión es la 4.4 (Icekit).
Blackberry OS (BBM)	Sistema operativo para teléfonos móviles de los populares teléfonos inteligentes Blackberry, muy utilizados en el mundo de empresa y de negocios. Su última versión es la 10.0.
Windows Phone 8	Versión de Windows 8 para teléfonos inteligentes y que Microsoft comercializa desde octubre de 2012.
Sistemas operativos de código abierto para dispositivos móviles	Tizen (Samsung), Firefox OS (Telefónica), Ubuntu OS, Salifish OS.

programadores e ingenieros de sistemas deberán conocer las características y funcionamiento de los numerosos tipos de servidores existentes como: aplicaciones, audio/video, bases de datos, fax, ftp, correo electrónico, impresión, chat, etc., y los dos grandes tipos de servidores desde el punto de vista práctico son los servidores web y los servidores proxy.

1.7 El lenguaje de la computadora

Los humanos nos entendemos con los lenguajes naturales como el español, el inglés o el portugués, pero ¿y las computadoras cómo se entienden? Cuando pulsamos la letra B en el teclado se visualiza la letra B en la pantalla del monitor. ¿Cuál es la información que se ha almacenado en la memoria central de la computadora? En resumen, ¿cuál es el lenguaje que entienden las computadoras y cómo se almacena cualquier otra información que se introduce en el teclado o se lee de una memoria USB?

Recordemos que una computadora es un dispositivo electrónico que procesa información de modo automático. Las señales eléctricas se mueven entre los diferentes componentes de una computadora. Existen dos tipos de señales eléctricas: analógica y digital. Las señales analógicas se utilizan para representar cosas como sonido; las cintas de audio, por ejemplo, almacenan los datos en señales analógicas. Las computadoras funcionan con señales digitales, las cuales representan la información como una secuencia de ceros y unos. Un 0 representa una señal de bajo voltaje, y un 1, una señal de alto voltaje. Un tema vital en el proceso de funcionamiento de una computadora es estudiar la forma de representación de la información en dicha computadora. Es necesario considerar cómo se puede codificar la informa-

ción en patrones de bits que sean fácilmente almacenables y procesables por los elementos internos de la computadora.

Las señales digitales se procesan en el interior de las computadoras y por esta razón, el lenguaje de una computadora, llamado lenguaje máquina, es una secuencia de dígitos 0 y 1. El dígito 0 o 1 se llama dígito binario o bit. Normalmente una secuencia de ceros (0) y unos (1) se conoce como código binario o número binario.

¿Sabía que..?

El sistema de numeración que se utiliza en la vida diaria se denomina sistema o código decimal o de base 10. En una computadora los caracteres se representan como secuencias de dígitos 0 y 1, es decir, números binarios; el sistema de numeración que utiliza una computadora se llama sistema o código binario o de base 2.

Representación de la información en las computadoras (códigos de caracteres)

Cuando se pulsa un carácter: una letra, un número o un símbolo especial (como &, %) en su teclado, se almacena en la memoria principal de una computadora una secuencia de bits que es diferente en cada caso. El código o esquema más utilizado es el código ASCII (*American Standard Code for Information Interchange*). Este código (apéndice B) consta de 128 caracteres ($2^7 = 128$, 0 a 127), de modo que la posición del primer carácter es 0, la posición del segundo carácter es 1 y así sucesivamente. Así, por ejemplo, el carácter A es el 66º carácter y de posición 65; el carácter B es el 67º carácter y de posición 66, etc. En el interior de las computadoras, cada carácter se representa por una secuencia de ocho bits, es decir un byte. La representación binaria del carácter A es 01000001 y el carácter 4 es 00110100.

Otro conjunto de caracteres, ya hoy muy popular, es el código Unicode que consta de 65 536 caracteres, en lugar de los 128 caracteres del código ASCII básico. Java utiliza el código Unicode en el que cada carácter se representa por 2 bytes, 16 bits, en lugar de 1 byte en el código ASCII. La gran ventaja de Unicode es que permite representar caracteres de numerosos idiomas internacionales como el chino, indio, ruso, etcétera.

Los lenguajes de programación ensambladores y de alto nivel

La mayoría de los lenguajes básicos de computadora, como el **lenguaje máquina** (lenguaje que entiende la máquina directamente), proporcionan las instrucciones a la computadora en bits. Aunque la mayoría de las computadoras utilizan los mismos tipos de operaciones, los diseñadores de los diferentes procesa-

Tabla 1.3 Códigos de representación de la información.

Carácter	Código ASCII	Código binario
A	65	01000001
B	66	01000010
C	67	01000011
—	—	—
1	49	00110001
2	50	00110010
3	51	00110011
—	—	—

dores (CPU) normalmente eligen conjuntos distintos de códigos binarios para realizar dichas operaciones. Por consiguiente, el lenguaje máquina de una computadora no es necesariamente el mismo que el lenguaje máquina de otra computadora. La única coincidencia entre computadoras es que en cualquiera de ellas, todos los datos se almacenan y manipulan en código binario.

Las primeras computadoras eran programadas en lenguaje máquina (código binario). La programación escrita que utiliza lenguaje máquina es un proceso tedioso y difícil de realizar. Por ejemplo, para sumar dos números se puede requerir una instrucción como esta:

```
1101101010101001
```

Para realizar cualquier operación, como la anterior, el programador tiene que recordar las posiciones de los datos en memoria. Esta necesidad de recordar los códigos específicos en binario hacen muy difícil la programación y muy propensa a errores.

Por estas razones comenzaron a crearse lenguajes ensambladores, para hacer la tarea del programador más fácil. En lenguaje ensamblador, una instrucción tiene un formato más fácil de recordar, llamado nemotécnico o nemónico. Las instrucciones en lenguaje ensamblador son abreviaturas o segmentos de palabras clásicas en inglés. Por ejemplo, algunas instrucciones se recogen en la tabla 1.4.

Tabla 1.4 / Ejemplos de instrucciones en lenguaje ensamblador.

Lenguaje ensamblador	Lenguaje máquina	Significado
Add	00100100	Suma
Sub	00100010	resta
Mult	00100110	Multiplicar
Sto	00100010	Almacenar

¿Sabía que..?

Ensamblador

Un programa que traduce un programa escrito en lenguaje ensamblador en un programa equivalente en lenguaje máquina.

Aunque es mucho más fácil escribir instrucciones en lenguaje ensamblador, una computadora no puede ejecutar instrucciones directamente en este lenguaje. Las instrucciones deben traducirse primero a lenguaje máquina. Un programa denominado ensamblador traduce el lenguaje ensamblador (*assembly language*) a instrucciones en lenguaje máquina.

El lenguaje ensamblador hizo la programación más fácil, sin embargo, el programador, para conseguir potencia y eficiencia en los programas, tenía que seguir pensando en términos de instrucciones de máquina con la dificultad que eso lleva consigo. El siguiente paso para hacer la programación más fácil fue el diseño y construcción de lenguajes de alto nivel que estaban más próximos a los lenguajes hablados como el inglés, francés o español. Así aparecieron lenguajes como COBOL, Basic, FORTRAN, Pascal, C, C++, Java o C#, denominados lenguajes de alto nivel y cuyas instrucciones eran: print, read, open, write, for, while....

Al igual que sucede con los lenguajes ensambladores las computadoras no pueden ejecutar directamente instrucciones escritas en un lenguaje de alto nivel. Se necesita un programa traductor denominado compilador que traduzca las instrucciones en lenguaje de alto nivel a instrucciones en lenguaje máquina, bien directamente como en el caso de C++ o C, o bien un lenguaje intermedio llamado bytecode que luego se interpreta en un lenguaje máquina en el caso de Java. En el capítulo 2 profundizaremos en estos conceptos.

¿Sabía que..?**Compilador**

Un programa que traduce un programa escrito en un lenguaje de alto nivel en un programa equivalente en lenguaje máquina (en el caso de Java, este lenguaje máquina es el bytecode).

El proceso de programación

La programación es un proceso de resolución de problemas. Para resolver estos problemas se requieren técnicas diferentes que comprenden desde el análisis del problema, especificación de requisitos o requerimientos y las etapas de diseño denominadas algoritmos.

Un algoritmo es el concepto fundamental de la ciencia de las computadoras (informática). Desde un punto de vista práctico, un algoritmo es un conjunto de pasos que definen cómo se realiza una tarea. Por ejemplo, hay un algoritmo para construir un modelo de avión, de un tren, una lavadora o un aparato de televisión. Antes de que una máquina pueda ejecutar una tarea, se debe diseñar un algoritmo que posteriormente se pueda convertir en un programa que entienda la computadora para lo cual se necesitará traducir el programa en lenguaje máquina mediante un traductor (compilador o intérprete).

¿Sabía que..?**Algoritmo**

Proceso de resolución de problemas compuesto por un conjunto de instrucciones que se realizan paso a paso para conseguir una solución y que se obtiene en un tiempo finito.

El estudio de algoritmos es uno de los temas centrales en estudios de ciencias e ingeniería y en particular en ingeniería de sistemas computacionales y en ingeniería informática. El conocimiento de las técnicas de diseño de algoritmos es fundamental para el programador y uno de los temas centrales de nuestra obra.

Para desarrollar un programa que pueda resolver un problema, se debe comenzar analizando y examinando el problema con detenimiento con objeto de obtener diferentes opciones para encontrar la solución; a continuación se selecciona una opción y se debe diseñar el algoritmo correspondiente escribiendo las instrucciones del programa en un lenguaje de alto nivel, etapa conocida como codificación del programa y a continuación introducir y ejecutar el programa en la computadora.

Las computadoras no entienden los algoritmos, por lo que es necesario indicarles exactamente las “acciones que deben hacer” en un lenguaje comprensible para la máquina. La descripción de las acciones o tareas que debe hacer la computadora se denomina “programa” y programación a la actividad de escribir y verificar tales programas. Por esta razón se necesitará convertir el algoritmo en un programa mediante un lenguaje de programación cuyas instrucciones sí entiende ya la computadora.

El proceso de resolución de problemas (proceso de programación) con una computadora consta de las siguientes etapas:

- Análisis del problema, con estudio de los requisitos y del dominio del problema.
- Diseño del algoritmo que resuelva el problema.
- Verificar el funcionamiento correcto del algoritmo.
- Implementar el algoritmo en un lenguaje de programación como Java.
- Ejecutar el programa.
- Depurar el programa para obtener y corregir errores.
- Mantener el programa.
- Actualizar el programa.

En el capítulo 2 profundizaremos en el proceso de programación que nos permita adquirir las competencias necesarias para iniciar el aprendizaje de las técnicas de programación y obtener los conocimientos precisos a fin de comenzar a diseñar y construir programas. Como ya se ha comentado, los programas que construyen los programadores se dividen en dos grandes grupos: programas de sistemas,

los más especializados, y programas de aplicaciones, que son los más utilizados en negocios, ingeniería y educación.

Los programas de aplicación realizan una tarea específica. Ejemplos de este tipo de programas son los procesadores de texto, hojas de cálculo, juegos, simulaciones. Tanto los programas de sistemas, en torno a sistemas operativos, como los programas de aplicaciones se escriben en lenguajes de programación.

1.8 Internet y la Web

El origen de la actual red de internet se remonta a la creación de la red ARPANET en noviembre de 1969 que conectaba a diferentes computadoras de la época, con objetivos estrictamente militares y con la finalidad de transmisión de datos entre computadoras conectadas. Victor Cerf y Bob Khan publicaron en 1974 el protocolo TCP/IP (Protocolo de Control de Transmisión/protocolo de internet) y fue el detonante para la expansión en la década de 1980 de la ya conocida como red internet. Comienza a expandirse el correo electrónico, la mensajería instantánea, los sistemas de nombres de dominio (DNS). México es el primer país hispano que tuvo conexión a internet en 1989 y un año más tarde lo hicieron España, Argentina y Chile. En 1989 el investigador Tim Berners Lee del CERN suizo presentó un software basado en protocolos que permitían visualizar la información con el uso de hipertexto. Suele considerarse a 1989 como el año de creación de la Web (World Wide Web) aunque fue unos años más tarde cuando se comenzó a expandir; asimismo, se considera a 1991 como el año del lanzamiento del lenguaje HTML (HyperText Markup Language), que se convertiría en los años posteriores en el estándar de diseño web. En 1994 se creó el W3C (World Wide Web Consortium), organismo mundial que gestiona la Web actual, aunque actualmente la ISOC (Internet Society), la Sociedad Internet, creada en 1992, junto con InterNic y la ICANN (Internet Corporation for Assigned Names and Numbers) son las instituciones que gestionan los nombres de dominio en la red.

La información en la Web se presenta en páginas que se entrelazan unas con otras en la telaraña universal que constituye la World Wide Web. Las páginas web residen en un sitio web que se identifica por su dirección, la URL (*Uniform Resource Locator*). La World Wide Web (*WWW*) o simplemente la Web fue creada en 1989 por Bernards Lee en el CERN (European Laboratory for Particles Physics) aunque su difusión masiva comenzó en 1993 como medio de comunicación universal. La Web es un sistema de estándares aceptados universalmente para almacenamiento, recuperación, formateado y visualización de información, utilizando una arquitectura cliente/servidor. Se puede utilizar la Web para enviar, visualizar, recuperar y buscar información o crear una página web. La Web combina texto, hipermedia, sonidos y gráficos, utilizando interfaces gráficas de usuario para una visualización fácil.

La navegación por internet y a través de la Web se realiza mediante navegadores (*browsers*). Para acceder a la Web se necesita un programa denominado navegador web (*browser*). Un navegador es una interfaz gráfica de usuario que permite “navegar” a través de la Web. Se utiliza el navegador para visualizar textos, gráficos y sonidos de un documento web y activar los enlaces (*links*) o conexiones a otros documentos. Cuando se hace clic (con el ratón) en un enlace a otro documento se produce la transferencia de ese documento situado en otra computadora a su propia computadora. Los navegadores más populares son: Explorer 8 de Microsoft, Firefox 3.6 de Mozilla, Chrome de Google, Safari de Apple y Opera.

La World Wide Web está constituida por millones de documentos enlazados entre sí, denominados páginas web. Una página web, normalmente, está construida por texto, imágenes, audio y video, al estilo de la página de un libro. Una colección de páginas relacionadas, almacenadas en la misma computadora, se denomina sitio web (*Website*). Un sitio web está organizado alrededor de una página inicial (*home page*) que sirve como página de entrada y punto de enlace a otras páginas del sitio. Cada página web tiene una dirección única, conocida como URL. Por ejemplo, la URL de la página inicial de este libro es: www.mhe.es/joyanes.

La Web se basa en un lenguaje estándar de hipertexto denominado HTML que da formatos a documentos e incorpora enlaces dinámicos a otros documentos almacenados en la misma computadora o en computadoras remotas. El navegador web está programado de acuerdo con el estándar citado. Los documentos HTML, cuando ya se han situado en internet, se conocen como páginas web y el conjunto de páginas web pertenecientes a una misma entidad (empresa, departamento, usuario individual) se conoce como sitio web (*Website*). En los últimos años ha aparecido un nuevo lenguaje de marcación para formatos, heredero de HTML, y que se está convirtiendo en estándar universal: el lenguaje XML.

Otros servicios que proporciona la Web y ya muy populares para su uso en el mundo de la programación son: el correo electrónico y la mensajería instantánea. El correo electrónico (*e-mail*) utiliza protocolos específicos para el intercambio de mensajes: SMTP (*Simple Mail Transfer Protocol*), POP (*Post Office Protocol*) e IMAP (*Internet Message Action Protocol*). La mensajería instantánea o chat permite el diálogo en línea simultánea entre dos o más personas, y cuya organización y estructura han sido trasladadas a los teléfonos celulares, en los que también se puede realizar este tipo de comunicaciones con mensajes conocidos como “cortos” SMS (*short message*) o MMS (*multimedia message*).

La revolución Web 2.0

La revista norteamericana *Time* declaró “Personaje del año 2006” a “You” (usted, el usuario) para referirse a la democracia digital de la Web, a la nueva Web participativa y colaboradora, en la que el usuario anónimo, las personas ordinarias que navegaban por la Web, eran los actores principales de esta nueva sociedad que se estaba creando en torno a lo que comenzó a denominarse la Web Social y que se apoyaba en el nuevo concepto de Web 2.0.

La Web 2.0 se caracteriza por una arquitectura de participación y que impulsa la interacción de los usuarios junto a la colaboración y participación de la comunidad. El desarrollo del software se apoya en la arquitectura de participación y colaboración.

El software de código abierto (*open source*) está disponible para todos aquellos usuarios y programadores que desean utilizarlo y modificarlo sin restricciones. El usuario no solo contribuye con contenido y con el desarrollo de software de código abierto sino que controla el uso de los medios y decide en qué fuentes de información debe confiar.

Desde la publicación en 2005 del artículo “What is Web 2.0?” de Tim O'Reilly, considerado el padre intelectual del concepto de Web 2.0, mucho ha evolucionado la Web, aunque la mayoría de las características que definían a la Web 2.0 no solo se han consolidado sino que se han popularizado en límites inimaginables. Las características tecnológicas de la Web 2.0 son innumerables aunque desde el punto de vista de desarrollo de software para Web se apoyan en Ajax, un conjunto de tecnología en torno a componentes, como XHTML, Hoja de Estilo en Cascada (CSS), Java Script, el Modelo de Objetos de Documento (DOM), XML y el objeto XMLHttpRequest y kit de herramientas Ajax como Doj.

La Web 2.0 ha generado un nuevo conjunto de aplicaciones ricas de internet (RIA, *Rich Internet Applications*). Google Maps o Gmail han facilitado el desarrollo y comprensión de estas aplicaciones. Se está produciendo un cambio significativo en el modo de construcción de las aplicaciones. En lugar de centrarse en el sistema operativo como plataforma para construir y utilizar aplicaciones, el foco se centra en el navegador web como plataforma para la construcción y utilización de las aplicaciones: el concepto nuevo de “la Web como plataforma”. La Web ha desarrollado una plataforma rica que puede ejecutar aplicaciones independientes del dispositivo o sistema operativo que se esté utilizando. RIA depende esencialmente de AJAX (acrónimo de *Asynchronous Java Script* y XML). El soporte de AJAX ha mejorado la experiencia web convirtiendo las aplicaciones sencillas de HTML en aplicaciones ricas, potentes e interactivas.

Social Media

Uno de los fenómenos más sobresalientes de la Web es la emergencia y advenimiento de la Social Media que se refiere a las plataformas en línea y las herramientas que utilizan las personas para compartir experiencias y opiniones, incluidas fotos, videos, música, etc. La Social Media comprende blogs, microblogs, wikis, agregadores de contenidos RSS, redes sociales, etcétera.

El contenido generado por el usuario es uno de los resultados de mayor impacto de la Social Media y clave del éxito de empresas Web 2.0 populares y reputadas como Amazon y eBay. El contenido puede ir desde artículos, presentaciones, videos, fotografías, enlaces a otros sitios web, etc. La inteligencia colectiva es otra característica clave de la Web 2.0 que combina el comportamiento, preferencia o ideas de un grupo para generar conocimiento en beneficio de los demás miembros del grupo o la comunidad. Los sistemas de reputación, como los utilizados por eBay, para generar confianza entre compradores y vendedores, que comparten la información con la comunidad. Los sitios de marcadores sociales, Digg, Del.icio.us, Menéame, StumbleUpon, que recomiendan y votan los sitios de favoritos en artículos como libros, videos o fotografías.

Los blogs, wikis, podcast, etc., favorecen la colaboración y participación de los usuarios. Encyclopedias como Wikipedia, Wikilengua o Europeana ayudan en la creación del conocimiento universal. Las

redes sociales, desde la popular Facebook, pasando por MySpace, Bebo, Tuenti, Twitter hasta las redes profesionales como LinkedIn, Xing o Ning, que favorecen las relaciones de todo tipo y un nuevo medio de comunicación de masas.

Desarrollo de programas web

Una de las normas de desarrollo del software Web 2.0 es que sea sencillo y de tamaño reducido. La Web, como ya se ha comentado, se ha convertido en una plataforma de aplicaciones, desarrollo, entrega y ejecución. El escritorio web permite ejecutar aplicaciones en un navegador web, en un entorno similar al de un escritorio. El uso de la Web como plataforma forma parte de un movimiento dirigido a aplicaciones independientes del sistema operativo.

Por último cabe citar otra característica notable que ha traído la Web 2.0: la beta perpetua y el desarrollo ágil. El ciclo de creación y publicación de versiones del software tradicional se está transformando. El desarrollo tradicional de software exigía pruebas exhaustivas y versiones beta para crear una versión definitiva. Hoy la preocupación fundamental de la creación de software es el desarrollo de aplicaciones con versiones más frecuentes, es decir, el periodo de beta perpetua utilizando la Web como plataforma. Las actualizaciones se realizan en los servidores web en los que se almacena la aplicación y la distribución de software en CD se reduce a la mínima expresión.

La Web Semántica y la Web 3.0

La Web Semántica fue un nuevo intento del creador de la World Wide Web, Tim Berners-Lee, para señalar la futura evolución de internet en la Web Semántica. Se trataba de encontrar un nuevo método de estructurar el contenido de internet, por otra parte totalmente desestructurado. El reto era encontrar la posición exacta de la información que se necesitaba para responder a una pregunta concreta. Se trataba de que el contenido de las estructuras de la Web pueda ser comprendido por las máquinas. Se trataba, por ejemplo, de entender que una dirección física es una dirección física y no un número de teléfono o cualquier otro dato. Es decir, añadir la capacidad de marcar o etiquetar información en internet utilizando semántica (significado) que facilita a las computadoras analizar el contenido con más precisión. La Web Semántica introducirá un nuevo nivel de sofisticación a los motores de búsqueda, de modo tal que respondan exactamente a la pregunta que se les plantea y no con una serie, a veces interminable, de enlaces en los que aparecen términos o palabras incluidas en la pregunta. El artículo desencadenante de la ola de popularidad de la Web Semántica se publicó en mayo de 2001 en la revista *Scientific American*.⁶ En este artículo se plantea que la Web Semántica es una extensión de la Web actual dotada de significado, esto es, un espacio en el que la información tendría un significado bien definido de manera que pudiera ser interpretado tanto por agentes humanos como por agentes computarizados (de computadora).

El advenimiento de la Web 2.0 como web social, participativa y colaborativa ha traído un nuevo resurgimiento de Web Semántica y su llegada al gran público. En nuestra opinión consideramos que en esta nueva década del siglo XXI está emergiendo el concepto de Web 3.0 como una nueva generación de la Web, fruto de la convergencia de la Web Semántica y la Web 2.0. Un sitio web representativo de la nueva Web es el buscador semántico Wolfram Alpha.

1.9 Cloud computing (computación en la nube)

Desde mediados de 2008 una nueva arquitectura o paradigma de computación ha nacido y se está implantando de modo acelerado: *cloud computing* (computación en la nube). ¿Qué es la computación en la nube? Este concepto, que en realidad no es nuevo, es una filosofía de trabajo en la computación y en la informática que proviene de ofrecer el hardware, el software, los programas, como servicios, al igual que sucede con cualquier otro servicio común, como la electricidad, el gas, el teléfono, etc. La computación en la nube es un nuevo modelo que ofrece a usuarios, programadores y desarrolladores, la posibilidad de ejecutar los programas directamente, sin necesidad de instalación, ni mantenimiento. Este es el caso de las aplicaciones actuales de la Web, y, en particular, la Web 2.0, en la que el usuario descarga el pro-

⁶ Berners-Lee, Tim, Hendler, Jim y Lasilla, Ora. "The Semantic Web" en *Scientific American*, mayo de 2001.

grama y a continuación lo ejecuta. Este es el caso de Google Maps, Google Street View, las redes sociales, los blogs, etc., o los casos usuales del correo electrónico web como Gmail, Yahoo!, etc., en el que el usuario solo debe ejecutar los programas que se encuentran en la nube (*cloud*), los infinitos sistemas de computadoras y centros de datos existentes en el mundo, en los que hay numerosos servicios (programas, sistemas operativos, redes de comunicaciones, todos ellos “virtuales”) procedentes de numerosos proveedores que ofrecen estos servicios ya sea gratuitos o bien con pago de una tasa diaria, semanal, mensual, etc. Cada día hay más proveedores de servicios en la nube, aunque los más conocidos son: Google, Amazon, Microsoft, IBM, Salesforce, etcétera.

Esta nueva arquitectura y filosofía de computación requiere nuevos conceptos para el desarrollo de programas, centrados fundamentalmente en servicios web, aplicaciones web que residirán en la nube de computadoras, y que los usuarios descargarán y ejecutarán de manera inmediata.

Software como servicio (SaaS)

El software como servicio (SaaS, *Software as a Service*) es un tipo de software que se descarga de un servidor web y no necesita instalación en el equipo del cliente sino solo su ejecución. Este tipo de modelo de software se apoya en la nube. En los servidores del distribuidor de software correspondiente se almacenan los diferentes programas y aplicaciones; el usuario selecciona aquel que desea ejecutar y solo necesita ejecutarlo. El usuario paga una cuota mensual, anual o por cualquier otro periodo, o bien la descarga y ejecución del usuario es gratis. Existen numerosas aplicaciones de software como servicio, entre ellos destacan Windows Office 365, el programa clásico de ofimática de Windows, pero que se puede descargar de la nube y no requiere instalación, y Salesforce. com, un programa de gestión y administración con los clientes, que está configurando un nuevo tipo de desarrollo de aplicaciones de software y unos nuevos modelos de negocios y de aplicaciones comerciales.

Cada día es más frecuente el uso de software como servicio, sobre todo en pequeñas y medianas empresas, y en usuarios finales. En la computación o informática móvil, este tipo de software es todavía más utilizado, ya que existen miles de aplicaciones de software que no requiere más que la descarga y ejecución por parte del usuario; este el caso de numerosos videojuegos, programas de gestión, programas didácticos, etcétera.

1.10 Movilidad: tecnologías, redes e internet móvil

La movilidad (uso de tecnologías móviles o celulares), los medios sociales (*Social Media*), la computación en la nube, los grandes volúmenes de datos (*Big Data*) e internet de las cosas, son las cinco grandes tendencias de la sociedad en general, y los negocios en particular. Todo ello ha potenciado el uso creciente de internet móvil.

¿Qué es **internet móvil**? La respuesta no es fácil, pero se puede concretar en que es la propia red internet pero cuyo acceso se realiza mediante tecnologías móviles (teléfonos inteligentes, tabletas, consolas, etc.) y de un modo ubicuo (en cualquier lugar, con cualquier dispositivo y en cualquier momento). Internet móvil seguirá creciendo como lo demuestran las estadísticas de su penetración en todo el mundo y, en particular, en América Latina, Caribe y España.

Los servicios ofrecidos por las tecnologías de comunicación móvil son innumerables, aunque queremos centrar nuestra atención en el objetivo final de este libro “aprendizaje de la programación”, por lo cual, los servicios de mayor interés son el “desarrollo de aplicaciones (*apps*)”. El soporte de infraestructuras de las tecnologías móviles son las redes inalámbricas (“sin cable” y las redes móviles o celulares), los dispositivos electrónicos (tabletas, teléfonos inteligentes, consolas...), los sistemas operativos móviles y las aplicaciones móviles (*apps*).

Redes inalámbricas

El internet móvil tiene como infraestructura principal dos tipos de comunicaciones, ambas sin cables: inalámbricas y móviles. Las redes inalámbricas se dividen en cuatro grandes categorías:

- WPAN (Red inalámbrica de área personal). Constituida por tecnologías móviles como Bluetooth, RFID, NFC, sensores...

- WLAN (Red inalámbrica de área local). Integrada, fundamentalmente por las redes WiFi que pueblan los campus de las universidades, los aeropuertos, los centros comerciales, y de ocio... y cada día más las grandes ciudades y sus grandes superficies.
- WMAN (Red inalámbrica de área metropolitana). Constituida fundamentalmente por redes WiMax (similares a las WiFi pero de mayor alcance) y LMDC.
- WWAN (Red inalámbrica de área amplia). Integrada por redes de gran alcance como las formadas por satélites y las propias redes móviles.

Redes móviles

Las redes móviles (celulares) dan soporte a los teléfonos móviles, y, especialmente, a los teléfonos inteligentes, tabletas, etc. Las redes existentes actuales se apoyan en tecnologías GSM, GRPS, EDGE, CDMA...y, las modernas HSDPA, HSUPA, HSPA, HSPA+ ..., que han aumentado su alcance considerablemente. Las redes móviles son muy conocidas por la generación a la cual pertenecen. Las redes más populares implantadas en la actualidad son las 3G (3G, 3.5G y 3.75G) y las 4G (LTE, *Long Term Evolution*).

Las tecnologías 3G de gran velocidad nacieron con las redes UMTS y pronto fueron evolucionando a HSDPA (3.5G), HSUPA (3.75G), HSPA+ (3.5g+). Estas son las redes más extendidas en el mundo. Sin embargo desde 2012 y, en particular, 2013, tanto en numerosos países de Latinoamérica como en España han comenzado a desplegarse redes 4G que ofrecen altas velocidades de descarga de datos de hasta 326 Mbps (Megabits por segundo) y de hasta 86.4 Mbps de subida de datos. Las redes 3G y 4G convivirán durante muchos años, como ha sucedido con las anteriores generaciones, pero posiblemente 2014 y 2015 serán los años del despegue masivo de las redes 4G.

A finales de 2011, la UIT (Unión Internacional de Telecomunicaciones) acordó las especificaciones para las tecnologías móviles de cuarta generación denominadas IMT Advanced, que agrupa a dos tecnologías: LTE Advanced (la evolución de la actual LTE) y WiMax Advanced (también conocida como WiMax 2).

Sistemas operativos móviles

Ya hicimos anteriormente un avance de los sistemas operativos móviles. Desde el punto de vista de la programación, cada sistema operativo ofrece a los desarrolladores una interfaz de programación denominado API (*Application Programming Interface*) que facilita que las aplicaciones se puedan ejecutar en dispositivos diferentes que funcionen sobre el mismo sistema operativo.

Los sistemas operativos propietarios más comunes en el mercado móvil actual son:

- iOS de Apple para dispositivos iPhone, iPad, iPod, computadoras personales Mac (su última versión presentada es la iOS 7 y la iOS 7.1).
- Android de Google, que si bien trabaja en modo abierto, no deja de ser propietario de todos aquellos fabricantes que lo soportan, La última versión es Android 4.4.
- Blackberry. El sistema operativo de los populares teléfonos Blackberry de la empresa canadiense RIM cuyo nombre se cambió a Blackberry. Su última versión presentada a los primeros días de enero de 2013 es BBM 10.
- Windows Phone 8. El sistema operativo de Microsoft que tiene una alianza con Nokia para el desarrollo de teléfonos móviles con su sistema operativo.

Los sistemas operativos abiertos comienzan a presentarse y comercializarse a lo largo del 2013 y serán incorporados a teléfonos de diferentes fabricantes. Entre ellos destacan, Firefox OS, soportado por numerosos fabricantes y operadoras de telecomunicaciones, en particular el fabricante chino ZEN y la operadora española Telefónica (que presentaron comercialmente sus primeros teléfonos el 1 de julio de 2013 en Madrid), Tizen y Ubuntu de Canonical.

Aplicaciones móviles

Las aplicaciones (programas de aplicación) para móviles son las creadas de manera expresa para entornos o plataformas móviles, fundamentalmente teléfonos inteligentes y tabletas. Existen dos tipos de aplicaciones móviles: nativas (*apps*) y web.

- **Aplicación móvil nativa (app).** Son aplicaciones diseñadas para ejecutarse sobre una plataforma móvil y un sistema operativo concreto (iOS y Android tienen la mayor cuota de mercado, en la actualidad). Necesitan descargarse de una tienda de aplicaciones (App Store de Apple, Google Play de Android, Windows Mobile Marketplace de Microsoft, Marketplace de Blackberry... o de otras empresas como Telefónica, Vodafone, Verizon o Telmex. Se descargan, instalan y ejecutan en el dispositivo móvil de modo gratuito o con el pago de una cuota.
- **Aplicación web móvil.** Se desarrollan con tecnologías web y se puede acceder a ellas y visualizarlas con un navegador de internet (Safari, Firefox, Chrome, Opera...)

Los lenguajes de programación empleados en el desarrollo de aplicaciones móviles son muy variados y van desde los lenguajes tradicionales como C++, Java, HTML5... hasta clásicos de la programación orientada a objetos como Objective-C que fue seleccionado por Apple para sus desarrollos.

1.11 Geolocalización y realidad aumentada

Los servicios de geolocalización (LBS, *Location Based System*) son servicios móviles basados en la localización o ubicación física (posición geográfica) del dispositivo del usuario de un teléfono móvil. La geolocalización implica el posicionamiento que define la ubicación de un objeto (el teléfono móvil de un usuario) en un sistema de coordenadas geográficas determinada. La fijación de la posición puede realizarse mediante distintos procedimientos:

- La dirección IP (*Internet Protocol*) de internet.
- Identificación de las redes WiFi cercanas.
- Estaciones base de telefonía móvil.
- Señales GPS (sistemas de posicionamiento global).

Existen numerosas aplicaciones de geolocalización.

- Navegación basada en la localización (búsqueda en Google, publicidad en línea)....
- Servicios propios basados en localización (Google Places, Layar, Google Latitude...).
- Geolocalización en redes sociales (Twitter, Foursquare, Facebook Places, Waze...).

¿Qué es la realidad aumentada?

La realidad aumentada es una tecnología que consiste en sobreponer imágenes virtuales sobre la realidad que vemos a través de una pantalla. De esta manera, y gracias a que sus aplicaciones son infinitas, difumina la frontera entre el mundo real y el virtual abriendo una nueva dimensión en la manera en la que interactuamos con lo que nos rodea. El término realidad aumentada fue acuñado por Tom Caudell en la década de 1990 cuando desarrolló un visor para guiar a los trabajadores en las instalaciones eléctricas de los aviones. Durante años, la realidad aumentada ha sido utilizada en el mundo de los videojuegos y recientemente se encuentra en pleno auge debido a las posibilidades que ofrece esta tecnología en los dispositivos móviles. La realidad aumentada en móviles utiliza como escenario todo aquello que nos rodea y gracias a sus GPS incorporados es posible saber en dónde estamos y lo que hay a nuestro alrededor para mostrarnos información útil en ese momento y lugar. Además, con el acelerómetro o sensor de movimiento los *smartphones* pueden saber hacia dónde mira el móvil y así determinar qué imagen mostrar en esa dirección.

La realidad aumentada ha disparado su uso en el ámbito comercial, desde su integración con los dispositivos móviles. Existen numerosas aplicaciones móviles de realidad aumentada, tal vez la más conocida y popular sea Wikitude.

1.12 Internet de las cosas

La tecnología internet de las cosas (*Internet of Things*) se ha ido asentando y popularizando en los últimos años gracias a la implantación de tecnologías como RFID (identificación por radiofrecuencia), Bluetooth, NFC (proximidad), sensores y tecnologías móviles o inalámbricas como WiFi y WiMax. Todas estas tecnologías han hecho que cada día más existan millones de cosas que transmiten información en tiempo real y bajo demanda, entre sí y a otras cosas.

Internet de las cosas o internet de los objetos, como también comienza ya a conocerse, es una tendencia emergente e imparable y que se refiere al hecho de que miles de cosas u objetos de todo tipo: libros, zapatos, electrodomésticos, vehículos, trenes, sensores, se pueden conectar entre sí y, a su vez, conectarse a internet para intercambiar información de todo tipo entre ellos mismos. Es decir, los objetos se interconectan entre sí. Como a los objetos se les puede asociar una dirección de internet (necesaria para su conexión a la red) IP, se podrán formar grandes redes de objetos. Como el nuevo protocolo IPv6 va a permitir miles y miles de millones de direcciones IP, eso significa que miles de millones de objetos podrán tener asociada una dirección IP y por consiguiente conectarse con internet y con otros objetos en tiempo real.

En esencia, el internet de las cosas se está refiriendo a la integración de sensores, chips RFID, chips NFC, en dispositivos y objetos cotidianos que quedarán conectados a internet a través de las redes fijas o inalámbricas. Serán fácilmente integrables en hogares, entornos de trabajo y lugares públicos. Cualquier objeto podrá conectarse con otro objeto y a su vez ambos a internet.

Esta nueva tendencia tecnológica exigirá la programación de aplicaciones de todo tipo que deberán escribir programadores para atender a esta nueva demanda social y tecnológica. La programación del futuro, ya casi presente, se verá muy afectada por el internet de las cosas, ya que todas tendrán conexión a internet en cualquier momento y lugar. El mundo conectado en que vivimos no solo conectará a personas sino que conectará a esas personas con miles de millones de objetos a través de internet.

1.13 Big data. Los grandes volúmenes de datos

Big Data (grandes datos, grandes volúmenes de datos o *macrodatos*) como recomienda utilizar la Fundación Fundéu BBVA, *Fundación del español urgente*) supone la confluencia de una multitud de tendencias tecnológicas que venían madurando desde la primera década del siglo XXI, y que se han consolidado durante los años 2011 a 2013, cuando han detonado e irrumpido con gran fuerza en organizaciones y empresas, en particular, y en la sociedad, en general: movilidad, redes sociales, aumento de la banda ancha y reducción de su costo de conexión a internet, medios sociales (en particular las redes sociales) internet de las cosas, geolocalización, y de modo muy significativo la computación en la nube (*cloud computing*).

Una definición muy significativa es del McKinsey Global Institute, que en un informe muy reconocido y referenciado, de mayo de 2011, define el término del siguiente modo: “*Big Data* se refiere a los conjuntos de datos cuyo tamaño está más allá de las capacidades de las herramientas típicas de software de bases de datos para capturar, almacenar, gestionar y analizar”. El término *Big Data* en realidad ha surgido debido esencialmente al crecimiento exponencial de los datos, especialmente, de datos no estructurados (los datos no guardados en las bases de datos tradicionales) como datos procedentes de redes sociales, de sensores, de la biometría, audio, video, fotografía, etcétera.

Big Data se caracteriza por sus tres grandes dimensiones conocidas como el “Modelo de las tres V” (3 V o V3): volumen, velocidad y variedad (*variety*). El volumen se refleja hoy día en petabytes y exabytes. En 2000, se almacenaron en el mundo 800 000 petabytes y se espera que en 2020 se alcancen los 40 zettabytes (ZB). Solo Twitter genera más de 12 terabytes (TB) de datos cada día, Facebook, 15 TB; los servidores de Google, 1 petabyte cada hora, y algunas empresas generan terabytes de datos cada hora cada día del año. La importancia de la velocidad de los datos o el aumento creciente de los flujos de datos en las organizaciones junto con la frecuencia de las actualizaciones de las grandes bases de datos son características importantes a tener en cuenta. Esto requiere que su procesamiento y posterior análisis, normalmente, ha de hacerse en tiempo real para mejorar la toma de decisiones sobre la base de la información generada. Las fuentes de datos son de cualquier tipo. La variedad se manifiesta en que los datos pueden ser estructurados y no estructurados (texto, datos de sensores, audio, video, flujos de clics, archivos logs), y cuando se analizan juntos se requieren nuevas técnicas. Imaginemos el registro en vivo de imágenes de las cámaras de video de un estadio de fútbol o de vigilancia de calles y edificios. En los sistemas de *Big Data* las fuentes de datos son diversas y no suelen ser estructuras relacionales típicas. Los datos de redes sociales, de imágenes pueden venir de una fuente de sensores y no suelen estar preparados para su integración en una aplicación.

1.14 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, ya que el pseudocódigo o el diagrama de flujo no son comprensibles por la computadora, aunque pueda entenderlos cualquier programador. 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. El proceso de traducir un algoritmo en pseudocódigo a un lenguaje de programación se denomina **codificación**, y el algoritmo escrito en un lenguaje de programación se denomina **código fuente**.

En la realidad la computadora no entiende directamente los lenguajes de programación sino que se requiere un programa que traduzca el código fuente a otro lenguaje que sí entiende la máquina directamente, pero es muy complejo para las personas; este lenguaje se conoce como **lenguaje máquina** y el código correspondiente **código máquina**. Los programas que traducen el código fuente escrito en un lenguaje de programación, tal como C++, a código máquina se denominan **traductores**. El proceso de conversión de un algoritmo escrito en pseudocódigo hasta un programa ejecutable comprensible por la máquina, se muestra en la figura 1.8.

Hoy en día, la mayoría de los programadores emplean lenguajes de programación como **C++**, **C**, **C#**, **Java**, **Visual Basic**, **XML**, **HTML**, **Perl**, **PHP**, **JavaScript**, aunque todavía se utilizan, sobre todo profesionalmente, los clásicos COBOL, FORTRAN, Pascal o el mítico BASIC. Estos lenguajes se denominan lenguajes de alto nivel y permiten a los profesionales resolver problemas convirtiendo sus algoritmos en programas escritos en alguno de estos lenguajes de programación.

Los **lenguajes de programación** se utilizan para escribir programas. Los programas de las computadoras modernas constan de secuencias de instrucciones que se codifican como secuencias de dígitos numéricos que podrán entender dichas computadoras. El sistema de codificación se conoce como lenguaje máquina, que es el lenguaje nativo de una computadora. Desgraciadamente la escritura de programas en lenguaje máquina es una tarea tediosa y difícil ya que sus instrucciones son secuencias de 0 y 1 (patrones de bit, como 11110000, 01110011...), que son muy difíciles de recordar y manipular por las personas. En consecuencia, se necesitan lenguajes de programación “amigables con el programador” que permitan escribir los programas para poder “charlar” con facilidad con las computadoras. Sin embargo, las computadoras solo entienden las instrucciones en lenguaje máquina, por lo que será preciso traducir los programas resultantes a lenguajes de máquina antes de que puedan ser ejecutadas por ellas.

Cada lenguaje de programación tiene un conjunto o “juego” de instrucciones (acciones u operaciones que debe realizar la máquina) que la computadora podrá entender directamente en su código máquina o bien se traducirán a dicho código máquina. Las instrucciones básicas y comunes en casi todos los lenguajes de programación son:

- *Instrucciones de entrada/salida*. Instrucciones de transferencia de información entre dispositivos periféricos y la memoria central, como “leer de...” o bien “escribir en...”
- *Instrucciones de cálculo*. Instrucciones para que la computadora pueda realizar operaciones aritméticas.
- *Instrucciones de control*. Instrucciones que modifican la secuencia de la ejecución del programa.

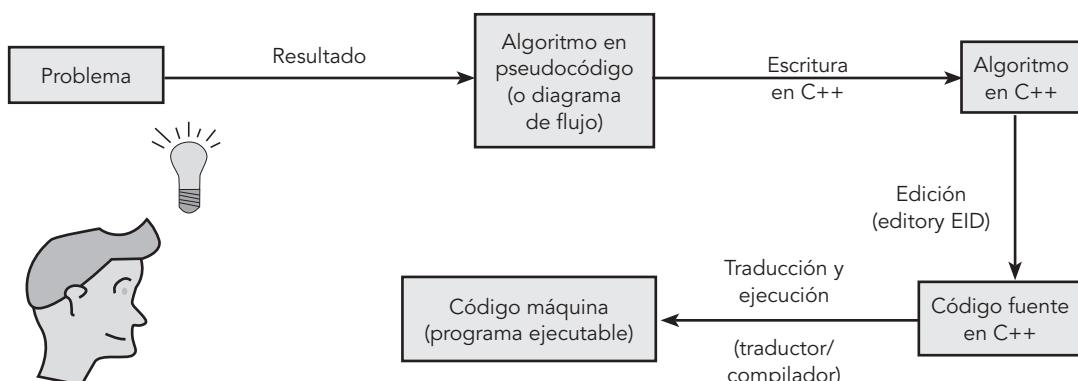


Figura 1.8 Proceso de transformación de un algoritmo en pseudocódigo en un programa ejecutable.

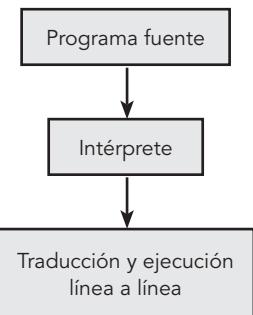


Figura 1.9 Intérprete.

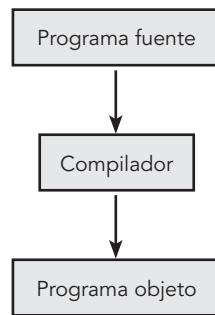


Figura 1.10 La compilación de programas.

Además de estas instrucciones y dependiendo del procesador y del lenguaje de programación existirán otras que conformarán el conjunto de instrucciones y junto con las reglas de sintaxis permitirán escribir los programas de las computadoras. Los principales tipos de lenguajes de programación son:

- *Lenguajes máquina.*
- *Lenguajes de bajo nivel (ensambladores).*
- *Lenguajes de alto nivel.*

Traductores de lenguaje: el proceso de traducción de un programa

El proceso de traducción de un programa fuente escrito en un lenguaje de alto nivel a un lenguaje máquina comprensible por la computadora, se realiza mediante programas llamados “**traductores**”. 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 **compiladores** e **intérpretes**.

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, mas que en circunstancias especiales. Sin embargo, está muy extendida la versión interpretada del **lenguaje Smalltalk**, un lenguaje orientado a objetos puro. El sistema de traducción consiste en: traducir la primera sentencia del programa a lenguaje máquina, se detiene la traducción, se ejecuta la sentencia; a continuación, se traduce la siguiente sentencia, se detiene la traducción, se ejecuta la sentencia y así sucesivamente hasta terminar el programa (figura 1.9).

Compiladores

Un **compilador** es un programa que traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. La traducción del programa completo se realiza en una sola operación denominada compilación del programa; es decir, se traducen todas las instrucciones del programa en un solo bloque. El programa compilado y depurado (eliminados los errores del código fuente) se denomina programa ejecutable porque ya se puede ejecutar directamente y cuantas veces se desee; solo deberá volver a compilarse de nuevo en el caso de que se modifique alguna instrucción del programa. De este modo el programa ejecutable no necesita del compilador para su ejecución. Los lenguajes compilados típicos más utilizados son: **C, C++, Java, C#, Pascal, FORTRAN y COBOL** (figura 1.11).

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.

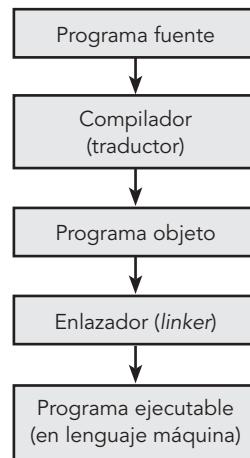


Figura 1.11 Fases de la compilación.

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 (figura 1.11).

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).
2. Introducir el *programa fuente* en la memoria.
3. *Compilar* el *programa* con el compilador del lenguaje de programación.
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 se muestra en las figuras 1.12 y 1.13.

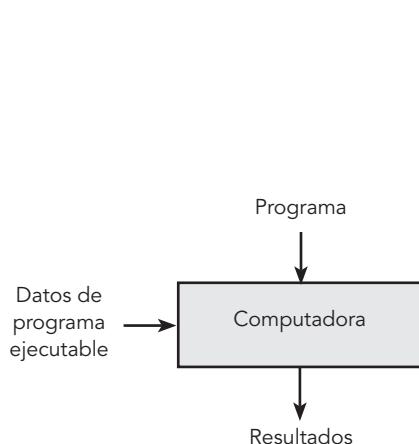


Figura 1.12 Ejecución de un programa.

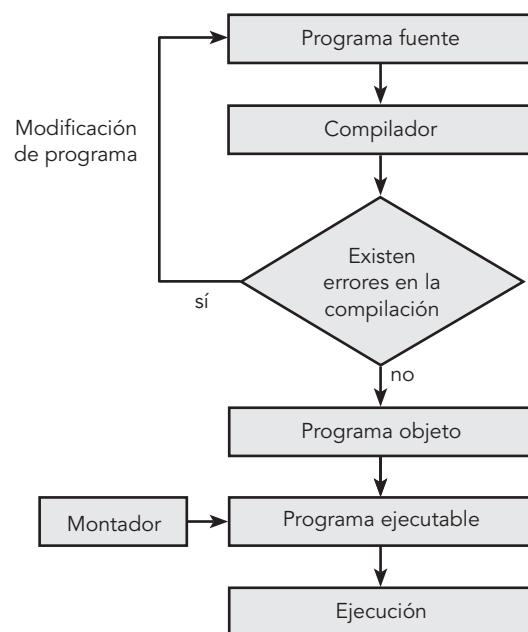


Figura 1.13 Fases de ejecución de un programa.

1.15 Evolución de los lenguajes de programación

En la década de 1940 cuando nacían las primeras computadoras digitales el lenguaje que se utilizaba para programar era el *lenguaje máquina* que traducía directamente el código máquina (código binario) comprensible para las computadoras. Las instrucciones en lenguaje máquina dependían de cada computadora y debido a la dificultad de su escritura, los investigadores de la época simplificaron el proceso de programación inventando sistemas de notación en los cuales las instrucciones se representaban en formatos **nemáticos (nemotécnicos)** en vez de en formatos numéricos que eran más difíciles de recordar. Por ejemplo, mientras la instrucción

Mover el contenido del registro 4 al registro 8

se podía expresar en lenguaje máquina como

4048 o bien 0010 0000 0010 1000

en código nemotécnico podía aparecer como

MOV R5, R6

Para convertir los programas escritos en código nemotécnico a lenguaje máquina, se crearon programas ensambladores (*assemblers*). Es decir, los ensambladores son programas que traducen otros programas escritos en código nemotécnico en instrucciones numéricas en lenguaje máquina que son compatibles y legibles por la máquina. Estos programas de traducción se llaman ensambladores porque su tarea es ensamblar las instrucciones reales de la máquina con los nemotécnicos e identificadores que representan las instrucciones escritas en ensamblador. A estos lenguajes se les denominó de segunda generación, reservando el nombre de primera generación para los lenguajes de máquina.

En las décadas de 1950 y 1960 comenzaron a desarrollarse lenguajes de programación de tercera generación que diferían de las generaciones anteriores en que sus instrucciones o primitivas eran de alto nivel (comprendibles por el programador, como si fueran **lenguajes naturales**) e **independientes de la máquina**. Estos lenguajes se llamaron lenguajes de alto nivel. Los ejemplos más conocidos son **FORTRAN** (FORmula TRANslator) que fue desarrollado para aplicaciones científicas y de ingeniería, y **COBOL** (COmmon Business Oriented Language), que fue inventado por la U.S. Navy de Estados Unidos, para aplicaciones de gestión o administración. Con el paso de los años aparecieron nuevos lenguajes como **Pascal, BASIC, C, C++, Ada, Java, C#, HTML, XML...**

Los lenguajes de programación de alto nivel se componen de un conjunto de instrucciones o primitivas más fáciles de escribir y recordar su función que los lenguajes máquina y ensamblador. Sin embargo, los programas escritos en un lenguaje de alto nivel, como C o Java necesitan ser traducidos a código máquina; para ello se requiere un programa denominado **traductor**. Estos programas de traducción se denominaron técnicamente, **compiladores**. De este modo existen compiladores de C, FORTRAN, Pascal, Java, etcétera.

También surgió una alternativa a los traductores compiladores como medio de implementación de lenguajes de tercera generación que se denominaron **intérpretes**.⁸ Estos programas eran similares a los traductores excepto que ellos ejecutaban las instrucciones a medida que se traducían, en lugar de guardar la versión completa traducida para su uso posterior. Es decir, en vez de producir una copia de un programa en lenguaje máquina que se ejecuta más tarde (este es el caso de la mayoría de los lenguajes, C, C++, Pascal, Java...), un intérprete ejecuta realmente un programa desde su formato de alto nivel, instrucción a instrucción. Cada tipo de traductor tiene sus ventajas e inconvenientes, aunque hoy día prácticamente los traductores utilizados son casi todos compiladores por su mayor eficiencia y rendimiento.

Sin embargo, en el aprendizaje de programación se suele comenzar también con el uso de los lenguajes algorítmicos, similares a los lenguajes naturales, mediante instrucciones escritas en pseudocódigo (o seudocódigo) que son palabras o abreviaturas de palabras escritas en inglés, español, portugués, etc. Posteriormente se realiza la conversión al lenguaje de alto nivel que se vaya a utilizar realmente en la computadora, como C, C++ o Java. Esta técnica facilita la escritura de algoritmos como paso previo a la programación.

⁸ Uno de los intérpretes más populares en las décadas de 1970 y 1980 fue Basic.

1.16 Paradigmas de programación

La evolución de los lenguajes de programación ha ido paralela a la idea de paradigma de programación: enfoques alternativos a los procesos de programación. En realidad un paradigma de programación representa fundamentalmente enfoques diferentes para la construcción de soluciones a problemas y por consiguiente afectan al proceso completo de desarrollo de software. Los paradigmas de programación clásicos son: procedimental (o imperativo), funcional, declarativo y orientado a objetos. En la figura 1.14 se muestra la evolución de los paradigmas de programación y los lenguajes asociados a cada paradigma [Brooks 05].⁹

Lenguajes imperativos (procedimentales)

El paradigma imperativo o procedimental representa el enfoque o método tradicional de programación. Un lenguaje imperativo es un conjunto de instrucciones que se ejecutan una por una, de principio a fin, de modo secuencial excepto cuando intervienen instrucciones de salto de secuencia o control. Este paradigma define el proceso de programación como el desarrollo de una secuencia de órdenes (comandos) que manipulan los datos para producir los resultados deseados. Por consiguiente, el paradigma imperativo señala un enfoque del proceso de programación mediante la realización de un algoritmo que resuelve de modo manual el problema y a continuación expresa ese algoritmo como una secuencia de órdenes. En un lenguaje procedimental cada instrucción es una orden u órdenes para que la computadora realice alguna tarea específica.

Lenguajes declarativos

En contraste con el paradigma imperativo el paradigma declarativo solicita al programador que describa el problema en lugar de encontrar una solución algorítmica al problema; es decir, un lenguaje declarativo

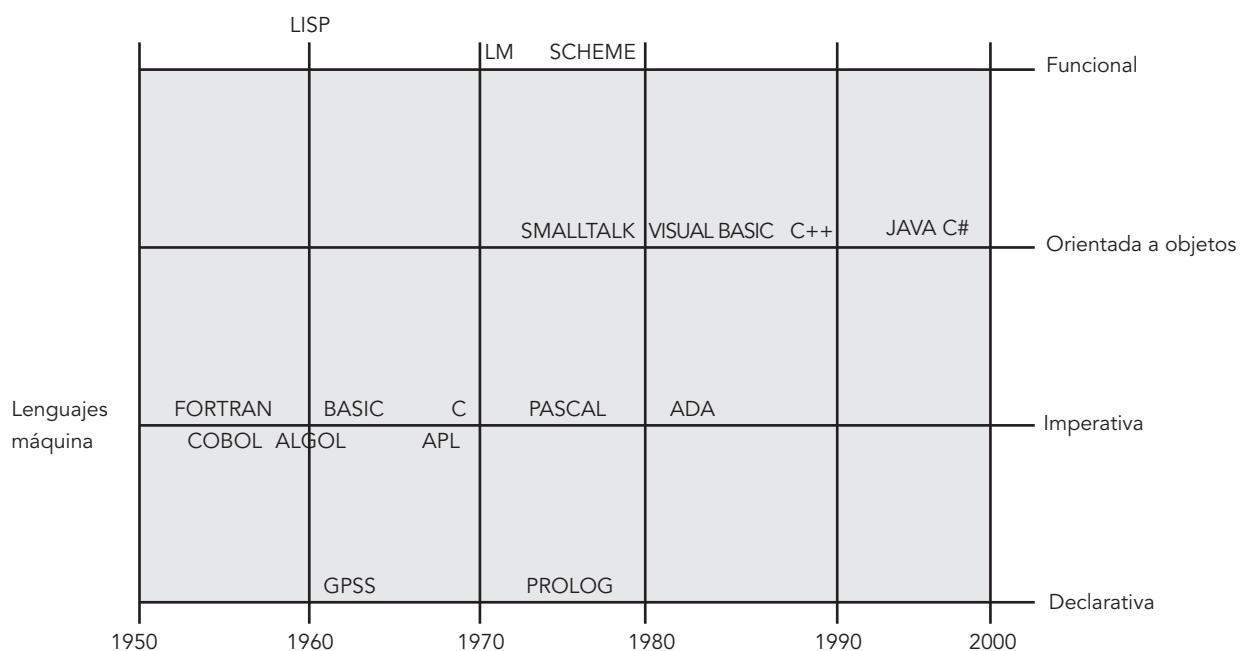


Figura 1.14 Paradigmas de programación (evolución de lenguajes). Fuente: Brockshear, 2005.

⁹ J. Glenn Brockshear, *Computer Science: An overview*, 8a. ed., Boston: Pearson/Addison Wesley, 2005, p. 230. Obra clásica y excelente para la introducción a la informática y a las ciencias de la computación en todos sus campos fundamentales. Esta obra se recomienda a todos los lectores que deseen profundizar en los diferentes temas tratados en este capítulo y ayudará considerablemente al lector como libro de consulta en su aprendizaje en programación.

utiliza el principio del razonamiento lógico para responder a las preguntas o cuestiones consultadas. Se basa en la lógica formal y en el cálculo de predicados de primer orden. El razonamiento lógico se basa en la deducción. El lenguaje declarativo por excelencia es **Prolog**.

Lenguajes orientados a objetos

El paradigma orientado a objetos se asocia con el proceso de programación llamado **programación orientada a objetos (POO)** consistente en un enfoque totalmente distinto al proceso procedimental. El enfoque orientado a objetos guarda analogía con la vida real. El desarrollo de software **OO** se basa en el diseño y construcción de objetos que se componen a su vez de datos y operaciones que manipulan esos datos. El programador define en primer lugar los objetos del problema y a continuación los datos y operaciones que actuarán sobre esos datos. Las ventajas de la programación orientada a objetos se derivan esencialmente de la estructura modular existente en la vida real y el modo de respuesta de estos módulos u objetos a mensajes o eventos que se producen en cualquier instante.

Los orígenes de la POO se remontan a los **Tipos Abstractos de Datos** como parte constitutiva de una estructura de datos.

C++, lenguaje orientado a objetos, por excelencia, es una extensión del lenguaje C y contiene las tres propiedades más importantes: *encapsulamiento, herencia y polimorfismo*. **Smalltalk** es otro lenguaje orientado a objetos muy potente y de gran impacto en el desarrollo del software orientado a objetos que se ha realizado en las últimas décadas.

Hoy día **Java** y **C#** son herederos directos de C++ y C, y constituyen los lenguajes orientados a objetos más utilizados en la industria del software del siglo XXI. **Visual Basic** y **VB.Net** son otros lenguajes orientados a objetos, no tan potentes como los anteriores pero extremadamente sencillos y fáciles de aprender.



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 **Windows**, **Linux**, en computadoras personales y los lenguajes de programación. Los lenguajes de programación de alto nivel están diseñados para hacer más fácil la escritura de programas que los lenguajes de bajo nivel. Existen numerosos lenguajes de programación,

cada uno de los cuales tiene sus propias características y funcionalidades, y normalmente son más fáciles de transportar a máquinas diferentes que los escritos en lenguajes de bajo nivel.

Los programas escritos en lenguaje de alto nivel deben ser traducidos por un compilador antes de que se puedan ejecutar en una máquina específica. En la mayoría de los lenguajes de programación se requiere un compilador para cada máquina en la que se desea ejecutar programas escritos en un lenguaje específico...

Los lenguajes de programación se clasifican en:

- *Alto nivel*: Pascal, FORTRAN, Visual Basic, C, Ada, Modula2, C++, Java, Delphi, C#, etcétera.
 - *Bajo nivel*: Ensamblador.
 - *Máquina*: Código máquina.
 - *Diseño de Web*: SMGL, HTML, XML, PHP, Phyton, JavaScript, AJAX...
- Los programas traductores de lenguajes son:
- *Compiladores*.
 - *Intérpretes*.



Algoritmos, programas y metodología de la programación

Contenido

- 2.1 Resolución de problemas con computadoras: fases
- 2.2 Algoritmo: concepto y propiedades
- 2.3 Diseño de algoritmos
- 2.4 Escritura de algoritmos
- 2.5 Representación gráfica de los algoritmos
- 2.6 Metodología de la programación

2.7 Herramientas de programación

- › Resumen
- › Ejercicios
- › Actividades de aprendizaje
- › Actividades complementarias

Introducción

En este capítulo se introduce la metodología que hay que seguir para la resolución de problemas con computadoras.

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 a las herramientas que permiten “dialogar” al usuario con la máquina: *los lenguajes de programación*.

2.1 Resolución de problemas con computadoras: fases

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución. 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.*

Las características más sobresalientes de la resolución de problemas 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 y se obtiene un programa fuente que se compila a continuación.
- **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 fase (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.

Las fases de compilación y ejecución traducen y ejecutan 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, se considera el concepto y significado de la palabra algoritmo.

La palabra **algoritmo** se deriva de la traducción al latín de la palabra *Al-Khôwârizmi*, 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.

¿Sabía que...?

Características de un algoritmo

- *preciso* (indica 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 para su uso con lenguajes estructurados.

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.

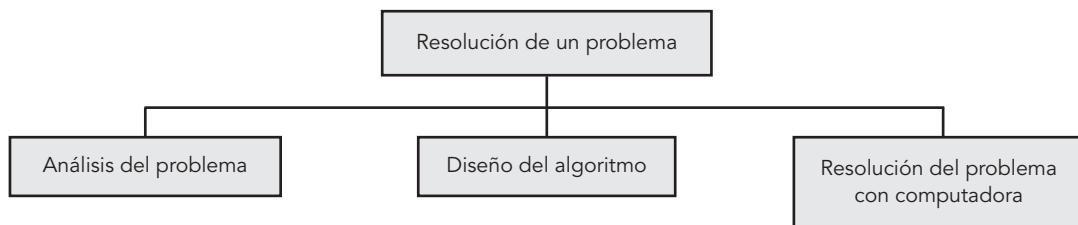


Figura 2.1 Análisis del problema.

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

- ¿Qué entradas se requieren? (tipo de datos con los cuales se trabaja y cantidad).
- ¿Cuál es la salida deseada? (tipo de datos de los resultados y cantidad).
- ¿Qué método produce la salida deseada?
- Requisitos o requerimientos adicionales y restricciones a la solución.



Problema 2.1

Se desea obtener una tabla con las depreciaciones acumuladas y los valores reales de cada año, de un automóvil comprado por 20 000 dólares en el año 2005, durante los seis años siguientes suponiendo un valor de recuperación o rescate de 2000. Realizar el análisis del problema, conociendo la fórmula de la depreciación anual constante D para cada año de vida útil.

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

Entrada $\begin{cases} \text{costo original} \\ \text{vida útil} \\ \text{valor de recuperación} \end{cases}$

$$D = \frac{20\,000 - 2\,000}{6} = \frac{18\,000}{6} = 3\,000$$

Salida $\begin{cases} \text{depreciación acumulada en cada año} \\ \text{depreciación anual por año} \\ \text{valor del automóvil en cada año} \end{cases}$

La tabla siguiente muestra la salida solicitada:

Tabla 2.1 / Análisis del problema.

Año	Depreciación	Depreciación acumulada	Valor anual
1 (2006)	3 000	3 000	17 000
2 (2007)	3 000	6 000	14 000
3 (2008)	3 000	9 000	11 000
4 (2009)	3 000	12 000	8 000
5 (2010)	3 000	15 000	5 000
6 (2011)	3 000	18 000	2 000

Diseño del problema

En la etapa de análisis del proceso de programación se determina *qué* hace el programa. En la etapa de diseño se determina *cómo* hace el programa la tarea solicitada. Los métodos más eficaces para el proceso

de diseño se basan en el conocido *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividiendo 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. Se dice que los programas estructurados de esta forma tienen un diseño modular y el método de dividir 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.

Herramientas gráficas y alfanuméricas

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

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 (figura 2.3). En la figura 2.4 se representa el diagrama de flujo que resuelve el problema 2.1.

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*.

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 español. Las palabras reservadas básicas se representarán en letras negritas minúsculas. Estas palabras son traducción libre de palabras reservadas de lenguajes como C. Más adelante se indicarán los pseudocódigos fundamentales para utilizar en esta obra.

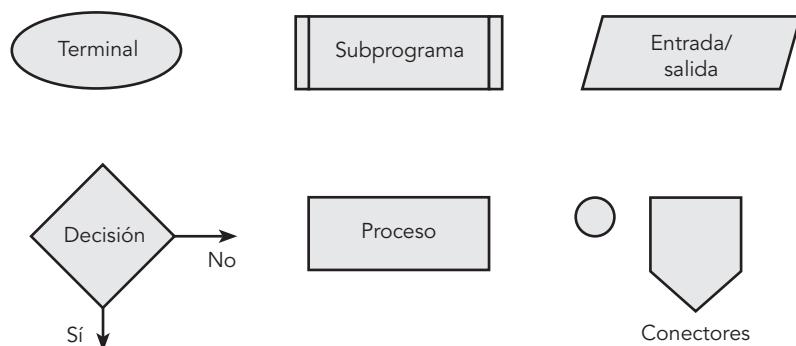


Figura 2.2 Símbolos más utilizados en los diagramas de flujo.

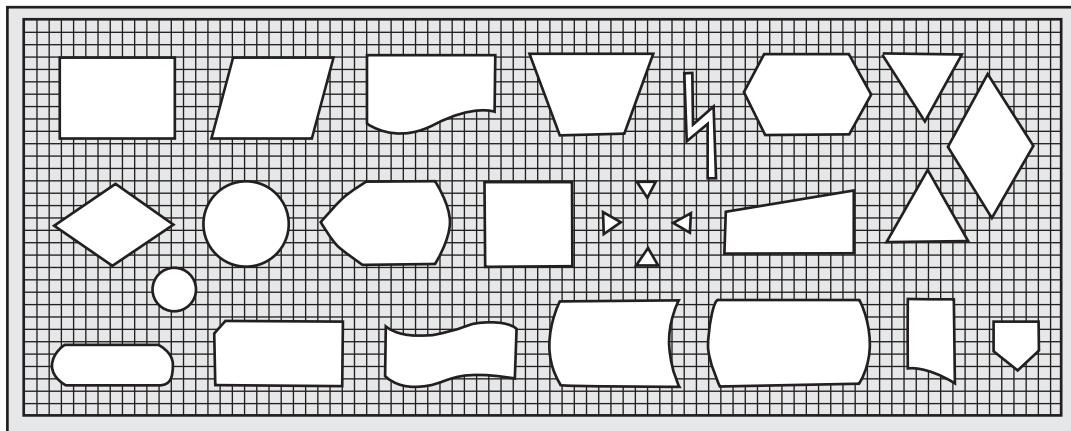


Figura 2.3 Plantilla para dibujo de diagramas de flujo.

El pseudocódigo que resuelve el problema 2.1 es:

```

Previsiones de depreciación
Introducir costo
    vida útil
    valor final de rescate (recuperación)
imprimir cabeceras
Establecer el valor inicial del año
Calcular depreciación
mientras año <= vida útil hacer
    imprimir una línea 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



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

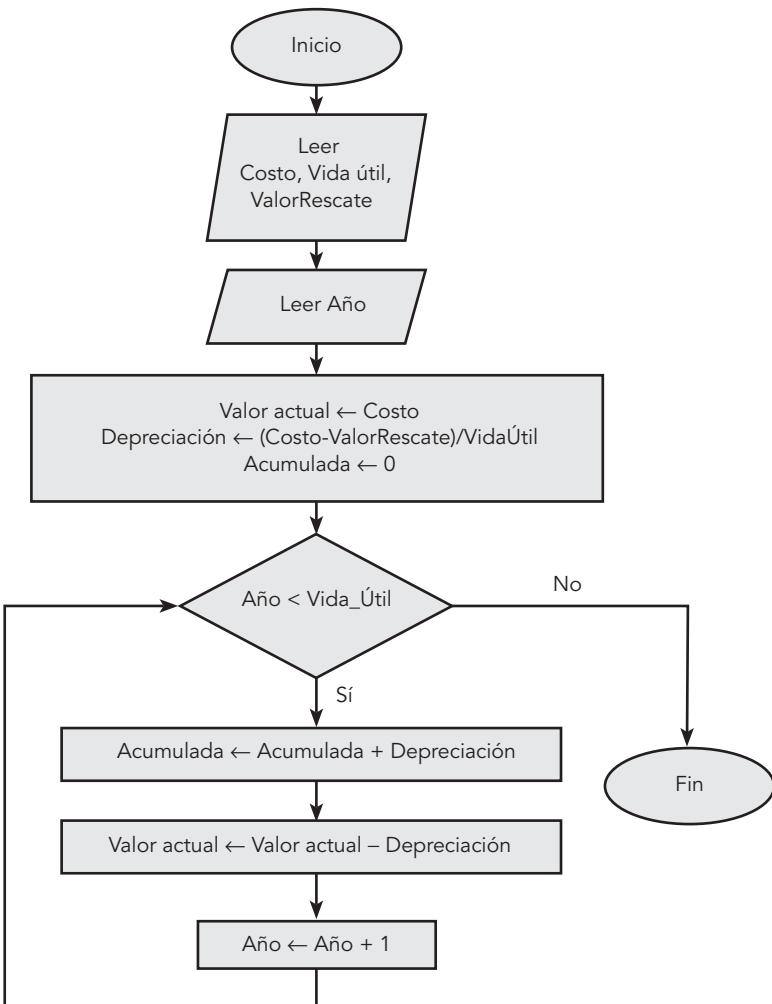


Figura 2.4 Diagrama de flujo (algoritmo primero).

Codificación de un programa

La *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 por el lenguaje de programación correspondiente.

```
{Este programa obtiene una tabla de depreciaciones
acumuladas y valores reales de cada año de un
determinado producto}
```

```
Algoritmo primero
real:  Costo, Depreciación,
       Valor_Recuperacion
       Valor_Actual,
       Acumulado
       Valor_Anual;
entero: Año, Vida_Util;
```

```

    inicio
        escribir('introduzca costo, valor recuperación y vida útil')
        leer (Costo, Valor_Recuperacion, Vida_Util)
        escribir('Introduzca año actual')
        leer (Año)
        Valor_Actual ← Costo;
        Depreciacion ← (Costo-Valor_Recuperacion) /Vida_Util
        Acumulado ← 0
        escribir('Año Depreciación Dep. Acumulada')
        mientras(Año < Vida_Util)
            Acumulado ← Acumulado + Depreciacion
            Valor_Actual ← Valor_Actual - Depreciacion
            escribir(Año, Depreciacion, Acumulado)
            Año ← Año + 1;
        fin mientras
    fin

```

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 solo 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 (1Gb a 4 Gb de memoria central mínima en computadoras personales) no es necesario recurrir a técnicas de ahorro de memoria, por lo que es recomendable que se incluya el mayor número de comentarios posibles, pero eso sí, que sean significativos.

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. Después 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 bibliotecas 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.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con solo 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.

Verificación y depuración de un programa

La *verificación* o depuración 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 o no 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.

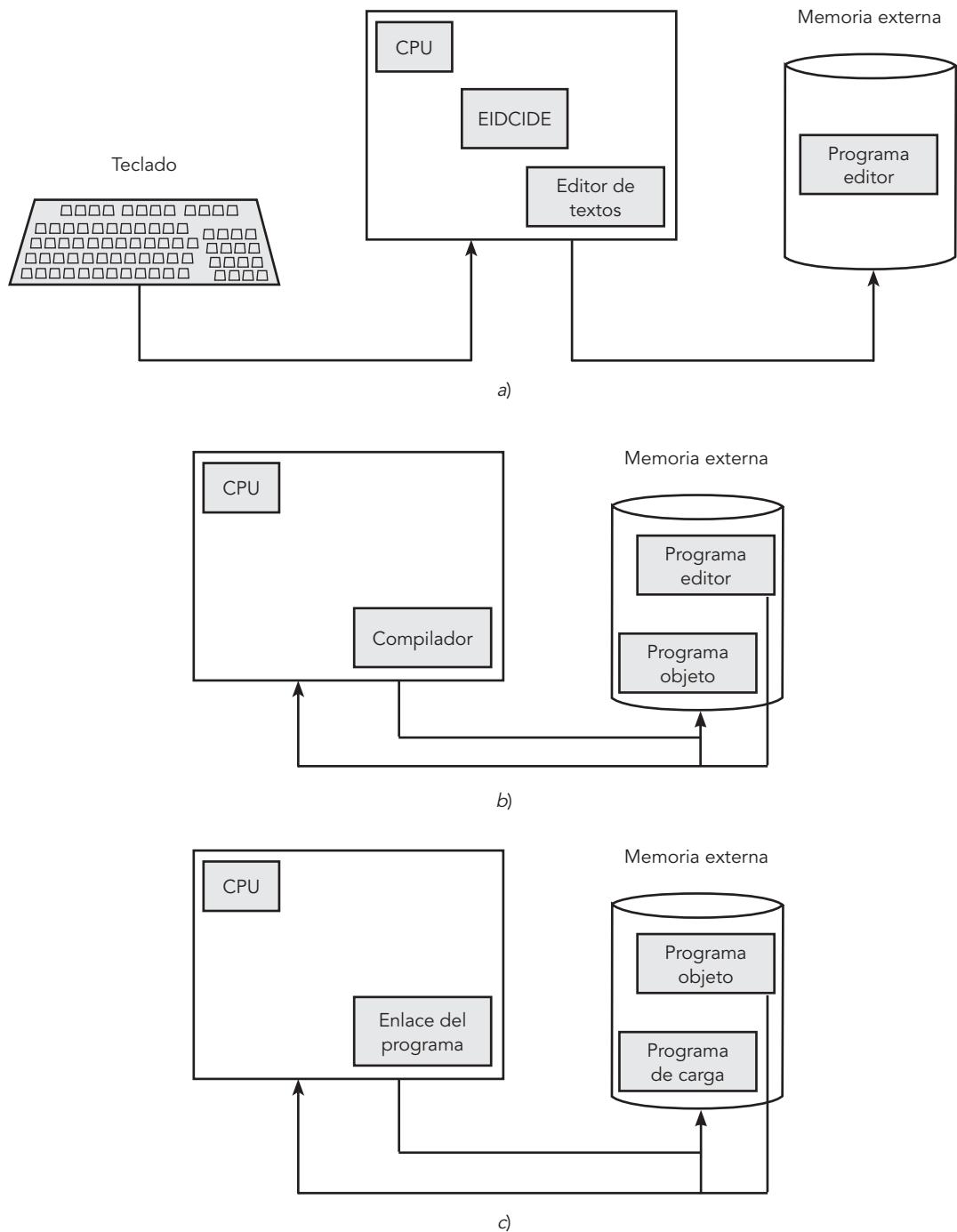


Figura 2.5 Procesos de la compilación/ejecución de un programa: a) edición; b) compilación; c) montaje o enlace.

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 entre 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 solo puede advertirse 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.

Mantenimiento y documentación

La *documentación de un problema* consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Programas deficientemente 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 solo se varía el segundo dígito [**2.0**, **2.1**,...].)

2.2 Algoritmo: concepto y propiedades

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, como se comentó anteriormente, de Mohammed Al-Khôwârizmi, 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.), quien inventó un método para encontrar el máximo común divisor de dos números, se considera junto con Al-Khôwârizmi 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 solo 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.

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.



Figura 2.6 Resolución de un problema.

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 solo un medio para expresar un algoritmo y una computadora es solo 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.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Debe ser *preciso*.
- Debe tener un orden.

Entrada: Ingredientes y utensilios empleados.

Proceso: Elaboración de la receta en la cocina.

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

Un cliente ejecuta un pedido a 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, lo rechazará. Redactar el algoritmo correspondiente.

Ejemplo 2.3

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.

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

Un número es primo si *solo* puede dividirse entre sí mismo y entre 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 entre números distintos a ellos mismos y a la unidad. Así, 9 es divisible entre 3, 8 lo es entre 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número entre 2, 3, 4..., etcétera.

Ejemplo 2.4

1. Inicio.
2. Poner X igual a 2 ($X \leftarrow 2$, X variable que representa a los divisores del número que se busca N).
3. Dividir N entre X (N/X).
4. Si el resultado de N/X es entero, entonces N no es número primo y se bifurca 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 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.
4. $X \leftarrow 2 + 1$, luego $X = 3$.
5. Como X no es 131, se continúa el proceso.
6. $131/X$ resultado no es entero.
7. $X \leftarrow 3 + 1$, $X = 4$.
8. Como X no es 131 se continúa el proceso.
9. $131/X$. . ., etc.
10. Fin.



Ejemplo 2.5

Realizar la suma de todos los números pares entre 2 y 1 000.

El problema consiste en sumar $2 + 4 + 6 + 8 \dots + 1\,000$. Utilizaremos las palabras **SUMA** y **NÚMERO** (*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 **NÚMERO** a 2.
4. Sumar **NÚMERO** a **SUMA**. El resultado será el nuevo valor de la suma (**SUMA**).
5. Incrementar **NÚMERO** en 2 unidades.
6. Si **NÚMERO** < 1 000 bifurcar al paso 4; en caso contrario, escribir el último valor de **SUMA** y terminar el proceso.
7. Fin.

2.3 Diseño de algoritmos

Una computadora no tiene capacidad para solucionar problemas mas que cuando se le proporcionan los sucesivos pasos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya conocemos, el *algoritmo*.

La información proporcionada al algoritmo constituye su *entrada* y la información producida por el algoritmo constituye su *salida*.

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se dividen en subproblemas que sean más fáciles de solucionar que el original. Es el método de *divide y vencerás* (*divide and conquer*), mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o *subproblemas* (figura 2.7).

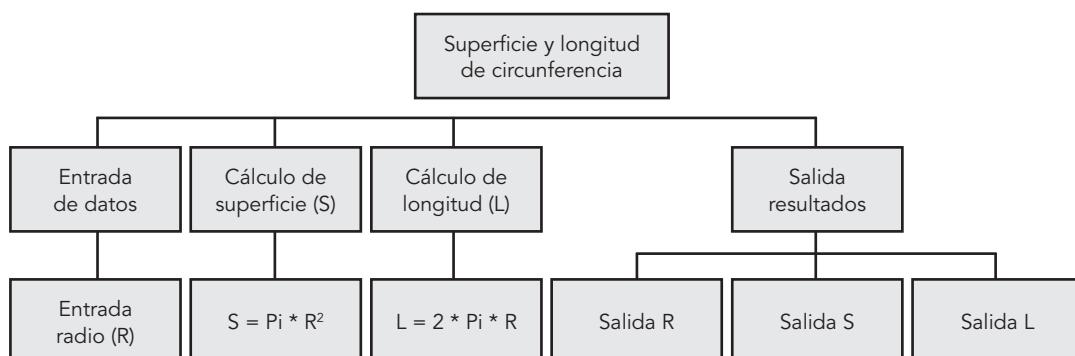


Figura 2.7 Refinamiento de un algoritmo.

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina *diseño descendente (top-down design)*. Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán solo unos pocos pasos (un máximo de doce pasos aproximadamente). Tras esta primera descripción, estos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina *refinamiento del algoritmo (stepwise refinement)*. Para problemas complejos se necesitan con frecuencia diferentes niveles de refinamiento antes de que se pueda obtener un algoritmo claro, preciso y completo.

El problema de cálculo de la circunferencia y superficie de un círculo se puede descomponer en subproblemas más simples: 1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados (datos de salida).

Subproblema	Refinamiento
leer radio	leer radio
calcular superficie	superficie = 3.141592 * radio ^ 2
calcular circunferencia	circunferencia = 2 * 3.141592 * radio
escribir resultados	escribir radio, circunferencia, superficie

Las *ventajas* más importantes del diseño descendente son:

- El problema se comprende más fácilmente al dividirse en partes más simples denominadas *módulos*.
- Las modificaciones en los módulos son más fáciles.
- La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (diseño descendente y refinamiento por pasos) es preciso representar el algoritmo mediante una determinada herramienta de programación: *diagrama de flujo*, *pseudocódigo* o *diagrama N-S*. Así pues, el diseño del algoritmo se descompone en las fases recogidas en la figura 2.8.

2.4 Escritura de algoritmos

Como ya se ha comentado anteriormente, el sistema para describir (“escribir”) un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

- deben ir seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente,
- solo puede ejecutarse una operación a la vez.

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

```
ir al cine
comprar una entrada (boleto o ticket)
ver la película
regresar a casa
```

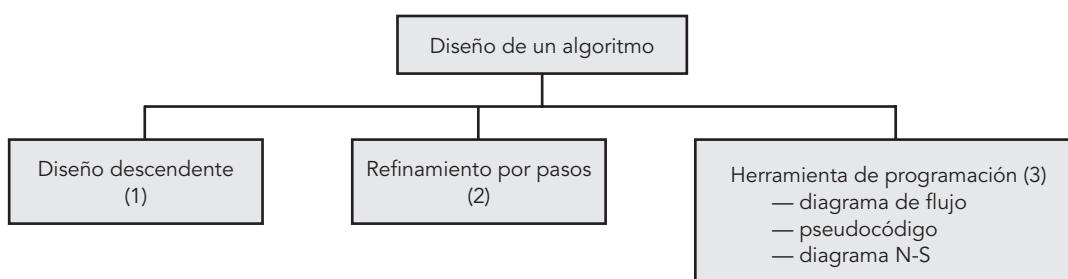


Figura 2.8 Fases del diseño de un algoritmo.

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado *refinamiento sucesivo*, ya que cada acción puede descomponerse a su vez en otras acciones simples. Así, por ejemplo, un primer refinamiento del algoritmo ir al cine se puede describir de la forma siguiente:

```

1. inicio
2. ver la cartelera de cines en el periódico
3. si no proyectan "Harry Potter" entonces
   3.1. decidir otra actividad
   3.2. bifurcar al paso 7
   si_no
   3.3. ir al cine
   fin_si
4. si hay cola entonces
   4.1. formarse en ella
   4.2. mientras haya personas delante hacer
      4.2.1. avanzar en la cola
   fin_mientras
   fin_si
5. si hay localidades entonces
   5.1. comprar un boleto
   5.2. pasar a la sala
   5.3. localizar la(s) butaca(s)
   5.4. mientras proyectan la película hacer
      5.4.1. ver la película
   fin_mientras
   5.5. abandonar el cine
   si_no
   5.6. refunfuñar
   fin_si
6. volver a casa
7. fin

```

En el algoritmo anterior existen diferentes aspectos a considerar. En primer lugar, ciertas palabras reservadas se han escrito deliberadamente en negrita (**mientras**, **si_no**; etc.). Estas palabras describen las estructuras de control fundamentales y procesos de toma de decisión en el algoritmo. Estas incluyen los conceptos importantes de selección (expresadas por **si-entonces-si_no**, *if-then-else*) y de repetición (expresadas con **mientras-hacer** o a veces **repetir-hasta** e **iterar-fin_iterar**, en inglés, *while-do* y *repeat-until*) que se encuentran en casi todos los algoritmos, especialmente en los de proceso de datos. La capacidad de decisión permite seleccionar alternativas de acciones a seguir o bien la repetición una o otra vez de operaciones básicas.

```

si proyectan la película seleccionada ir al cine
si_no ver la televisión, ir al fútbol o leer el periódico

```

Otro aspecto a considerar es el método elegido para describir los algoritmos: empleo de *indentación* (sangrado o justificación) en escritura de algoritmos. En la actualidad es tan importante la escritura de programa como su posterior lectura. Ello se facilita con la indentación de las acciones interiores a las estructuras fundamentales citadas: selectivas y repetitivas. A lo largo de todo el libro la indentación o sangrado de los algoritmos será norma constante.

```

Localizar la(s) butaca(s).
1. inicio //algoritmo para encontrar la butaca del espectador
2. caminar hasta llegar a la primera fila de butacas
3. repetir
   hasta_que número de fila igual número fila boleto
   compara número de fila con número impreso en boleto
4. mientras número de butaca no coincide con número de boleto

```

```

    hacer avanzar a través de la fila a la siguiente butaca
    fin_mientras
5. sentarse en la butaca
6. fin

```

En este algoritmo la repetición se ha mostrado de dos modos, utilizando ambas notaciones, **repetir...** **hasta_que** y **mientras... fin_mientras**. Se ha considerado también, como ocurre normalmente, que el número del asiento y fila coincide con el número y fila rotulado en el boleto.

2.5 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 de manera indistinta en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica 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. *lenguaje de especificación de algoritmos: pseudocódigo*,
3. *lenguaje español, inglés...*
4. *fórmulas*.

Los métodos anteriores no suelen ser fáciles de transformar en programas. Una descripción en español narrativo no es satisfactoria, ya que es demasiado prolífica y generalmente ambigua. Una fórmula, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado: $ax^2 + bx + c = 0$) son un medio sucinto de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

1. *Elevar al cuadrado b*.
2. *Tomar a; multiplicar por c; multiplicar por 4*.
3. *Restar el resultado obtenido de 2 del resultado de 1, etcétera*.

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

Pseudocódigo

El **pseudocódigo** es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL, Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un *primer borrador*, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La *ventaja del pseudocódigo* es que en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica, una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes estructurados como Pascal, C, FORTRAN 77/90, C++, Java, C#, etcétera.

El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés, similares a sus homónimas en los lenguajes de programación, como **start**, **end**, **stop**, **if-then-else**, **while-end**, **repeat-until**, etc. La escritura de pseudocódigo exige normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas.

La representación en pseudocódigo del diagrama de flujo es la siguiente:

```

start
// cálculo de impuestos y salario

```

```

read nombre, horas, precio
salario ← horas * precio
tasas ← 0,25 * salario
salario_neto ← salario - tasas
write nombre, salario, tasas, salario_neto

```

La línea precedida por // se denomina *comentario*. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, solo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes o llaves.

No es recomendable el uso de apóstrofos o simples comillas como representan algunos lenguajes primitivos los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.

```

inicio
  //arranque matinal de un coche
  introducir la llave de contacto
  tirar del estrangulador del aire
  girar la llave de contacto
  pisar el acelerador
  oír el ruido del motor
  pisar de nuevo el acelerador
  esperar unos instantes a que se caliente el motor
  llevar el estrangulador de aire a su posición
fin

```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como **inicio**, **fin**, **parada**, **leer**, **escribir**, **si-entonces-si_no**, **mientras**, **fin_mientras**, **repetir**, **hasta_que**, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación. En esta obra, al igual que en otras nuestras, utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, con el objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

En consecuencia, en los pseudocódigos citados anteriormente, se deberían sustituir las palabras **start**, **end**, **read** y **write**, por **inicio**, **fin**, **leer** y **escribir**, respectivamente.

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 2.2 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 2.9 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 25% sobre el salario bruto.

Los símbolos estándar normalizados por **ANSI** (American National Standard Institute) son muy variados. En la figura 2.3 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:

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

El diagrama de flujo de la figura 2.9 resume sus características:

- existe una caja etiquetada “**inicio**”, que es de tipo elíptico,
- existe una caja etiquetada “**fin**” de igual forma que la anterior,
- existen cajas rectangulares de proceso: rectángulo y rombo.

Tabla 2.2 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, etcétera).
	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 Sí o NO).
	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 ordinograma 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 organigrama situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	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).

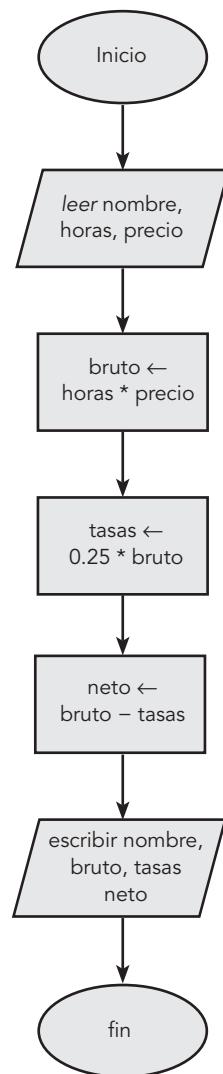


Figura 2.9 Diagrama de flujo.

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.

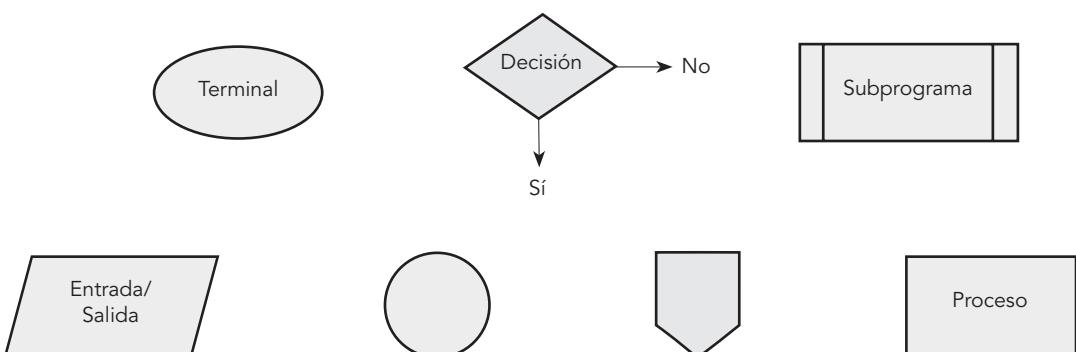


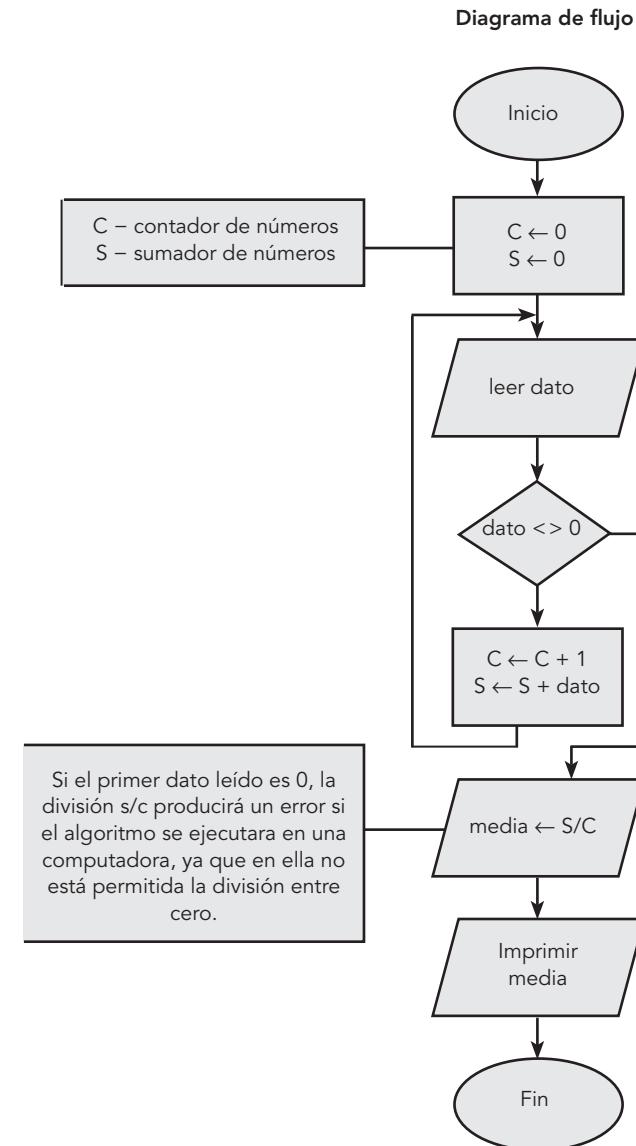
Figura 2.10 Símbolos básicos de diagramas de flujo (véase figura 2.3).

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.

Ejemplo 2.6

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 números C y variable sumas.
2. Leer un número.
3. si el número leído es cero:
 - calcular la media;
 - imprimir la media;
 - calcular la suma;
 - incrementar en uno el contador de números;
 - ir al paso 2.
4. Fin.



Pseudocódigo

```

entero: dato, C
real: Media, S

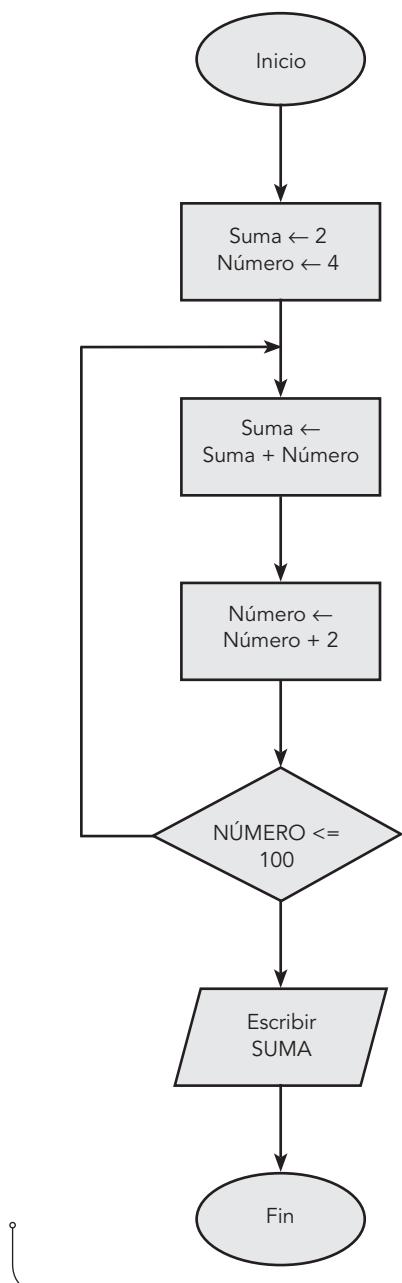
C ← 0
S ← 0
escribir ('Datos numéricos;
para finalizar se introduce 0')

repetir
  leer(dato)
  si dato <> 0 entonces
    C ← C + 1
    S ← S + dato
  fin si
hasta dato = 0

{Calcula la media y la escribe}
si (C > 0) entonces
  Media ← S/C
  escribir (Media)
fin si
  
```


Ejemplo 2.7

Suma de los números pares comprendidos entre 2 y 100.

Diagrama de flujo

Pseudocódigo

```

entero: numero, Suma
Suma ← 2
numero ← 4
mientras (numero <= 100) hacer
    suma ← suma + numero
    numero ← numero + 2
fin mientras
escribe ('Suma pares entre 2 y 100 =', suma)
  
```

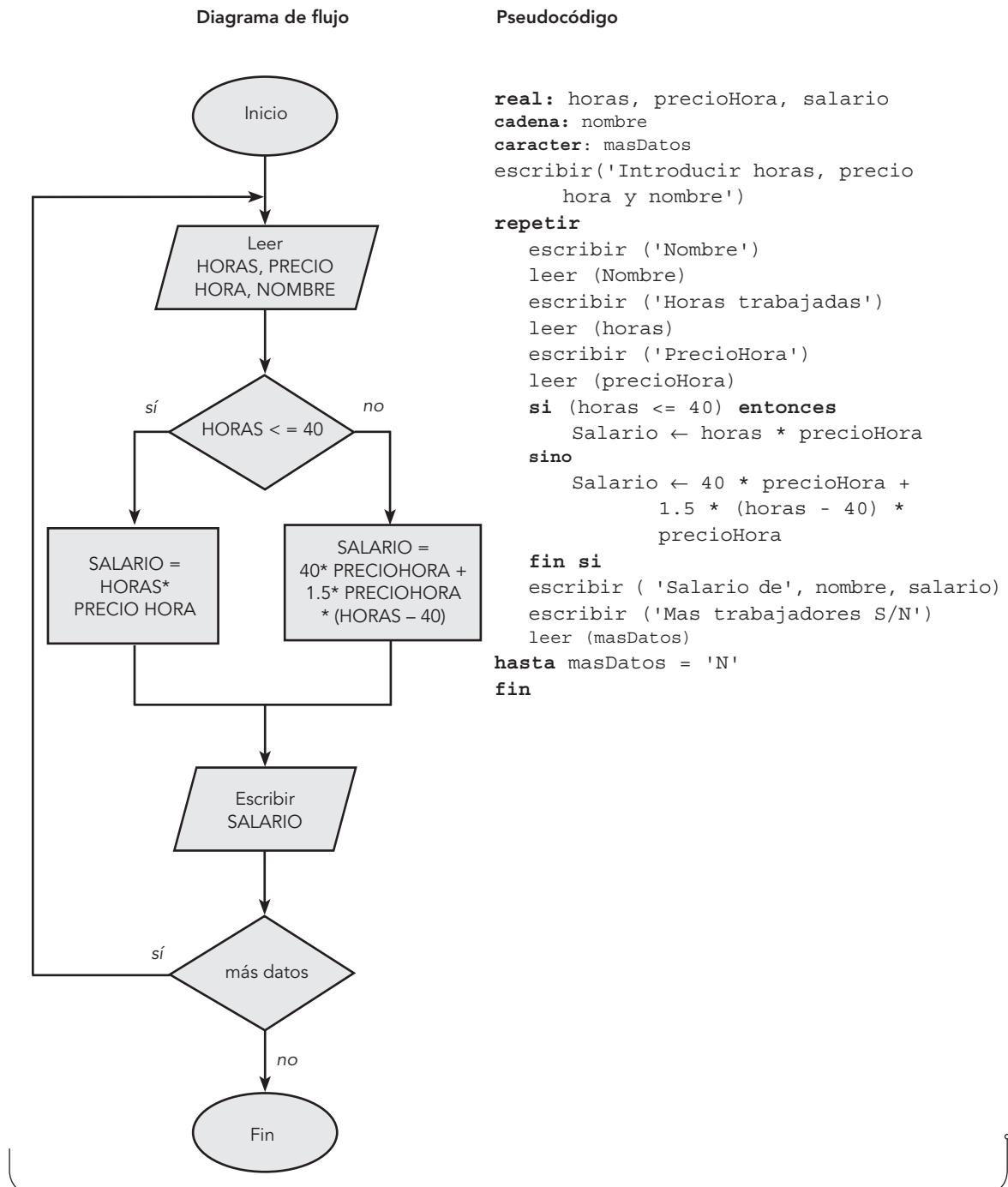

Ejemplo 2.8

Se desea realizar el algoritmo que resuelva el siguiente problema: Cálculo de los salarios mensuales de los empleados de una empresa, sabiendo que estos se estiman con base en las horas semanales trabajadas y de acuerdo con un precio específico por horas. Si se pasan de 40 horas semanales, las horas extraordinarias se pagarán a razón de 1.5 veces la hora ordinaria.

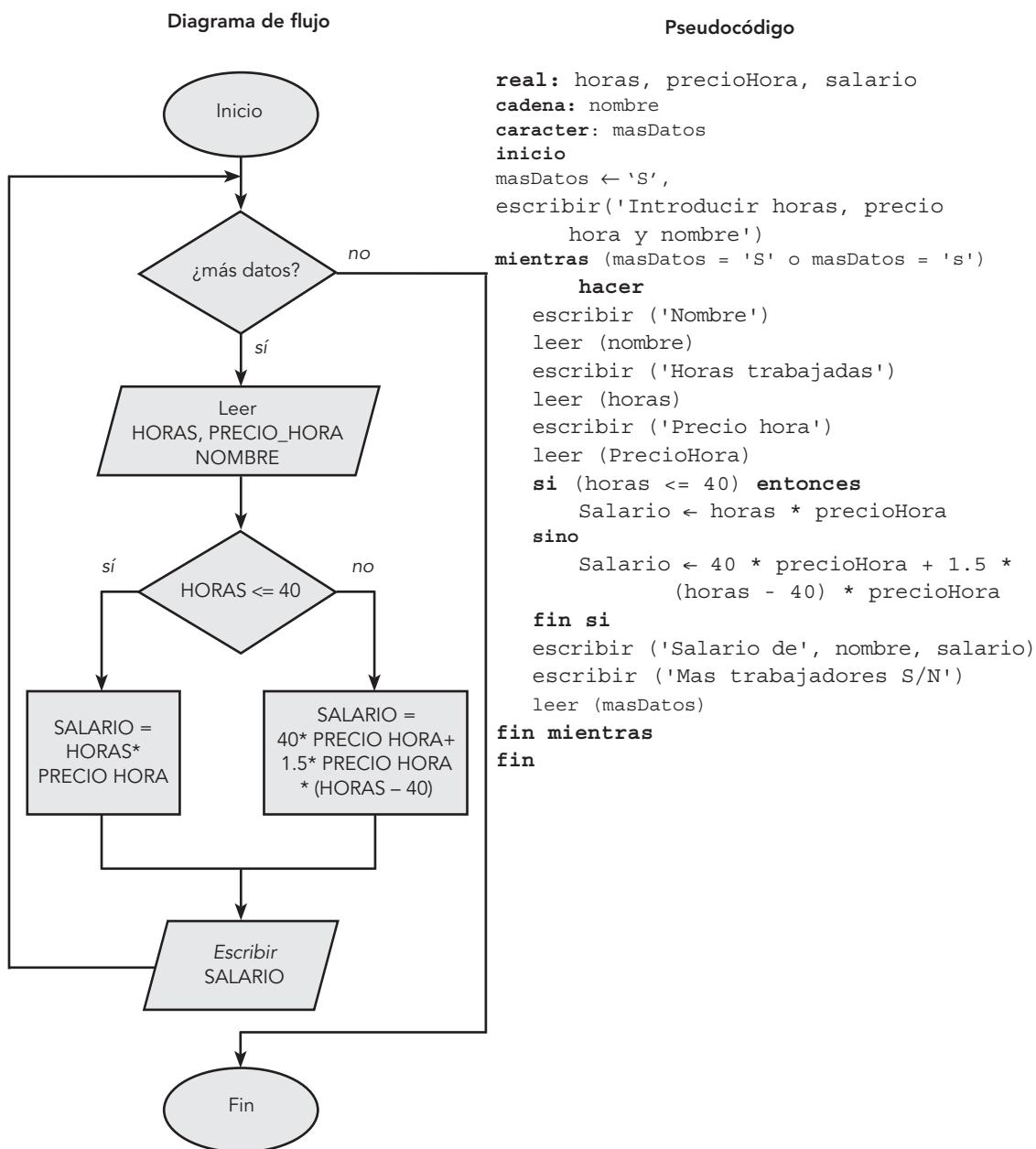
Los cálculos son:

1. Leer datos del archivo de la empresa, hasta que se encuentre la ficha final del archivo (HORAS, PRECIO_HORA, NOMBRE).
2. Si HORAS <= 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 y la codificación en pseudocódigo se indican a continuación:



Una variante también válida del diagrama de flujo anterior es:



Ejemplo 2.9

La escritura de algoritmos para realizar operaciones sencillas de conteo es una de las primeras cosas que una computadora puede aprender.

Supongamos que se proporciona una secuencia de números, como

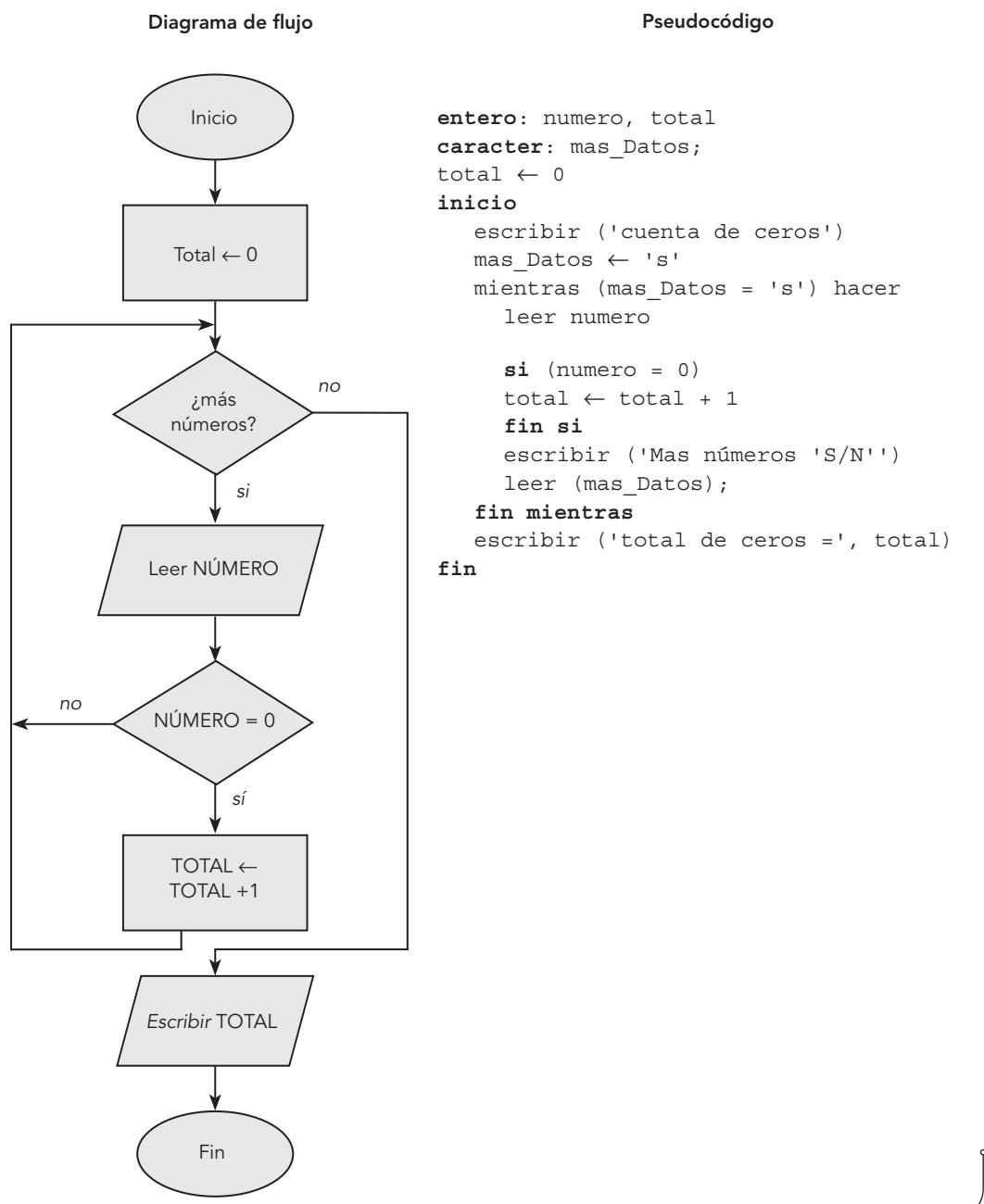
5 3 0 2 4 4 0 0 2 3 6 0 2

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

El algoritmo es muy sencillo, ya que solo 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 números, imprimir el valor de TOTAL y fin.
4. Si existen más números, ejecutar los pasos 5 a 8.
5. Leer el siguiente número 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 y la codificación en pseudocódigo correspondiente es:



**Ejemplo 2.10**

Dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. si se cumple la condición, escribir “iguales” y en caso contrario, “distintos”.

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:

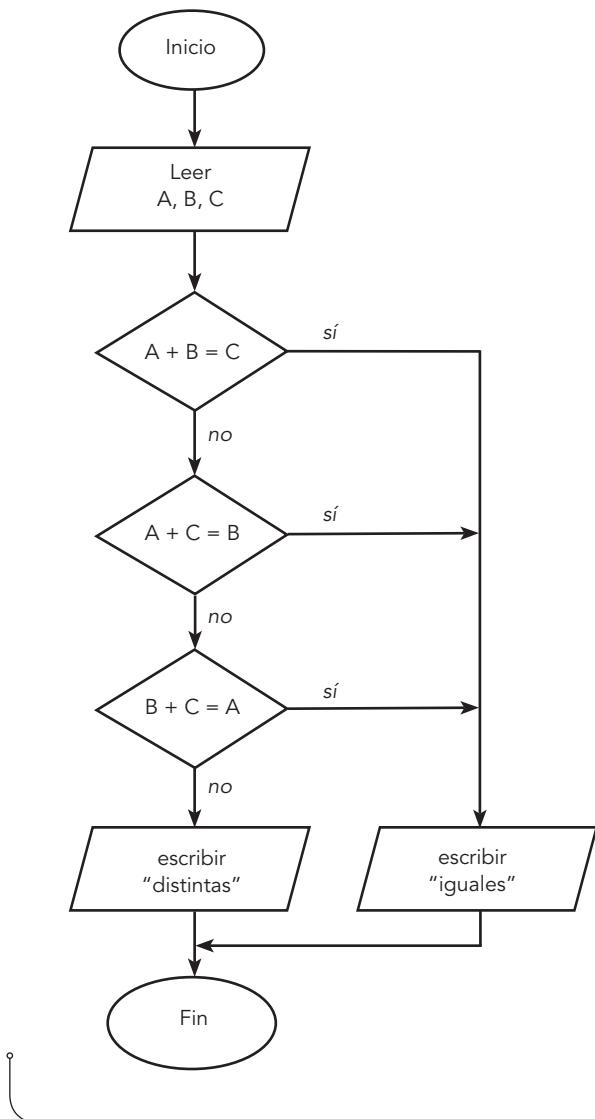
2 3 4

el resultado sería “Distintas”. El algoritmo es

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 y la codificación en pseudocódigo correspondiente es:

Diagrama de flujo



Pseudocódigo

```

entero: a, b, c
inicio
leer (a,b,c)

si (a + b = c) entonces
  escribir ("Iguales")
  sino si (a + c = b) entonces
    escribir ("Iguales")
  sino si (b + c = a) entonces
    escribir ("Iguales")
  sino
    escribir ("Distintas")
  fin si
fin si
fin si
fin
  
```

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).

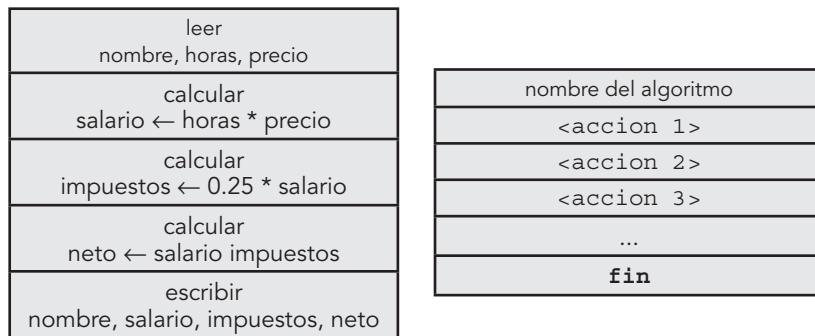


Figura 2.11 Representación gráfica N-S de un algoritmo.

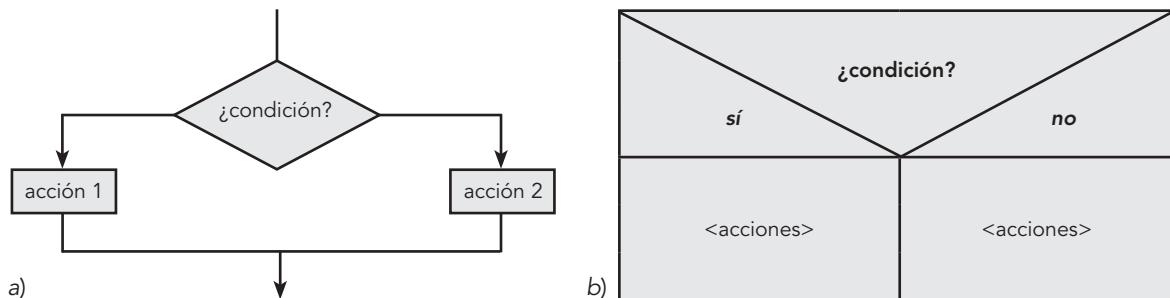


Figura 2.12 Estructura condicional o selectiva: a) diagrama de flujo; b) diagrama N-S.

Escriba un algoritmo que lea el nombre de un empleado, las horas trabajadas, el precio por hora y calcule los impuestos a pagar (tasa = 25%) y el salario neto.

El diagrama N-S correspondiente es la figura 2.11.

Ejemplo 2.11

Se desea calcular el salario neto semanal de un trabajador (en dólares) 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) los primeros 1 000 dólares son libres de impuestos,
 - b) los siguientes 400 dólares tienen un 25% de impuestos,
 - c) los restantes, un 45% de impuestos,
- la tarifa horaria es 15 dólares.

Ejemplo 2.12

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

2.6 Metodología de la programación

Existen dos enfoques muy populares para el diseño y construcción de programas: el *enfoque estructurado* y el *enfoque orientado a objetos*. Estos dos enfoques conducen a dos tipos o metodologías de programación: **programación estructurada** y **programación orientada a objetos**.¹ Un tercer enfoque, la **programación modular** está directamente conectado con los otros dos enfoques.

Programación modular

La **programación modular** es uno de los métodos de diseño más flexible y potente para mejorar la productividad de un programa. En programación modular el programa se divide en *módulos* (partes independientes), cada uno de los cuales ejecuta una única actividad o tarea y se codifican independientemente de otros módulos. Cada uno de estos módulos se analiza, codifica y pone 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, este deberá romperse en otros módulos más pequeños. El proceso sucesivo de subdivisión de módulos continúa hasta que cada módulo tenga solamente una tarea específica qué ejecutar. Esta tarea puede ser *entrada, salida, manipulación de datos, control de otros módulos* o alguna *combinación de estos*. Un módulo puede transferir temporalmente (*bifurcar*) el control a otro módulo; sin embargo, cada módulo debe a la larga 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.

Dado que los módulos son independientes, diferentes programadores pueden trabajar simultáneamente en distintas 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)*. Cada módulo se diseña 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.

Programación estructurada

La **programación estructurada** utiliza las técnicas tradicionales del campo de programación y que data de las décadas de 1960 y 1970, especialmente desde la creación del lenguaje Pascal por Niklaus Wirth. La programación estructurada es un enfoque específico que, normalmente, produce programas bien escritos y muy legibles, aunque no necesariamente un programa bien escrito y fácil de leer ha de ser estructurado. La programación estructurada trata de escribir un programa de acuerdo con unas reglas y un conjunto de técnicas.

Las reglas de programación estructurada o diseño estructurado se basan en la modularización; es decir, en la división de un problema en subproblemas más pequeños (módulos), que a su vez se pueden dividir en otros subproblemas. Cada subproblema (módulo) se analiza y se obtiene una solución para ese subproblema. En otras palabras, la programación estructurada implica un diseño descendente.

Una vez que se han resuelto los diferentes subproblemas o módulos se combinan todos ellos para resolver el problema global. El proceso de implementar un diseño estructurado se denomina programación estructurada. El diseño estructurado también se conoce como diseño descendente (*topdown*), diseño

¹ En la obra *Programación en C++*. Un enfoque práctico, de los profesores Luis Joyanes y Lucas Sánchez, Madrid: McGrawHill, 2006, puede encontrar un capítulo completo (capítulo 1) donde se analiza y comparan con detalle ambos tipos de métodos de programación.

ascendente (*bottomup*) o refinamiento sucesivo y programación modular. La descomposición de un problema en subproblemas también conocidos como módulos, funciones o subprogramas (métodos en programación orientada a objetos) se conoce también como programación modular, ya estudiada anteriormente, y uno de los métodos tradicionales para la resolución de los diferentes subproblemas e integración en un único programa global es la programación estructurada.

C, Pascal y FORTRAN, y lenguajes similares, se conocen como lenguajes procedimentales (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir entre cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias.

En el caso de pequeños programas, estos principios de organización (denominados *paradigmas*) se demuestran eficientes. El programador solo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y esta, a su vez, ejecuta estas instrucciones. Cuando los programas se vuelven más grandes, cosa que lógicamente sucede a medida que aumenta la complejidad del problema a resolver, la lista de instrucciones se incrementa considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de funciones (métodos, procedimientos, subprogramas o subrutinas según la terminología de lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de dividir el programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas módulos (normalmente, en el caso de C, denominadas *archivos* o *ficheros*); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se vuelven más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resulta muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son estas: primero, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo deficiente del mundo real.

Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos: locales y globales. *Datos locales* que están ocultos en el interior de la función y son utilizados, de manera exclusiva, por la función. Estos datos locales están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Otro tipo de datos son los *datos globales* a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que estos datos sean globales.

Un programa grande se compone de numerosas funciones y datos globales y ello conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil conceptualizar la estructura del programa. En segundo lugar, el programa es difícil de modificar ya que cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales pueden no ser aceptadas por todas o algunas de las funciones.

Técnicas de programación estructurada

Las técnicas de programación estructurada incluyen construcciones o estructuras (instrucciones) básicas de control.

- **Secuencia.**
- **Decisión** (también denominada *selección*).
- **Bucles o lazos** (también denominada *repetición* o *iteración*).

Las estructuras básicas de control especifican el orden en que se ejecutan las distintas instrucciones de un algoritmo o programa. Una construcción (estructura, instrucción o sentencia en la jerga de lenguaje)

jes de programación) es un bloque de instrucciones de un lenguaje y una de las operaciones fundamentales del lenguaje.

Normalmente la ejecución de las sentencias o instrucciones de un programa o subprograma, se realiza una después de otra en orden secuencial. Este procedimiento se llama ejecución secuencial. Existen, no obstante, diferentes sentencias que especifican cómo saltar el orden secuencial, es decir, que la sentencia a ejecutar sea distinta de la siguiente en la secuencia. Esta acción se denomina transferencia de control o control del flujo del programa. Los primeros lenguajes de programación tenían entre las sentencias de control del flujo una denominada *goto* (*ir_a*) “que permitía especificar una transferencia de control a un amplio rango de destinos de un programa y poco a poco se fue abandonando por los muchos problemas que conllevaba un control eficiente (en su tiempo a este tipo de programación se denominó “código espagueti” a aquellos programas en los que se usaba esta sentencia).

En la década de 1970 nació la ya citada tendencia de programación denominada programación estructurada que preconizaba la no utilización de la sentencia *goto* en los programas y su sustitución por otras sentencias de transferencia de control debido al gran daño que suponía a la eficiencia de los programas. Böhm y Jacopini demostraron que los programas se podían escribir sin sentencias *goto*.² Sin embargo, fue también en la citada década de 1970 con la aparición de lenguajes de programación como Pascal y C, en el que la tendencia se hizo una realidad y prácticamente se desechó el uso de la sentencia *goto*.

Böhm y Jacopini demostraron también que todos los programas pueden ser escritos solo con tres estructuras de control: secuencial, de selección y de repetición. En terminología de lenguaje las estructuras de control se denominan *estructuras de control*.

La estructura secuencial ejecuta las sentencias en el orden en que están escritas o se señala expresamente. La estructura de selección se implementa en uno de los tres formatos siguientes:

Sentencia *if* (*si*): selección única

Sentencia *if-else* (*si-entonces-sino*): selección doble

Sentencia *switch* (*según_sea*): selección múltiple

La estructura de repetición se implementa en tres formatos diferentes

Sentencia *while* (*mientras*)

Sentencia *do-while* (*hacer-mientras*)

Sentencia *for* (*desde/para*)

La *programación estructurada* promueve el uso de las tres sentencias de control:

Secuencia

Selección (sentencias, *if*, *if-else*, *switch*)

Repetición (sentencias *while*, *dowhile*, *for*)

Programación orientada a objetos

La **Programación orientada a objetos (POO) (OOP, Object Oriented Programming)** es el paradigma de programación dominante en la actualidad y ha sustituido las técnicas de programación estructurada comentada antes. **Java** es totalmente orientado a objetos y es importante que usted esté familiarizado con la POO para hacer a Java más productivo y eficiente. **C++** tiene carácter híbrido ya que posee las características de orientación a objetos y también las características estructuradas.

La *programación orientada a objetos* se compone de *objetos*. Un **objeto** es un elemento autosuficiente de un programa de computadora que representa un grupo de características relacionadas entre sí y está diseñado para realizar una tarea específica. Cada objeto tiene una funcionalidad específica que se expone a sus usuarios y una implementación oculta al usuario. Muchos objetos se obtienen de una biblioteca y otros se diseñan a la medida. Es decir, la programación orientada a objetos trabaja dentro de un mismo principio: un programa trabaja con objetos creados para una finalidad específica y otros objetos existen creados de modo estándar. Cada objeto sirve para un rol específico en el programa global.

Para problemas pequeños, la división en pequeños subproblemas puede funcionar bien. En el caso de problemas grandes o complejos, los objetos funcionan mejor. Consideremos el caso de una aplicación

² Böhm C. y Jacopini, D. “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”. *Communications of the ACM*, vol. 9, núm. 5, may 1966, pp. 366371.

web típica como un navegador web o un buscador también de la web. Supongamos, por ejemplo, que la aplicación requiere 3 000 módulos (funciones o procedimientos, en la jerga estructurada) que manipulan todos los datos globales del problema. Si el problema se resuelve con programación orientada a objetos, se podrían crear solo 100 clases (la plantilla que permite crear objetos) de modo que cada clase contuviese 30 procedimientos o funciones (métodos en orientación a objetos). Este sistema facilita el diseño y construcción de los programas ya que es fácil seguir la traza del problema o localizar errores entre los 30 métodos de cada clase en lugar de en 3 000.

En el **diseño orientado a objetos (OO)** el primer paso en el proceso de resolución de problemas es identificar los componentes denominados objetos que forman el soporte de la solución y determinar cómo interactúan estos objetos entre sí.

Los objetos constan de datos y operaciones que se realizan sobre esos datos. Un objeto combina en una única entidad o componente, datos y operaciones (funciones/procedimientos en programación estructurada y métodos en programación orientada a objetos). De acuerdo con este planteamiento, antes de diseñar y utilizar objetos se necesitará aprender cómo representar los datos en la memoria de la computadora, manipular los datos y el modo de implementar las operaciones.

La creación de operaciones requiere la escritura de los correspondientes algoritmos y su implementación en un lenguaje orientado a objetos como C++ o Java. En el enfoque orientado a objetos las múltiples operaciones necesarias de un programa utilizan métodos para implementar los algoritmos. Estos algoritmos utilizarán instrucciones o sentencias de control, de selección, repetitivas o iterativas.

El paso siguiente para trabajar con objetos requiere encapsular en una única unidad los datos y operaciones que manipulan esos datos. En Java (y en otros lenguajes orientados a objetos como C++ y C#) el mecanismo que permite combinar datos y operaciones sobre esos datos en una unidad se denomina clase. Una **clase** es una plantilla o modelo que permite crear objetos a partir de la misma.

El OO funciona bien combinado con el diseño estructurado. En cualquier caso ambos enfoques requieren el dominio de los componentes básicos de un lenguaje de programación. Por esta razón explicaremos, en detalle, en los siguientes capítulos los componentes básicos de C++ y Java, que utilizaremos para el diseño orientado a objetos y el diseño estructurado cuando sea necesario. De cualquier forma, en el diseño orientado a objetos nos centraremos en la selección e implementación de clases (objetos abstractos o plantillas generadoras de objetos) y no en el diseño de algoritmos. Siempre que sea posible se deseará reutilizar las clases existentes, o utilizarlas como componentes de nuevas clases, o modificarlas para crear nuevas clases.

Este enfoque permite a los programadores (diseñadores de software) utilizar clases como componentes autónomos para diseñar y construir nuevos sistemas de software al estilo de los diseñadores de hardware que utilizan circuitos electrónicos y circuitos integrados para diseñar y construir nuevas computadoras.

2.7 Herramientas de programación

La programación en C, C++, Java o cualquier otro lenguaje de programación requiere herramientas para la creación de un programa. Las herramientas más usuales son **editor, compilador y depurador de errores** y puesta a punto del programa, aunque existen otras herramientas, sobre todo en el caso de desarrollo profesional. Estas herramientas pueden ser independientes y utilizadas de esta forma o bien estar incluidas en entornos de desarrollo integradas y utilizadas como un todo. En el aprendizaje profesional se recomienda conocer ambas categorías de herramientas y a medida que las vaya dominando seleccionar cuáles considera las más idóneas para su trayectoria profesional.

Editores de texto

Un editor de textos es un programa de aplicación que permite escribir programas. Los editores que sirven para la escritura de programas en lenguajes de alto nivel son diferentes de los procesadores de texto tradicionales como Word de Microsoft, Google Docs o Zoho.

Un editor de software ya sea independiente o integrado es un entorno de desarrollo que normalmente debe proporcionar las características adecuadas para la adaptación de las normas de escritura de la sintaxis del lenguaje de programación correspondiente y también algunas propiedades relacionadas con la sintaxis de este lenguaje; que reconozca sangrados de línea y que reconozca y complete automáticamente

mente palabras clave (reservadas) del lenguaje después de que los programadores hayan tecleado los primeros caracteres de la palabra.

Un editor de texto clásico es NotePad que permite crear (escribir) un programa en Java o C++ siguiendo las reglas o sintaxis del lenguaje; otro editor típico es Edit (Edit.com) del sistema operativo MSDOS. El editor debe escribir el programa fuente siguiendo las reglas de sintaxis del lenguaje de programación y luego guardarlo en una unidad de almacenamiento como archivo de texto. Así, por ejemplo, en el caso de Java un programa que se desea llamar `MiPrimerPrograma` (el nombre de una clase en Java) se debe guardar después de escribir en un archivo de texto denominado `MiPrimerPrograma.java` (nombre de la clase; una característica específica de Java). Antiguamente se utilizaban editores como Emacs, JEdit o TextPad.

Programa ejecutable

El programa o archivo ejecutable es el archivo binario (código máquina) cuyo contenido es interpretado por la computadora como un programa. El ejecutable contiene instrucciones en código máquina de un procesador específico, en los casos de lenguajes como C o C++, ya que se requiere un compilador diferente para cada tipo de CPU, o bien *bytecode* en Java, que es el código máquina que se produce cuando se compila un programa fuente Java y que es el lenguaje máquina de la máquina virtual (JVM), que es independiente de cualquier tipo de CPU. En el ejemplo anterior y en el caso de Java el código traducido por el compilador viene en *bytecode* y se almacena en el compilador con el nombre `MiPrimerPrograma.class`.

El intérprete Java traduce cada instrucción en *bytecode* en un tipo específico de lenguaje máquina de la CPU y a continuación ejecuta la instrucción (desde un punto de vista práctico el compilador una vez que ha obtenido el código máquina *bytecode*, utiliza un cargador que es un programa que recibe a su vez las funciones o clases correspondientes de una biblioteca Java y la salida alimenta al intérprete que va ejecutando las sucesivas instrucciones). En el caso de Java se requiere un tipo diferente de intérprete para cada procesador o CPU específica. Sin embargo, los intérpretes son programas más sencillos que los compiladores, pero como el intérprete Java traduce las instrucciones en *bytecodes* una detrás de otras, el programa Java se ejecuta más lentamente.

Los programas ejecutables pueden ser portables (se pueden ejecutar en diferentes plataformas) o no portables (están destinados a una plataforma concreta).

Proceso de compilación/ejecución de un programa

Las computadoras solo entienden el lenguaje máquina. Por consiguiente, para ejecutar un programa con éxito, el código fuente (el programa escrito en un lenguaje de programación C/C++, Java o C#) o programa fuente, se debe traducir al lenguaje máquina o de la máquina, mediante un compilador o en su caso un intérprete. El proceso de un programa escrito en un lenguaje de programación de alto nivel consta de cinco etapas: editar, compilar, enlazar, cargar y ejecutar, aunque según el tipo de lenguaje de programación C/C++ o Java alguna de las etapas puede descomponerse en otras etapas.

Existen dos métodos para procesar programas completos (compilación y ejecución). Uno son los programas de consola, normalmente conocidos como consola de línea de comandos, que son herramientas en las que los programadores deben teclear los comandos (las órdenes) en una consola o ventana Shell y ejecutar paso a paso las diferentes etapas de compilación. El segundo método es el más utilizado ya que suele ser más cómodo de usar: son los EDI (Integrated Development Environment), entorno de desarrollo integrado, que permiten la edición, compilación y ejecución de un programa de modo directo.

Aunque los EDI son más fáciles de aprender puede resultar más tedioso su uso para el caso de programas pequeños, por lo que le aconsejamos que aprenda a manejar ambos métodos y aunque lo más fácil sea casi siempre el EDI, habrá ocasiones en que puede resultarle mejor el uso de la consola de línea de comandos para aprendizaje o incluso para desarrollos muy profesionales.

Consola de línea de comandos

La compilación con la consola de línea de comandos es el procedimiento más antiguo y clásico de la compilación/ejecución de un programa fuente. Con este método se edita el programa fuente con un editor

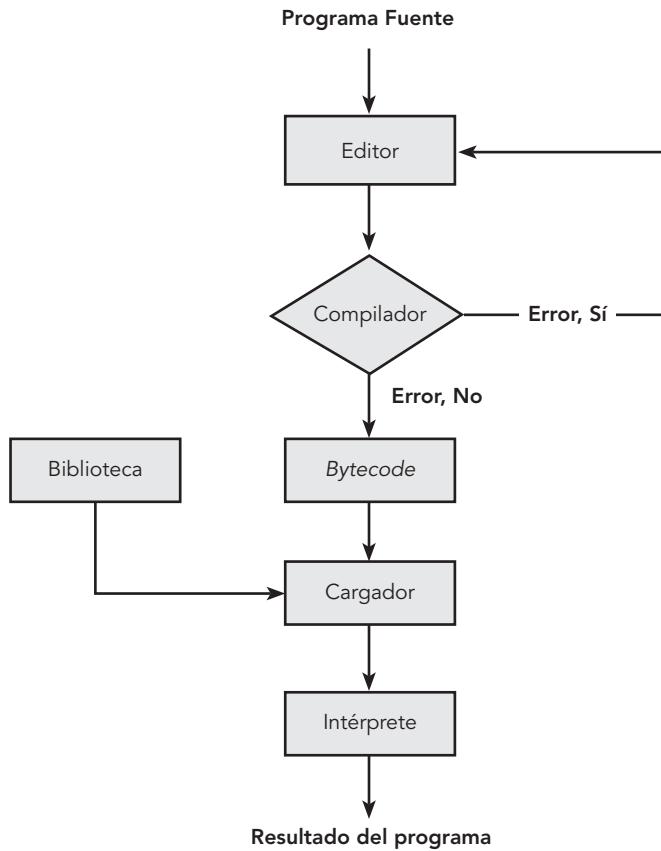


Figura 2.13 Proceso de ejecución de un programa en Java.

de archivos de texto como, por ejemplo, el bloc de notas del sistema operativo Windows (Notepad.exe) o el Edit (Edit.com) del sistema operativo MSDOS; una vez editado el programa fuente con el editor se compila el programa con el compilador. El editor y el compilador se ejecutan directamente desde la línea de comandos del sistema operativo en la ventana de comandos del sistema operativo.

En la fase de aprendizaje de programación se suele recomendar, en la etapa inicial, el uso de la edición, compilación y ejecución desde la ventana o consola de la línea de comandos, de modo que una vez adquirida la destreza y aprendido el mecanismo completo del proceso de ejecución, entonces se puede pasar a trabajar con un entorno de desarrollo integrado profesional.

En los últimos años han aparecido entornos de desarrollo profesionales excelentes y en muchos casos gratuitos, como es el caso de Eclipse y de NetBeans, con la gran ventaja de que en muchas ocasiones sirven para diferentes lenguajes de programación.

Entorno de desarrollo integrado

El entorno de desarrollo integrado (EDI) contiene herramientas que soportan el proceso de desarrollo de software. Se compone de un editor para escribir y editar programas, un **compilador**, un **depurador** para detectar errores lógicos (errores que originan una ejecución no correcta del programa) y un constructor de interfaz gráfica de usuario (GUI). Además suelen incluir herramientas para compilar e interpretar, en su caso, los programas fuente. Los EDI pueden estar orientados a uno o varios lenguajes de programación aunque generalmente están orientados a uno solo. Existen entornos de desarrollo para casi todos los lenguajes de programación, como C, C++, Python, Java, C#, Delphi, Visual Basic, Pascal, ObjectiveC (el lenguaje de desarrollo de aplicaciones de Apple para teléfonos inteligentes iPhone), etcétera.

Entornos de desarrollo integrados populares	
BlueJ	(www.bluej.org)
NetBeans	(www.netbeans.org)
JBuilder	(www.borland.com)
Eclipse	(www.eclipse.org)
JCreator	(www.jcreator.com)
JEdit	(www.jedit.org)
JGrasp	(www.jgrasp.org)
Dev-C++	(http://www.bloodshed.net/devcpp.html)
Microsoft Visual C++	(http://www.microsoft.com)

Algunos entornos de desarrollo integrados populares son: 1) En C++: Microsoft Visual Studio 2010, DevC++, NetBeans, Eclipse; 2) En Java: Eclipse, NetBeans, JBuilder, JCreator, JGrasp, BlueJ y Java Development Kit (JDK). Cualquiera de estos entornos de desarrollo le será a usted de gran utilidad, por lo que le aconsejamos que descargue e instale el entorno elegido o aquel recomendado por su profesor o maestro en clase.

Resumen

La *metodología necesaria para resolver problemas mediante programas* se denomina **metodología de la programación**. Las etapas generales para la resolución de un problema son:

1. *Análisis del programa*
2. *Diseño del algoritmo*
3. *Codificación del programa*
4. *Compilación y ejecución*
5. *Verificación y depuración*
6. *Mantenimiento y documentación*

Las herramientas más utilizadas en el diseño y escritura de algoritmos son: diagramas de flujo, pseudocódigos y diagramas N-S.

Un **algoritmo** es un método para la resolución de un problema paso a paso. Los métodos de programación más utilizados son: *programación modular*, *programación estructurada* y *programación orientada a objetos*. Las herramientas más utilizadas en el diseño y construcción de programas son: *editores*, *compiladores* y *depuradores*.

Ejercicios

- 2.1 Diseñar una solución para resolver cada uno de los siguientes problemas y tratar de refinar sus soluciones mediante algoritmos adecuados:
 - a) Realizar una llamada telefónica desde un teléfono público.
 - b) Cocinar unos huevos revueltos con papas.
 - c) Arreglar un pinchazo de una bicicleta.
 - d) Freír un huevo.
- 2.2 Escribir un algoritmo para:
 - a) Sumar dos números enteros.
 - b) Restar dos números enteros.
 - c) Multiplicar dos números enteros.
 - d) Dividir un número entero entre otro.
- 2.3 Escribir un algoritmo para determinar el máximo común divisor (MCD) de dos números enteros por el algoritmo de Euclides:
 - Dividir el mayor de los dos enteros positivos entre el más pequeño.
 - A continuación dividir el divisor entre el resto.
 - Continuar el proceso de dividir el último divisor entre el último resto hasta que la división sea exacta.
 - El último divisor es el MCD.

- 2.4 Diseñar un algoritmo que lea y visualice una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe visualizar. Visualizar el número de valores leídos.
- 2.5 Diseñar un algoritmo que visualice y sume la serie de números 3, 6, 9, 12..., 99.
- 2.6 Escribir un algoritmo que lea cuatro números y a continuación visualice el mayor de los cuatro.
- 2.7 Diseñar un algoritmo que lea tres números y descubra si uno de ellos es la suma de los otros dos.
- 2.8 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 visualizar 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.
- 2.9 Diseñar un algoritmo para determinar si un número N es primo. (Un número primo solo puede ser divisible entre él mismo y entre 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. Por ejemplo, "Mortimer" contiene dos "m", una "o", dos "r", una "i", una "t" y una "e".
- 2.15 Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente según las premisas siguientes. Un capital de 1 000 dólares, con una tasa de interés de 6%, renta un interés en un día de 0.06 multiplicado por 1 000 y dividido entre 365. Esta operación producirá 0.16 dólares de interés y el capital acumulado será de 1 000.16. El interés para el segundo día se calculará multiplicando 0.06 por 1 000 y dividiendo el resultado entre 365. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcular el capital total acumulado al final del periodo de tiempo especificado.



Actividades de aprendizaje

- 2.1 Reconozca los conceptos básicos de programa, programación, paradigmas de programación utilizando herramientas teóricas y prácticas.
- 2.2 Conozca el entorno de un lenguaje de programación: entorno de desarrollo integrado y consola de línea de comandos para la compilación y ejecución de programas.
- 2.3 Busque y analice información necesaria para instalar y configurar el compilador del lenguaje de programación a utilizar (lenguajes C, C++ y Java).
- 2.4 Practique con editores, compiladores e intérpretes.
- 2.5 Practique con la consola de línea de comandos y entornos de desarrollo integrado de C, C++ y Java.
- 2.6 Explique los conceptos básicos para la formulación de algoritmos, así como sus ventajas y desventajas.
- 2.7 Genere un catálogo de problemas para su análisis y solución.
- 2.8 Resuelva y analice problemas cotidianos.
- 2.9 Investigue los diferentes métodos para representar un algoritmo: diagrama de flujo, N-S (Nassi-Schneiderman).



Actividades complementarias

- 2.1 Imagine una aplicación simple de ingeniería (por ejemplo, resolver un sistema de ecuaciones, cálculo de la superficie de una figura geométrica).
- 2.2 Proponga las etapas para la resolución del problema.
- 2.3 Considere la descomposición del problema en módulos.
- 2.4 Escriba el algoritmo que resuelve el problema en un diagrama de flujo y en un pseudocódigo.
- 2.5 Busque en internet el concepto de programación estructurada y amplíe los conceptos explicados en clase y en el libro.
- 2.6 Busque en internet información sobre programación orientada a objetos y lenguajes de programación orientada a objetos.
- 2.7 Busque y estudie la información encontrada sobre programación modular y funciones.
- 2.8 Escriba un algoritmo para hacer guacamole; primero en texto narrado y después en pseudocódigo y diagrama de flujo.
- 2.9 Escriba un algoritmo para fabricar tequila reposado, primero en texto narrado y después en pseudocódigo y diagrama de flujo.
- 2.10 Escriba un algoritmo para ir al cine o a un concierto de Maná en Veracruz el próximo domingo comprando la entrada en un comercio electrónico de internet (se supone que puede estar en Veracruz o en otra ciudad de México).
- 2.11 Escriba un algoritmo para efectuar una compra de un libro de programación por internet en un comercio electrónico; por ejemplo en la tienda Amazon.
- 2.12 Busque información sobre diagramas N-S con objeto de ampliar los conceptos explicados en el libro y diseñar los algoritmos de “guacamole” y “tequila reposado” y “compra de un libro en Amazon” mediante diagramas N-S.
- 2.13 Busque y descargue alguna herramienta de software libre (preferentemente gratuita) para realización de diagramas de flujo. Una vez que la tenga instalada trate de hacer los gráficos de los algoritmos 5, 6 y 7, comparando con las realizadas por usted mismo manualmente.
- 2.14 Trate de buscar herramientas para la realización de diagramas N-S.

PARTE

II

Programación en C





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
- 3.5 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
 - › Resumen
 - › 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.

3.1 Estructura general de un programa en C

En este apartado se repasan 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>
int main()
{
    ...
}
```

archivo de cabecera stdio.h
cabecera de función
nombre de la función
sentencias

#include	Directivas del preprocesador
#define	Macros del preprocesador

Declaraciones globales

- prototipos de funciones
- variables

Función principal main
main()
{
 declaraciones locales
 sentencias
}

Definiciones de otras funciones
tipo1 func1(...)
{
 ...
}

Conceptos clave

- › Archivo de cabecera
- › Código ejecutable
- › Código fuente
- › Código objeto
- › Comentarios
- › Constantes
- › char
- › Directiva #include
- › float,double
- › Función main()
- › Identificador
- › int
- › Preprocesador
- › printf()
- › scanf()
- › Variables

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(); int main()
- funciones definidas por el usuario;
- comentarios del programa

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(). Observe 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 requiere los paréntesis `()` a continuación de `main()`.

La cuarta y séptima línea contienen solo 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*. Al poner 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*).

Observe el punto y coma `(;)` al final de la quinta y sexta líneas. 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.

¿Sabía que...?

- El programa más corto de C es el «programa vacío» 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.

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 numeral 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, observe 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, 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 cabecera* o *archivos de inclusión*.

Los *archivos de cabecera* (archivos con extensión `.h` que 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 los mismos 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 importa 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 de manera predeterminada los archivos se encuentran en el directorio `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 de manera predeterminada en el directorio, se encierra el archivo de cabecera y el camino entre comillas, como

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

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.

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` de tipo `int`:

```
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 MICONST2 0.75

/* Declaraciones globales */
int Calificaciones;

int main()
{
    ...
}
```

Función main()

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

```
int main()
{
    ... ←————— bloque de sentencias
}
```

Las sentencias incluidas entre las llaves `{...}` se denominan *bloque*. Un programa debe tener solo 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.

¿Sabía que...?

Una *función C* es un subprograma que devuelve un único valor, un conjunto de valores o realiza alguna tarea específica 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.

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, 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 invocada, el código asociado con la función se ejecuta y, a continuación, se retorna a la función llamadora.

Todas las funciones tienen nombre y reciben una lista de valores. 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 trazarcurva();
```

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
{

<i>sentencias</i>	cuerpo de la función
<i>return;</i>	retorno de la función
}	fin de la función
<i>tipo_retorno</i>	tipo de valor, o <code>void</code> , devuelto por la función
<i>nombre_función</i>	nombre de la función
<i>lista_de_parámetros</i>	lista de parámetros, o <code>void</code> , pasados a la función. Se conoce también como <i>argumentos de la función</i> o <i>argumentos formales</i> .

C proporciona también funciones predefinidas que se denominan *funciones de biblioteca*, las cuales son funciones listas para ejecutar que vienen con el lenguaje C. Requieren la inclusión del archivo de cabecera estándar, 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();
    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()`.

```
int main()
{
    ...
}
void obtenerdatos()
{
    ...
}
```

```

    }
    void alfabetizar()
{
    ...
}
...

```

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 solo 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("Un pueblo, JAÉN\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 demoinc1.c ilustra el uso de #include */
#include <stdio.h>
int main()
{
    printf("Luis Joyanes Aguilar\n");
    printf("Avda. de Andalucía, 48\n");
    printf("Un pueblo, JAÉN\n");
    printf("Andalucía, ESPAÑA\n");
    return 0;
}
```

Ejemplo 3.2

El siguiente programa copia un mensaje en un *array* (arreglo) 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 main()
{
    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;
}
```

¿Sabía que...?

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 de `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.

Para crear un programa se utilizan las siguientes etapas:

1. Definir su programa.
2. Definir directivas del preprocesador.
3. Definir declaraciones globales.
4. Crear `main()`.
5. Crear el cuerpo del programa.
6. Crear sus propias funciones.
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 documento de texto 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;
}
```

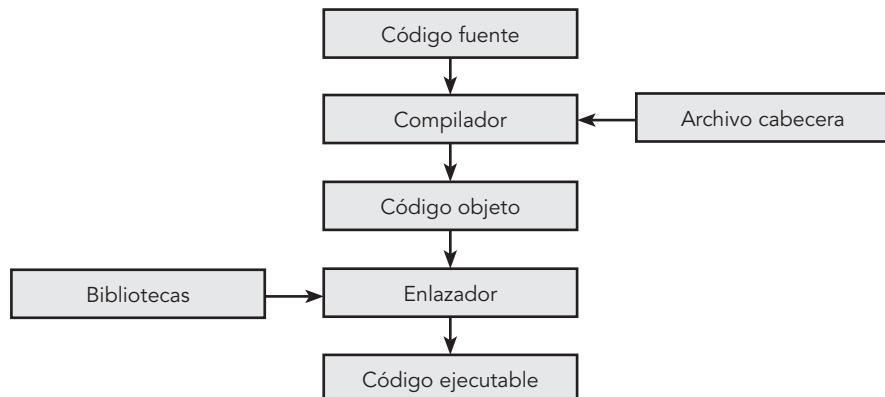


Figura 3.2 Etapas de creación y ejecución de un programa.

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 como `area.c`. Si la ventana del editor le muestra un nombre 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 solo 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 DVD o *pendrive* el programa se guardará de modo permanente, incluso después de apagar la computadora y siempre que esta se vuelva a arrancar.

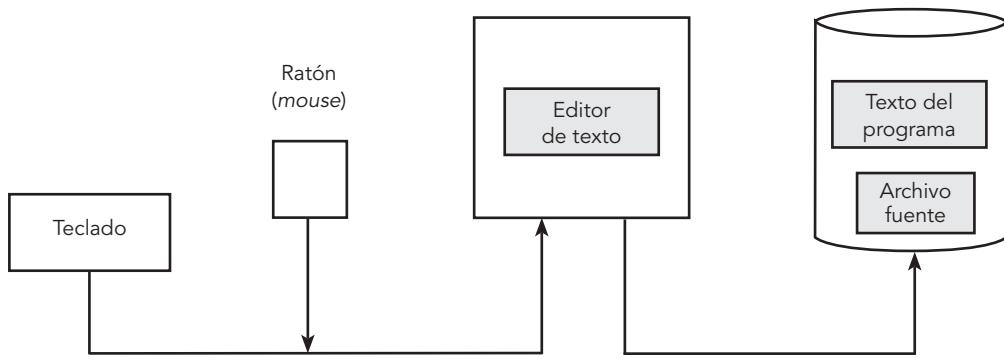


Figura 3.3 Programación modular.

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* (*archivo fuente*). Con la ayuda de un editor de texto se puede editar el texto fácilmente; es decir, cambiar, mover, cortar, pegar y 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.

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 Linux, 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 `área` anterior, se puede almacenar con el nombre `área.obj`.

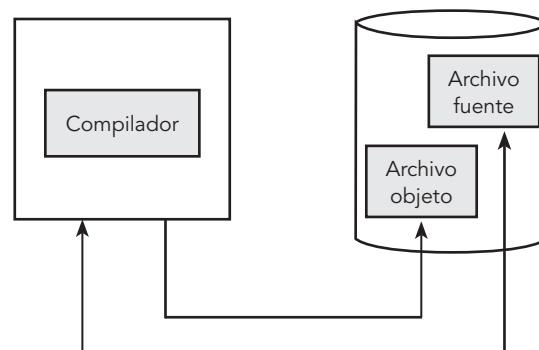


Figura 3.4 Proceso de edición de un archivo fuente.

El archivo objeto contiene solo 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, traducida y lista para utilizar en su programa.

Normalmente un programa consta de diferentes unidades o partes de programa que se han compilado en forma independiente. 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 máquina necesario para ejecutar el programa. Se puede ejecutar el programa escribiendo `area` en el indicador de órdenes o haciendo clic en el ícono del archivo.

Se puede guardar 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 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 3.5 muestra el proceso completo de puesta a punto 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 ya no se presenten errores.

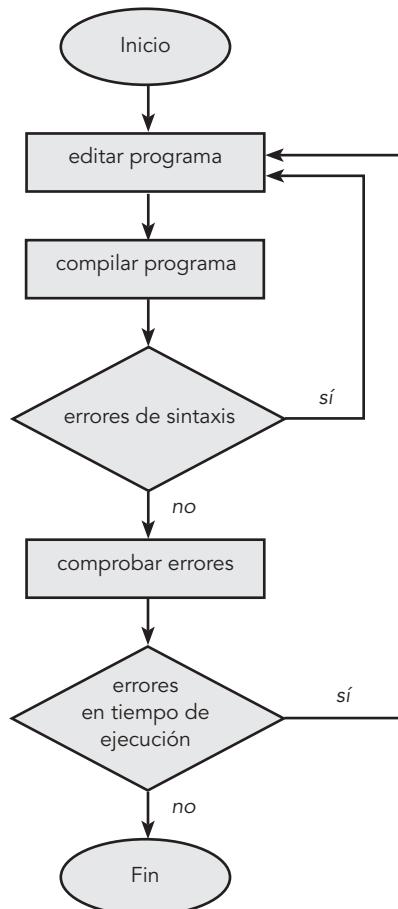


Figura 3.5 Proceso completo de depuración de un programa.

Etapas del proceso

- El código fuente (archivo del programa) se crea con la ayuda del editor de texto.
- El compilador traduce el archivo texto en un archivo objeto.
- El enlazador pone juntos a diferentes archivos objetos para poner un archivo ejecutable.
- 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, en tanto que la detección y aislamiento del error son 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 la 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. 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*.

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 los suele detectar 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* (o *diagnóstico*) 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.
- Etcétera.

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.

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
```

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 lo 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*), 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, verificando 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 este tipo de errores.

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 por completo la exactitud (corrección) para asegurarse que se fija el error que se está tratando y no produce otro error. Los errores de regresión son habituales, pero son fáciles de leer y corregir. Una ley no escrita es que: “un error se ha producido, probablemente, por el último código modificado”.

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 generar, normalmente, por errores de sintaxis y suelen 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.

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, como dividir un número entre 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 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.

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 *solo* 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 solo 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 distintas 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 esta 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 cuántas pruebas se hagan: “las pruebas nunca se terminan, solo 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 solo muestran la presencia de errores, no su ausencia. No se puede probar que un programa es correcto (exacto) solo 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

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

se controla por directivas introducidas por líneas que contienen # como primer carácter. El resultado del preprocessado es una secuencia de *tokens*.

Tokens (elementos léxicos de los programas)

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

Identificadores

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

nombre_clase	Indice	Dia_Mes_Año
elemento_mayor	Cantidad_Total	Fecha_Compra_Casa
a	Habitacion120	i

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 es:

1. Escribir identificadores de variables en letras minúsculas.
2. Escribir constantes en mayúsculas.
3. Escribir 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 solo son significativos los 32 primeros (éste es el caso de Borland y Microsoft).
4. Los identificadores no pueden ser palabras reservadas, como, if, switch o else.

Palabras reservadas

Una palabra reservada (*keyword* o *reserved word*), 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.

asm	enum	signed
auto	extern	sizeof
break	float	static
case	for	struct

```

char      goto      switch
const     if        typedef
continue  int       union
default   long      unsigned
do        register void
double   return   volatile
else     short    while

```

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.

```
/* Titulo: Demo_uno por Mr. Martinez */
```

Otra forma, el comentario en dos líneas:

```
/* Cabecera del programa text_uno
Autor: J.R. Mazinger */
```

Signos de puntuación y separadores

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

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

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

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 * (10 ⁻³⁸) .. 3.4 * (10 ³⁸)
double	0.00045	8	1.7 * (10 ⁻³⁰⁸) .. 1.7 * (10 ³⁰⁸)
long double	1e-8	8	igual que 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*, 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, representa, mediante identificadores, a valores constantes de tipo entero).

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 16 bits) 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
int	-32.768 .. +32.767	Aritmética de enteros, bucles for, conteo.
unsigned int	0 .. 65.535	Conteo, bucles for, índices.
short int	-128 .. +127	Aritmética de enteros, bucles for, 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, este 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_var1>, <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
<code>long</code>	-2147483648 .. 2147483647
<code>unsigned long</code>	0 .. +4294967295

Tipos de coma flotante (float/double)

Los tipos de datos de coma (*punto*) flotante representan números reales que contienen una coma (un punto) decimal, como 3.14159, o números muy grandes, 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 admite 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
<code>float</code>	$3.4 * 10^{-38} \dots 3.4 * 10^{38}$	7 dígitos
<code>double</code>	$1.7 * 10^{-308} \dots 1.7 * 10^{308}$	15 dígitos
<code>long double</code>	$3.4 * 10^{-4932} \dots 1.1 * 10^{4932}$	19 dígitos

Ejemplos

```
float f;           /* definición de la variable f */
f = 5.65;          /* 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 */
```

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 (como texto) utilizando el tipo de dato `char`. En unión con la estructura arreglo (*array*), que se estudiará posteriormente, se puede utilizar para almacenar *cadenas 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 suma con el código ASCII de valor */
```

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 apóstrofo se puede escribir como

'\''

y el carácter nueva línea

'\n'

La tabla 3.6 enumera las diferentes secuencias de escape de C.

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	5C
'\''	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>

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 */
```

¿Sabía que...?

Valor distinto de cero	representa <i>true</i> (verdadero)
0 (cero)	representa <i>false</i> (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.

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 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, como Color, con una secuencia de otros nombres, como Azul, Verde, Rojo y Amarillo.

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 reales,

- constantes caracteres,
- 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), este debe de estar precedido de `0`.

Formato decimal 123

Formato octal 0777 (están precedidas de la cifra 0)

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

Formato hexadecimal 0xFF3A (están precedidas de "0x" o bien "OX")

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 (puede cambiar según el compilador)`

Constantes caracteres

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

`'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.

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.

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 main()
{
    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. Observe 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 de cadena

Una *constante de 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 "\n"

```
"esto es una cadena\
que tiene dos lineas"
```

Se puede concatenar cadenas, escribiendo

```
"ABC" "DEF" "GHI"
"JKL"
```

que equivale a

```
"ABCDEFGHIJKLM"
```

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 como

```
#define NULL 0
```

Recuerde que una constante de caracteres se encierra entre comillas simples (apóstrofo), 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*.

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 las constantes simbólicas NUEVALINEA, PI y VALOR por los valores \n, 3.141592 y 54, respectivamente. 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.

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, Rojo 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 declare:

```
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 11 a Rojo.

Constantes declaradas `const` y `volatile`

El cualificador `const` permite dar nombres simbólicos como en otros lenguajes. El formato general para crear una constante es:

```
const tipo nombre = valor;
```

Si se omite *tipo*, C utiliza *int* (entero en forma predeterminada)

```
const int Meses = 12;           /* Meses es constante simbólica de valor 12 */
const char CARACTER = '@';
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 solo 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 utiliza 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* frente a *#define*

Los valores de los símbolos de *const* ocupan espacio de datos en tiempo de ejecución, mientras que *#define* solo 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*.

¿Sabía que...?

Sintaxis de *const*

```
const tipoDatos nombreConstante = valor Constante;
```

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
```

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 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 de un bloque de código, al principio de una función.

```
/* variable al principio del archivo */
#include <stdio.h>
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 kilómetros */
#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 y y1 en la función main() */
    x = 75;
    y1 = 89;
    if (x > 10)
    {
        int y2 = 50;          /* declara e inicializa la variable y2 en el
                               bloque if */
        y1 = y1+y2;
    }
    printf("x = %d, y1 = %d\n",x,y1);
    return 0;
}

```

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 = '/';
```

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 definen las variables de C, estas se pueden utilizar en la totalidad del programa, dentro de una función o pueden existir solo 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 locales

Las *variables locales* son aquellas definidas en el interior de una función y son visibles solo 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;
}
```

Variables globales

Las *variables globales* son variables que se declaran fuera de la función y son visibles en forma predeterminada 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;
    ...
}
```

¿Sabía que...?

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.

En el siguiente segmento de código C, `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, solo las sentencias interiores a `main` pueden utilizar `A`.

```
#include <stdio.h>
int Q; /* Alcance o ámbito global
          Q, variable global
void main()
{
    int A; /* Local a main
              A, variable local
```

```

A = 124; Q = 1;
{
    int B;           Primer subnivel en main
    B = 124;
    A = 457;
    Q = 2;
    {
        int C;       Subnivel más interno de main
        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 la pantalla de la computadora, 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.

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 trasmisir 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()` es:

```
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 en forma predeterminada a la izquierda en vez del ajuste a la derecha.

```
printf("%15s", "HOLA LUCAS");
printf("%-15s", "HOLA LUCAS");
```

visualizará en pantalla

```
HOLA LUCAS
```

```
HOLA LUCAS
```

Los códigos de formato más utilizados y su significado:

- `%d` El dato se convierte a entero decimal.
- `%o` El dato entero se convierte a octal.
- `%x` El dato entero se convierte a hexadecimal.
- `%u` El dato entero se convierte a entero sin signo.
- `%c` El dato se considera de tipo carácter.
- `%e` El dato se considera de tipo `float`. Se convierte a notación científica, de la forma `{-}n.aaaaaaaaE{+|-}dd`.
- `%f` El dato se considera de tipo `float`. Se convierte a notación decimal, con parte entera y los dígitos de precisión.
- `%g` El dato se considera de tipo `float`. Se convierte según el código `%e` o `%f` dependiendo de cuál sea la representación más corta.
- `%s` El dato ha de ser una cadena de caracteres.
- `%lf` El dato se considera de tipo `double`.

C utiliza *secuencias de escape* para visualizar caracteres que no están representados por símbolos tradicionales, como `\a`, `\b`, etc. Las secuencias de escape clásicas se muestran en la tabla 3.6.

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`).

El listado SECESC.C utiliza secuencias de escape, como emitir sonidos (pitidos) en el terminal dos veces y a continuación presentar dos retrocesos de espacios en blanco.

Ejemplo 3.4

```
/* 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;
}
```

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 sufijo 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);
```

¿Sabía que...?

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
}
```

Salida de cadenas de caracteres

Con la función `printf()` se puede dar salida a cualquier dato, asociándole 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 foto.");
    puts(st);
    puts(&st[8]);
    return 0;
}
```

Entrada de cadenas de caracteres

La entrada de una cadena de caracteres se puede hacer con la función `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 hay que 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 pone simplemente `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.



Resumen

Este capítulo le ha introducido a los componentes básicos de un programa C. En posteriores capítulos se analizarán en profundidad cada uno de dichos componentes. Las características fundamentales que se aprenderán son:

- 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; este, 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 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.
- Los programas y funciones utilizan constantes y variables para construir acciones requeridas. Las variables, se simbolizan con identificadores; pueden tomar distintos valores, según el tipo de dato con el que están definidas, durante la vida de una variable. En general, según la duración, una variable puede ser global, local y dinámica



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.

```
*****
*   *
*   *
*****
*   *
*   *
*   *
*****
*****
```



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 relationales
- 4.6 Operadores lógicos
- 4.7 Operador condicional ?:

4.8 Operador coma

4.9 Operadores especiales: (), []

4.10 El operador sizeof

4.11 Conversiones de tipos

4.12 Prioridad y asociatividad

- › Resumen
- › Ejercicios
- › Problemas

Introducción

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Este capítulo muestra cómo 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 relationales.
- 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.

4.1 Operadores y expresiones

Los programas C constan de datos, sentencias de programas y *expresiones*. Una *expresión* es, normalmente, una ecuación matemática, como $3 + 5$. En esta expresión, el símbolo más (+) es el *operador* de suma, y los números 3 y 5 se llaman *operando*s. En síntesis, una *expresión* es una secuencia de operadores y operando que especifica un cálculo.



Conceptos clave

- › Asignación
- › Asociatividad
- › Conversión explícita
- › Conversiones de tipos
- › Expresión
- › Incrementación y decrementación
- › Operador
- › Operador sizeof
- › Precedencia
- › Prioridad

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.

A tener en cuenta

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, como `25` o `'Z'`; pueden ser valores o variables combinadas con operadores (`a++`, `m==n`, etc.); o bien pueden ser valores combinados con funciones como `toupper('b')`.

4.2 Operador de asignación

El operador de asignación, `=`, 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;
```

C proporciona un operador abreviado de asignación (`*=`), que realiza una asignación equivalente,

```
i *= 10;  equivale a 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.

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 con 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, 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
$*, /, \%$	$5*5 \text{ es } 25$ $25/5 \text{ es } 5$ $25\%6 \text{ es } 1$	2
$+, -$	$2+3 \text{ es } 5$ $2-3 \text{ es } -1$	3

Observe que los operadores $+$ y $-$, cuando se utilizan delante de un operando, 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: $6 + 2 * 3 - 4/2$?

$$\begin{array}{r}
 6 + 2 \underbrace{* 3} - 4/2 \\
 6 + 6 \quad - 4/2 \\
 \underbrace{6 + 6} \quad - 2 \\
 \hline
 12 \quad - 2 \\
 \hline
 10
 \end{array}$$

2. ¿Cuál es el resultado de la expresión: $5 * (5 + (6 - 2) + 1)$?

$$\begin{array}{r}
 5 \quad * \quad (5 + \underbrace{(6 - 2)}_{}) + 1 \\
 5 \quad * \quad (5 + \underbrace{4}_{}) + 1 \\
 5 \quad * \quad \underbrace{10}_{50}
 \end{array}$$

3. ¿Cuál es el resultado de la expresión: $7 - 6/3 + 2*3/2 - 4/2$?

$$\begin{array}{r}
 7 - \underbrace{6/3}_{2} + \underbrace{2*3/2}_{6/2} - \underbrace{4/2}_{2} \\
 7 - 2 + \underbrace{2*3/2}_{6/2} - \underbrace{4/2}_{2} \\
 7 - 2 + 3 - \underbrace{4/2}_{2} \\
 \underbrace{7 - 2}_{5} + 3 - 2 \\
 \underbrace{5 + 3}_{8} - 2
 \end{array}$$

Asociatividad

En una expresión 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,

$x - y + z$ se agrupa como $(x - y) + z$

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
$+, -$ (unitarios)	izquierda-derecha (\rightarrow)
$*, /, \%$	izquierda-derecha (\rightarrow)
$+, -$	izquierda-derecha (\rightarrow)

¿Cuál es el resultado de la expresión: $7 * 10 - 5 \% 3 * 4 + 9$?

Existen tres operadores de prioridad más alta ($*$, $\%$ y $*$), primero se ejecuta $*$

$70 - 5 \% 3 * 4 + 9$

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 la 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:

71

Ejemplo 4.2



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 después 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 evalúen 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 debe tener un correspondiente paréntesis a la derecha que aparece posteriormente en la expresión, ya que si existen paréntesis desequilibrados se producirá un error de compilación.

`((8 - 5) + 4 - (3 + 7))` 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.

Por consiguiente,

`a++`

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++;`

Tabla 4.6 Operadores de incrementación `(++)` y decrementación `(--)`.

Incrementación	Decrementación
<code>++n</code>	<code>--n</code>
<code>n += 1</code>	<code>n -= 1</code>
<code>n = n + 1</code>	<code>n = n - 1</code>

tienen el mismo efecto; así como,

```
--n;
n--;
```

Sin embargo, cuando se utilizan como expresiones como,

```
m = n++;
printf(" n = %d", n--);
```

el resultado es distinto si se utilizan como prefijo.

```
m = ++n;
printf(" n = %d", --n);
```

`++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--`. Algunos ejemplos prácticos

```
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 */
```

Otros ejemplos adicionales,

```
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 */
```

¿Sabía que...?

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.

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 );
}
```

Ejemplo 4.3



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
```

Se podría pensar que el resultado de t sería 30, en realidad es 25, debido a que en la asignación de t , n se incrementa a 6 y a continuación se decrementa a 5 antes de que se evalúe el operador producto, calculando $5 * 5$. Por último, las tres subexpresiones se evalúan de derecha a izquierda, el resultado será 8 7 6 en lugar de 6 7 8 si la evaluación fuera de izquierda a derecha.

4.5 Operadores relacionales

C no tiene tipos de datos lógicos o booleanos 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*.

falso *cero*
verdadero *distinto de cero*

Operadores como `>` y `==` que comprueban una relación entre dos operandos se llaman **operadores relacionales** y se utilizan en expresiones de la forma:

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`, etcétera.

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,

$$c = 3 \leq 7;$$

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 experimentados, es confundir el operador de asignación (`=`) con el operador de igualdad (`==`).

Tabla 4.7 Operadores relacionales de C.

Operadores Relacionales de C		
Operador	Significado	Ejemplo
<code>==</code>	<i>Igual a</i>	<code>a == b</code>
<code>!=</code>	<i>No igual a</i>	<code>a != b</code>
<code>></code>	<i>Mayor que</i>	<code>a > b</code>
<code><</code>	<i>Menor que</i>	<code>a < b</code>
<code>>=</code>	<i>Mayor o igual que</i>	<code>a >= b</code>
<code><=</code>	<i>Menor o igual que</i>	<code>a <= b</code>

Ejemplos

- Si x , a , b y c son de tipo double, numero es 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 (*vea* 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 se utiliza la expresión:

```
if (Nota_asignatura == 9)
```

Si, por el contrario, se desea comprobar si la variable y el número *no son iguales*, entonces utiliza 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 (apuntadores, 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 solo* 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* solo si ambos operandos son *falsos*. La tabla 4.8. muestra los operadores lógicos de C.

Al igual que los operadores matemáticos, el valor de una expresión formada con operadores lógicos depende de: *a* el operador y *b* sus argumentos. Con operadores lógicos existen solo 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 cómo funcionan cada uno de los operadores lógicos.

Tabla 4.8 / Operadores lógicos.

Operador	Operación lógica	Ejemplo
Negación (<code>!</code>)	No lógica	<code>! (x >= y)</code>
Y lógica (<code>&&</code>)	<code>operando_1 && operando_2</code>	<code>m < n && i > j</code>
O lógica <code> </code>	<code>operando_1 operando_2</code>	<code>m = 5 n != 10</code>

Tabla 4.9 / Tabla de verdad del operador lógico NOT (`!`).

Operando (a)	<code>!a</code>
Verdadero (1)	Falso (0)
Falso (0)	Verdadero (1)

Tabla 4.10 / Tabla de verdad del operador lógico AND (`&&`).

Operandos		
a	b	<code>a && b</code>
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	<code>a b</code>
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Verdadero (1)
Falso (0)	Verdadero (1)	Verdadero (1)
Falso (0)	Falso (0)	Falso (0)

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.

¿Sabía que...?

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)))...
```

Asignaciones booleanas (lógicas)

Las sentencias de asignación booleanas se pueden escribir de modo que den 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 a MayorDeEdad.
Cuando edad es mayor que 18, MayorDeEdad es 1, si no 0 */
juvenil = (edad > 15) && (edad <= 18); /* asigna 1 a juvenil si edad está
comprendida entre 15 (mayor que 15) y 18 (inclusive 18). */
```

Ejemplo 4.6

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.

Tabla 4.12 / Operadores de direcciones.

Operador	Acción
*	Lee el valor apuntado por la expresión. El operando se corresponde con un puntero y el resultado es del tipo apuntado.
&	Devuelve un apuntador (puntero) al objeto utilizado como operando, que debe ser un lvalue (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.

4.7 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 expresiones. El formato del operador condicional es:

```
expresion_c ? expresion_v : expresion_f;
```

Se evalúa *expresion_c* y su valor (cero = *false*, 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 4.1 muestra el funcionamiento del operador condicional.

Otros ejemplos del uso del operador ?: son:

```
n >= 0 ? 1 : -1      /* 1 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 0, en caso contrario un tabulador (\t).
`printf("%d %c", x, x%5 ? '\t' : '\n');`

```
(ventas > 150000) ? comision = 100 : comision = 0;
```

si ventas es mayor

si ventas no es

que 150 000 se

mayor que 150 000 se

ejecuta:

ejecuta:

comision = 100

comision = 100

Figura 4.1 Formato de un operador condicional.

¿Sabía que...?

La procedencia de `:` es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es a derechas.

4.8 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:

`expresión1, expresión2, expresión3, ..., expresiónn`

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 `i` está declarada e inicializada antes que la segunda variable `j`. Otros ejemplos son:

<code>i++, j++ ;</code>	<i>equivale a</i>	<code>i++; j++;</code>
<code>i++, j++, k++ ;</code>	<i>equivale a</i>	<code>i++; j++; k++;</code>

¿Sabía que...?

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 *expresiónⁿ*. Por ejemplo,

```
int i, j, resultado;
resultado = j = 10, i = j, ++i;
```

El valor de esta expresión es 11. En primer lugar, a `j` se le asigna el valor 10, a continuación a `i` se le asigna el valor de `j`. Por último, `i` se incrementa en 1 y pasa a ser 11, que es el resultado de la expresión. 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.

4.9 Operadores especiales: (), []

C admite algunos operadores especiales que sirven para propósitos diferentes. Entre ellos destacan: `()`, `[]`.

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 y resolver los conflictos de prioridad entre operadores.

El operador []

Sirve para dimensionar los arreglos (*arrays*) y designar un elemento de un arreglo (*array*).

Ejemplos de ello:

```
double v[20];           /* define un arreglo (array) de 20 elementos */
printf("v[2] = %e", v[2]); /* escribe el elemento 2 de v */
return v[i-1];          /* devuelve el elemento i-1 */
```

4.10 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, arreglo [array], registro, etc.). El formato del operador es

```
sizeof (nombre_variable)
sizeof nombre_variable
sizeof (tipo_dato)
sizeof (expresión)
```

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));
```

Ejemplo 4.7



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 especificado. Debido a que la mayoría de los tipos de datos y variables requieren diferentes cantidades de almacenamiento interno en computadoras distintas, el operador `sizeof` permite consistencia de programas en diferentes tipos de computadoras.

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.

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 esa 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;
}
```

Ejemplo 4.8



Este programa producirá diferentes resultados en distintas clases de computadoras. 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
```

4.11 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 de manera específica por el programador). C hace muchas conversiones de tipos automáticamente:

- C convierte valores cuando se asigna un valor de un tipo a una variable de otro tipo.
- C convierte valores cuando se combinan tipos mixtos en expresiones.
- C convierte valores cuando se pasan argumentos a funciones.

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 double: 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 */
```

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 la lista se convierte en el que viene después. 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 resultado impredecible */
```

Conversión explícita

C fuerza la *conversión explícita* de tipos mediante el operador de *molde* (*cast*). El operador de *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 apuntadores (punteros). */
```

El operador *molde* (*tipo, cast*) tiene la misma prioridad que otros operadores unitarios como – y !

```
precios = (int)19.99 + (int)11.99;
```

4.12 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 quince 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.

Tabla 4.13 / Prioridad de operadores.

Prioridad	Operadores	Asociatividad
1	· -> [] ()	I – D
2	++ -- ~ ! - + & * sizeof (tipo)	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	? : (expresión condicional)	D – I
14	= *= /= %= += -= <=>= &= /= ^=	D – I
15	, (operador coma)	I – D

I-D: Izquierda-Derecha.

D-I: Derecha-Izquierda.



Resumen

Este capítulo examina los siguientes temas:

- Concepto de operadores y expresiones.
- Operadores de asignación: básicos y aritméticos
- Operadores aritméticos, incluyendo `+`, `-`, `*`, `/` y `%` (módulo).
- 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 y enteros.
- Operadores relacionales y lógicos que permiten construir expresiones lógicas. C no soporta tipo lógico (*boolean*) predefinido y en su lugar considera 0 (cero) como *falso* y cualquier valor un distinto de cero como *verdadero*.

- 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.



Ejercicios

- 4.1 Determinar el valor de las siguientes expresiones aritméticas:

$$\begin{aligned} 15 &/ 12 \quad 15 \% 12 \\ 24 &/ 12 \quad 24 \% 12 \\ 123 &/ 10 \quad 123 \% 100 \\ 200 &/ 100 \quad 200 \% 100 \end{aligned}$$

- 4.2 ¿Cuál es el valor de cada una de las siguientes expresiones?

$$\begin{aligned} a) \quad &15 * 14 - 3 * 7 \\ b) \quad &-4 * 5 * 2 \\ c) \quad &(24 + 2 * 6) / 4 \\ d) \quad &a / a / a * b \\ e) \quad &3 + 4 * (8 * (4 - (9 + 3) / 6)) \\ f) \quad &4 * 3 * 5 + 8 * 4 * 2 - 5 \\ g) \quad &4 - 40 / 5 \\ h) \quad &(-5) \% (-2) \end{aligned}$$

- 4.3 Escribir las siguientes expresiones aritméticas como expresiones de computadora: La potencia puede hacerse con la función `pow()`, por ejemplo $(x + y)^2$ es `pow(x+y, 2)`.

$$\begin{aligned} a) \quad &\frac{x}{y} + 1 & e) \quad &(a + b) \frac{c}{d} \\ b) \quad &\frac{x + y}{x - y} & f) \quad &[(a + b)^2]^2 \\ c) \quad &\frac{x + \frac{y}{z}}{x + \frac{y}{z}} & g) \quad &\frac{x + y}{1 - 4x} \\ d) \quad &\frac{b}{c + d} & h) \quad &\frac{xy}{mn} \\ i) \quad &(x + y)^2 \cdot (a - b) \end{aligned}$$

- 4.4 ¿Cuál de los siguientes identificadores son válidos?

n	85	Nombre
MiProblema	AAAAAAA	AAAAAAA
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 y *y* una variable carácter. Qué resultados producirá la sentencia `scanf ("%d %d", &x, &y)` si la entrada es

$$\begin{aligned} a) \quad &5 \ c \\ b) \quad &5C \end{aligned}$$

- 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* y *y*, y a continuación obtenga los valores de: $1 \cdot x/y$, $2 \cdot x \% y$. 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 (formato `% .4f`).

- 4.10 Escribir un programa que convierte 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. 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;
    puts ("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 Escribir 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 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.

$x < 0$	<i>o bien</i>	$0 \leq x \leq 100$
	<i>o bien</i>	$x > 100$

- 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 = año % 19
B = año % 4
C = 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.

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 entre el segundo. La salida será justificada a la derecha.

- 4.2 Una temperatura Celsius (centígrados) puede ser convertida a una temperatura equivalente F de acuerdo con la siguiente fórmula:

$$f = \frac{9}{5} C + 32$$

Escribir un programa que lea la temperatura en grados Celsius y la escriba en F.

- 4.3 Un sistema de ecuaciones lineales

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

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 y 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, $p = (a + b + c)/2$, y a, b, c son los tres lados del triángulo.

- 4.10 Escribir 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 dólares 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

$$G = 6.673 \cdot 10^{-8} \text{ cm}^3/\text{g} \cdot \text{s}^2$$

Escribir un programa que lea la masa de dos cuerpos y la distancia entre ellos y a continuación obtenga la fuerza gravitacional entre ellos. La salida debe ser en dinas; un dina es igual a $g \cdot \text{cm/s}^2$.

- 4.16 La famosa ecuación de Einstein para conversión de una masa m en energía viene dada por la fórmula

$$E = mc^2 \quad c \text{ es la velocidad de la luz}$$

$$c = 2.997925 \cdot 10^{10} \text{ cm/s}$$

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 la energía en ergs.

- 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 en 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

- 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 múltiples de 100 solo son bisiestos cuando a la vez son múltiples de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo es).

- 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 ?:.



Estructuras de selección: sentencias `if` y `switch`

Contenido

- 5.1 Estructuras de control
- 5.2 La sentencia `if`
- 5.3 Sentencia `if` de dos opciones: `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
 - › Resumen
 - › Ejercicios
 - › 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 y en qué momentos. Las *estructuras o construcciones de control* regulan 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

- › Condición
- › Estructuras de control
- › Estructura de control de selección
- › Sentencia `break`
- › Sentencia compuesta
- › Sentencia `if`
- › Sentencia `switch`
- › Expresiones lógicas en C

5.1 Estructuras de control

Las *estructuras de control* regulan 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.

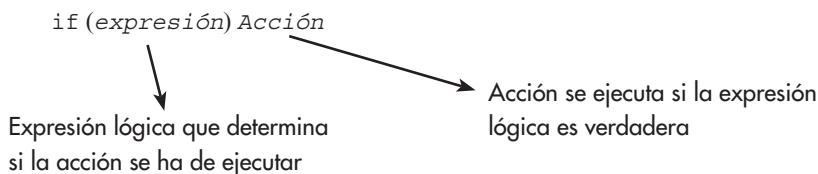
Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: *secuencia, selección (decisión)* y *repetición*. Hasta este momento solo se ha utilizado el flujo secuencial. Una *sentencia compuesta* es un conjunto de sentencias encerradas entre llaves (`{` y `}`) que se utiliza para especificar un flujo secuencial.

```
{
    sentencia1;
    sentencia2;
    .
    .
    .
    sentencian;
}
```

El control fluye de la *sentencia₁* a la *sentencia₂* y así sucesivamente. Sin embargo, existen problemas que requieren etapas con dos o más opciones 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:



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 para representar la sentencia `if` es:

```
if (condición) sentencia;
```

condición es una expresión entera (lógica).

sentencia es cualquier sentencia ejecutable, que se ejecutará solo si la *condición* toma un valor distinto de cero.

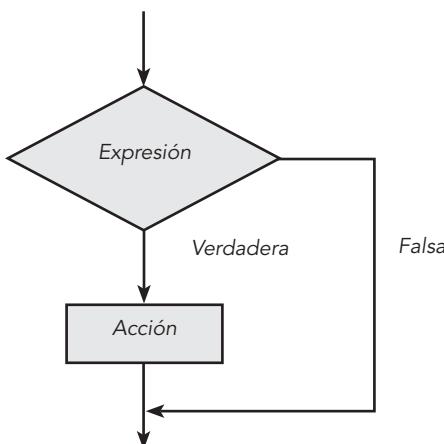


Figura 5.1 Diagrama de flujo de una sentencia básica `if`.

Prueba de divisibilidad.

Ejemplo 5.1

```
#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 entre %d\n", n, d);
    return 0;
}
```

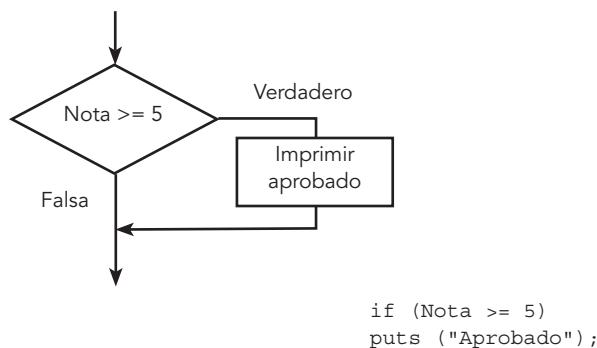
Ejecución

```
Introduzca dos enteros: 36      4
36 es divisible entre 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 entre d (en nuestro caso 36 es divisible entre 4, ya que $36 : 4 = 9$ y el resto es 0).

Representar la superación de un examen (Nota ≥ 5 , Aprobado).

Ejemplo 5.2



```
#include <stdio.h>
void main()
{
    float Nota;
    printf("Introduzca la nota obtenida (0-10): ");
    scanf("%f", &Nota);
    /* compara Nota con 5 */
    if (Nota >= 5)
        puts("Aprobado");
}
```

Una ejecución del programa produce:

```
Introduzca la nota obtenida (0-10): 6.5
Aprobado
```

Otra ejecución del programa:

```
Introduzca la nota obtenida (0-10): 3.5
```



Ejemplo 5.3

Comparar número introducido por usuario.

```
#include <stdio.h>

void main()
{
    float numero;

    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 con tres sentencias `if` comprueba si el número es positivo, negativo o cero.

11

Ejercicio 5.1

Visualizar la tarifa de la luz según el gasto de corriente eléctrica. Para un gasto menor de 1 000 kwh la tarifa es 1.2, entre 1 000 y 1 850 kwh es 1.0 y mayor de 1 850 kwh, 0.9.

```
#include <stdio.h>
#define TARIFA1 1.2
#define TARIFA2 1.0
#define TARIFA3 0.9

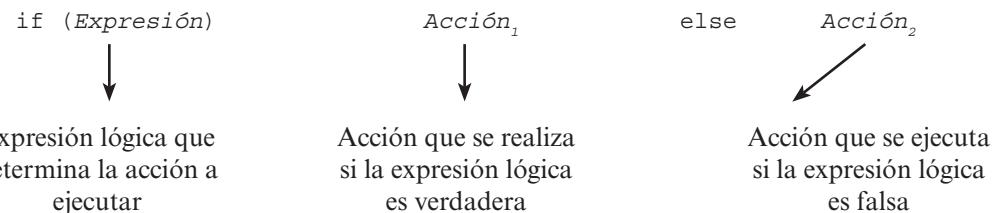
int main()
{
    float gasto, tasa;
    printf("\n Gasto de corriente: ");
    scanf("%f", &gasto);
    if (gasto < 1000.0)
        tasa = TARIFA1;
    if (gasto >= 1000.0 && gasto <= 1850.0)
        tasa = TARIFA2;
    if (gasto > 1850.0)
        tasa = TARIFA3;

    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 opciones: `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 Acción₁ y Acción₂ 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 Acción₁ y en caso contrario se ejecuta Acción₂. La figura 5.2 muestra la semántica de la 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 ("Reprobado");
Si Nota es mayor o igual que 5 se escribe Aprobado; en caso contrario, Nota menor que 5, se escribe Reprobado.
```

Formatos

1. if (expresión_lógica)
sentencia

3. if (expresión_lógica) sentencia

2. if (expresión_lógica)
sentencia₁
else
sentencia₂

4. if (expresión_lógica) sentencia₁
else sentencia₂

Si expresión lógica es verdadera se ejecuta sentencia o bien sentencia₁, si es falsa (sino, en caso contrario) se ejecuta sentencia₂.

Ejemplos

1. if (x > 0.0)
 producto = producto * x;

2. if (x != 0.0)
 producto = producto * x;

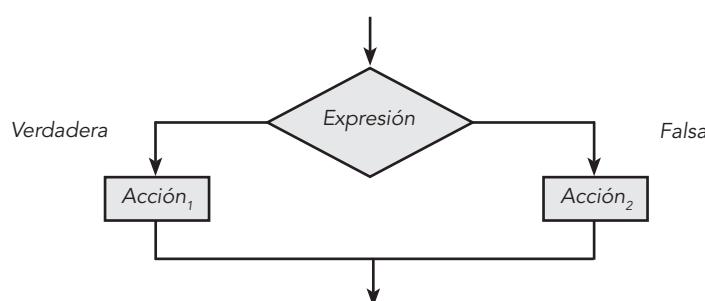


Figura 5.2 Diagrama de flujo de la representación de una sentencia if-else.

```

/* La segunda sentencia if la sentencia de asignación cuando x no es igual
a 0.0: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 entre %d\n",n,d);
    else
        printf("%d no es divisible entre %d\n",n,d);
    return 0;
}

```

Ejecución

```

Introduzca dos enteros: 36    5
36 no es divisible entre 5

```

Comentario

36 no es divisible entre 5 ya que 36 dividido entre 5 produce un resto de 1 ($n \% d == 0$, 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
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 , en caso contrario se imprime y .

Dada la función $f(x)$, calcular la función para un valor dado de x y visualizarlo en pantalla

Ejemplo 5.6



$$f(x) = \begin{cases} x^2 - x & \text{para } x \leq 0.0 \\ -x^2 + 3x & \text{para } x > 0 \end{cases}$$

```

#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 opciones. En esta sección, se mostrará cómo se puede utilizar la sentencia `if` para implementar decisiones que impliquen diferentes opciones.

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 multialternativas.

Sintaxis:

<pre> if (condición₁) sentencia₁ else if (condición₂) sentencia₂ . . . </pre>	<pre> else if (condición_n) sentencia_n else sentencia_{n+1} </pre>
---	---

**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;
    else
        num_ceros = num_ceros + 1;

```

La sentencia `if` anidada del ejemplo 5.7 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 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 *solo* si la primera condición es *falsa*.

Sangría en las sentencias `if` anidadas

El formato multibifurcación se compone de una serie de sentencias `if` anidadas, en cada línea se puede escribir una sentencia `if`. La sintaxis multibifurcación anidada es:

Formato 1:

```

if expresión_lógica1) sentencia1
else
    if (expresión_lógica2)
        sentencia2
    else
        if (expresión_lógica3)
            sentencia3
        else
            if (expresión_lógica4)
                sentencia4
            else
                sentencia5

```

Formato 2:

```

if (expresión_lógica1)
    sentencia1
else if (expresión_lógica2)
    sentencia2
else if (expresión_lógica3)
    sentencia3
else if (expresión_lógica4)
    sentencia4
else
    sentencia5

```

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");`

El siguiente programa realiza selecciones múltiples con la sentencia compuesta `if-else`.

Ejemplo 5.8

```
#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 positivo");
    }
    else
    {
        printf("%.2f %s", numero, "es igual a cero\n");
        puts( " ¿por qué no introduce otro número? ");
    }
}
```



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 cuál 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 ( ...)
        {
            ...
        }
    }
    ...
}

```



Ejemplo 5.9

Calcular el mayor de tres números enteros.

```

#include <stdio.h>
int main()
{
    int a, b, c, mayor;
    printf("\n Introduzca 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 el segundo `if`. En el segundo `if` la condición (`a > c`) es falsa, en consecuencia se ejecuta `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 etiqueta1 : sentencias1;
    case etiqueta2 : sentencias2;
    .
    .
    .
    case etiquetan : sentenciasn;
    default: sentenciasd; /* 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, `etiqueta1`, entonces la ejecución comenzará con la primera sentencia de la secuencia `sentencia1` y continuará hasta que se encuentre 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. La sentencia `break` hace que siga la ejecución en la siguiente sentencia al `switch`.

Sintaxis con break

```
switch (selector)
{
    case etiqueta1 : sentencias1;
        break;
    case etiqueta2 : sentencias2;
        break;
    .
    .
    .
    case etiquetan : sentenciasn;
        break;
    default: sentenciasd; /* opcional */
}
```

El tipo de cada etiqueta debe ser el mismo que la expresión de `selector`. Las expresiones están permitidas como etiquetas pero solo si cada operando de la expresión es por sí mismo 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 en forma predeterminada. 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 secuencialmente las siguientes sentencias.

**Ejemplo 5.10**

Elección de tres opciones y un valor en forma predeterminada.

```
switch (opcion)
{
    case 0:
        puts("Cero!");
        break;
    case 1:
        puts("Uno!");
        break;
    case 2:
        puts("Dos!");
        break;
    default:
        puts("Fuera de rango");
}
```

**Ejemplo 5.11**

Selección múltiple, tres etiquetas ejecutan la misma sentencia.

```
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 5.12**

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 'o': case 'O':
case 'u': case 'U':
    printf( "%c es una vocal\n",car);
    break;
default:
    printf( "%c no es una vocal\n",car);
}

```

Dada una nota de un examen mediante un código escribir el literal que le corresponde a la nota.

Ejemplo 5.13

```

/* Programa resuelto con la sentencia switch */
#include <stdio.h>

int main()
{
    char nota;
    printf("Introduzca calificación (A-F) y pulse Intro:");
    scanf("%c",&nota);

    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("Reprobado");
                    break;
        default:
                    puts("No es posible esta nota");
    }
    puts("Final de programa");
    return 0;
}

```

Cuando se ejecuta la sentencia `switch` en el ejemplo 5.13 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 1 del ejemplo 5.13.

Introduzca calificación (A-F) y pulse Intro: A
 Excelente. Examen superado
 Final de programa

Ejecución 2 del ejemplo 5.13.

Introduzca calificación (A-F) y pulse Intro: **B**
 Notable. Suficiencia
 Final de programa

Ejecución 3 del ejemplo 5.13.

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 un sentencia `switch` correcta sintácticamente pero no realizará las tareas previstas.

**Ejemplo 5.14**

Seleccionar un tipo de vehículo según un valor numérico.

```
int tipo_vehiculo;
printf("Introduzca tipo de vehiculo:");
scanf("%d", &tipo_vehiculo);

switch(tipo_vehiculo)
{
    case 1:
        printf("turismo\n");
        peaje = 500;
        break; ----->Si se omite esta break el vehículo primero será
                      turismo y luego autobús
    case 2:
        printf("autobus\n");
        peaje = 3000;
        break;
    case 3:
        printf("motocicleta\n");
        peaje = 300;
        break;
    default:
        printf("vehículo no autorizado\n");
}
```

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`.

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 blancos++; /* se incrementa en 1 el valor de num blancos */
        break;
    default:
        num distintos++;
}

```

Uso de sentencias switch en menús

La sentencia `if-else` es más versátil que la sentencia `switch` y se pueden utilizar unas sentencias `if-else` anidadas o multidecisión, en cualquier parte que se utilice 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 opciones 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, 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 ? expresión1 : expresión2

```

condición es una expresión lógica
 expresión₁/expresión₂ son cualquier expresión

Se evalúa `condición`, si el valor de `condición` es verdadera (distinto de cero) entonces se devuelve como resultado el valor de `expresión1`; si el valor de `condición` es falsa (cero) se devuelve como resultado el valor de `expresión2`.

Una aplicación del operador condicional (`?:`) es llamar a una de dos funciones según el valor de la variable.

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 5.15

Seleccionar el mayor de dos números enteros con la sentencia `if-else` y con el operador `?:`.

```
#include <stdio.h>

void main()
{
    int 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 emplea 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 cuál 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, `a1 && a2` se detiene si la subexpresión `a1` 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 C no evalúa la segunda expresión.

Ejemplo 5.16

Si `x` es negativo, la expresión

```
(x >= 0) && (y > 1)
```

se evalúa en cortocircuito ya que `x >= 0` será falso y, en consecuencia, el valor final de la expresión será falso. En el caso del operador `||` se produce una situación similar. Si la primera de las dos expre-

siones 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.

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*”).

Ejemplo 5.17

Precaución

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:

<pre>if (expresión_lógica) sentencia₁ else sentencia₂</pre>	<pre>if (expresión_lógica) { sentencia₁ : sentencia_k } else { sentencia_{k+1} : sentencia_n }</pre>
---	---

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_lógica1)
    sentencia1
else if (expresión_lógica2)
    sentencia2
```

```

:
else if (expresión_lógican)
    sentencian
else
    sentencian+1

```

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",&nota);
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`; cinco 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",&nota);
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 algebraicamente iguales:

```

a * (1/a)
1.0

```

la expresión

```
a * (1/a) == 1.0
```

puede ser falsa debido a que `a` es un número real.

4. *Cuando en una sentencia switch o en un bloque de sentencias falta una de las llaves ({{, }}) aparece un mensaje de error 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.*

7. *Normalmente deberá escribir la sentencia break después de la acción que se desea que ejecute cada uno de los case de la sentencia switch.*



Resumen

Sentencia `if`

Una opción

```
if (a != 0)
    resultado = a/b;
```

Dos opciones

```
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`

```
switch(sig_car)
{
    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 ("Reprobado");
        break
    default:
        puts("nota no válida");
}
```

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.

```
if (x >= 0)           if (x >= 0)
    x++;                x++;
else if (x >= 1);    if (x >= 1)
    x+= 2;              x+= 2;
```

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 negativo. 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:

$x < 0$	o bien	$0 \leq x \leq 100$
o bien		$x > 100$

- 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 Escribir un programa que clasifique enteros leídos del teclado de acuerdo con 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:

$$\begin{aligned}
 A &= \text{año} \bmod 19 \\
 B &= \text{año} \bmod 4 \\
 C &= \text{año} \bmod 7 \\
 D &= (19 * A + 24) \bmod 30 \\
 E &= (2 * B + 4 * C + 6 * D + 5) \bmod 7 \\
 N &= (22 + D + E)
 \end{aligned}$$

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 $a < 5.0$	Reprobado
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.



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 con el 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 2014

producirá la salida

15 2 2014

- 5.4 Escribir un programa que acepte un número de tres dígitos escrito en palabras 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 y M = 1000

IV = 4

MCM = 1900

MCMLX = 1960

MCMLXXXIX = 1989

XL = 40

CM = 900

MCML = 1950

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, 2008). Sin embargo, los años múltiplos de 100 solo son bisiestos cuando a la vez son múltiples de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo es).

- 5.9 Escribir un programa que calcule el número de días de un mes, dado 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 con las siguientes normas:

- Horas semanales trabajadas < 38 a una tasa dada.
- Horas extra (38 o más) a una tasa 50% superior a la ordinaria.
- Impuestos 0%, si el salario bruto es menor o igual a 800 dólares.
- Impuestos 10%, si el salario bruto es mayor de 800 dólares.

- 5.11 Determinar el menor número de billetes y monedas de curso legal equivalentes a cierta cantidad de euros (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`.



Estructuras de control: bucles

Contenido

- 6.1 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 Enumeraciones
 - › Resumen
 - › Ejercicios
 - › Problemas

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 son 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 lista de sentencias se ejecuta.

Un tipo de dato enumerado es una colección de miembros con nombre que tienen valores enteros equivalentes. Los bucles se pueden controlar con variables de tipo enumerado.



Conceptos clave

- › Bucle (ciclo)
- › Condición
- › Control de bucles
- › Diseño de bucles
- › Repetición o iteración
- › Sentencia break
- › Sentencia do-while
- › Sentencia for
- › Sentencia while
- › Terminación de un bucle

6.1 Sentencia while

Un *bucle* (*ciclo* o *lazo*, loop en inglés) 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

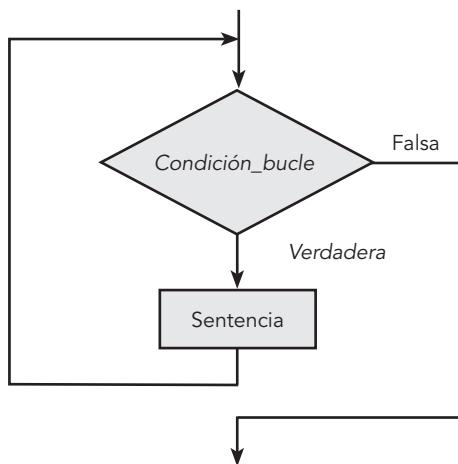


Figura 6.1 Diagrama de flujo de la sentencia while.

del cuerpo del bucle y significa que un bucle `while` es un bucle *pretest*, de modo que, cuando se ejecuta, 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 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 *ceros o más veces*.

Sintaxis:

```

1 while (condición_bucle)
    sentencia; -----> cuerpo

2 while (condición_bucle)
{
    sentencia-1;
    sentencia-2;
    :
    sentencia-n; } cuerpo
}

```

<i>while</i>	es una palabra reservada C
<i>condición_bucle</i>	es una expresión lógica o booleana
<i>sentencia</i>	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 *cuerpo* del bucle, se ejecuta.
 - b) Devuelve el control al paso 1.
3. En caso contrario:

El control se transfiere a la sentencia siguiente al bucle o *sentencia while*.

A recordar

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**.

Por ejemplo, el siguiente bucle cuenta hasta 10:

```
int x = 0;
while (x < 10)
    printf("X: %d", x++);
```

Ejemplo

Visualizar n asteriscos.

```
contador = 0;           -----> inicialización
while (contador < n) -----> prueba/condición
{
    printf(" * ");
    contador++; -----> actualización (incrementa en uno contador)
}      /* fin de while */
```

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 en el ejemplo anterior 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 o pasada*).
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 produce 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--; -----> decrementa en uno contador
}
```

La variable `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 ser 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
```

Operadores de incremento y decremento (++, --)

C ofrece los operadores de incremento (++) y decremento (--) que admiten 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 sucesivos valores 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 ++).

```
/* El programa realiza un cálculo de calorías */
#include <stdio.h>
int main()
{
    int num_de_elementos, cuenta, calorias_por_alimento, calorias_total;
    printf("¿Cuántos alimentos ha comido hoy? ");
    scanf("%d ", &num_de_elementos);
    calorias_total = 0;
```

```

cuenta = 1;
printf("Introducir el número de calorías de cada uno de los ");
printf("%d %s", num_elementos, "alimentos tomados: \n");
while (cuenta++ <= num_de_elementos)
{
    scanf("%d", &calorias_por_alimento);
    calorias_total += calorias_por_alimento;
}
printf("Las calorías totales consumidas hoy son = ");
printf("%d \n", calorias_total);
return 0;
}

```

Ejecución

```

¿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

```

Terminaciones anormales de un bucle (ciclo o llave)

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++;
}

```

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.

Bucles while con cero iteraciones

El cuerpo de un bucle `while` 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, y en consecuencia su cuerpo no se ejecuta.

Bucles controlados por centinelas

Normalmente, no se conoce con exactitud cuántos elementos de datos se procesarán antes de comenzar su ejecución. Esto se produce, bien porque hay muchos datos a contar 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 para que introduzca 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. Este valor 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",&nota);
while (nota != centinela)
{
    cuenta++;
    suma += nota;
    printf("Introduzca siguiente nota: ");
    scanf("%d",&nota);
} /* 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 esta:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1      /* valor del centinela */
Final
```

Bucles controlados por indicadores (banderas)

En lenguajes como Pascal, que tienen el tipo `boolean`, se utilizan variables *booleanas* (lógicas) 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 `boolean`, 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.

Se desean 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’ a ‘9’).

Ejemplo 6.3



La variable bandera, `digito_leido`, se utiliza como un indicador que representa el hecho de que un dígito se ha introducido por teclado.

Variable bandera	Significado
<code>digito_leido</code>	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, este debe continuar ejecutándose mientras el dato leído sea un carácter y en consecuencia la variable `digito_leido` tenga valor falso y se debe detener el bucle cuando el dato leído sea un dígito; en este caso el valor de la variable `digito_leido` se debe cambiar a verdadero (1). 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.

A recordar

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, en consecuencia, 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 $f(x) = x * \log(x)$.

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, este debe continuar ejecutándose mientras el número leído sea negativo o cero y en consecuencia la variable `x_positivo` tenga el valor falso; 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 (1). 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 x_positivo;
    x_positivo = 0; /* inicializado a falso */
    while (!x_positivo)
    {
        printf("\n Valor de x: ");
        scanf("%f", &x);
        x_positivo = (x > 0.0); /* asigna verdadero(1) si cumple la condición */
    }
    f = x * log(x);
    printf(" f(%1.1f) = %1.3e", x, f);
    return 0;
}
```

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`, que es su uso más frecuente.

El esquema de un bucle `while` con sentencia `break` es:

```
while (condición)
{
    if (condición2)
        break;
    /* sentencias */
}
```

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-archivo*. Al devolver un valor distinto de cero el bucle se ejecutaría indefinidamente, sin embargo, cuando se introduce `entrada==clave` la ejecución sigue por `else` y la sentencia `break` que hace que la ejecución siga en la sentencia siguiente al bucle `while`.

Ejemplo 6.5

Consejo de programación

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 `break`:

```
int clave= -9;
int entrada;
while (scanf("%d", &entrada) && (entrada != clave))
{
    printf("%d\n", entrada);
}
```

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. El siguiente programa contiene un bucle `while (true)`:

```
#include <stdio.h>
int main()
{
    int flag = 1, contador = 0;
    while(flag)
    {

```

```

    contador++;
    if (contador > 10)
        break;
    }
    printf("Contador: %d\n", contador);
    return 0;
}

```

Ejecución

Contador:11

El bucle `while` se establece con una condición que nunca puede ser falsa. El bucle incrementa la variable `contador`, y a continuación comprueba si el contador es mayor que 10. Si no es así el bucle itera de nuevo. Si `contador` es mayor que 10, la sentencia `break` termina el bucle `while`, y la ejecución del programa pasa a la sentencia `printf ()`.

6.2 Repetición: el bucle `for`

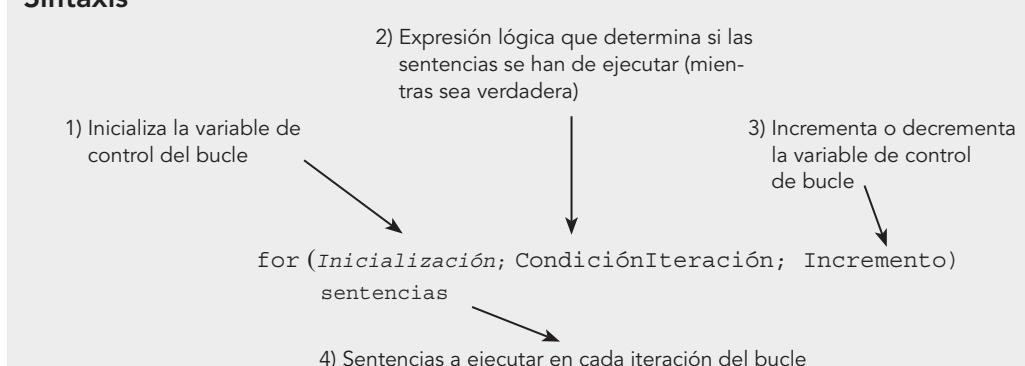
El bucle `for` de C es más potente que los bucles `for` de otros lenguajes de programación clásicos como Pascal o FORTRAN ya que ofrece más control sobre la inicialización y el incremento de las variables de control del bucle.

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 con el 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;
}
```

Imprimir `Hola!` 10 veces.

```
int i;
for (i = 0; i < 10; i++)
    printf("Hola!");
```

Ejemplo 6.6

Escribe `Hola!` y el valor de la variable de control.

```
int i;
for (i = 0; i < 10; i++)
{
    printf("Hola!\n");
    printf("El valor de i es: %d", i);
}
```

Ejemplo 6.7

Programa que imprime 15 valores de la función $f(x) = e^{2x} - x$.

```
#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(%lf) = %.4g\n", x, f(x));
    }
    return 0;
}
```

Ejemplo 6.8

En el ejemplo 6.8 se define la constante simbólica `M` y una “función en línea” (también llamada “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:

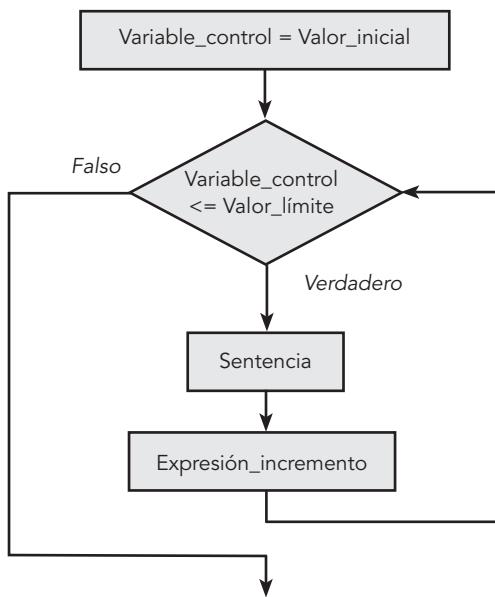
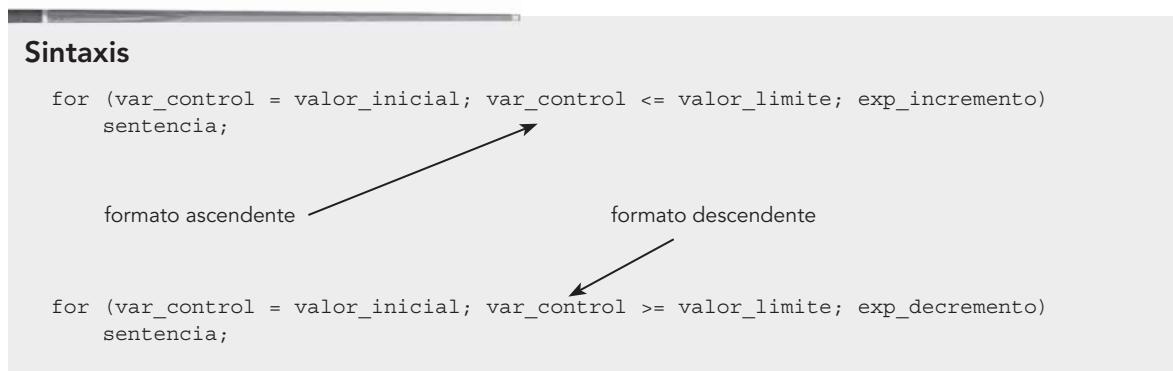


Figura 6.2 Diagrama de sintaxis de un bucle for.

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.



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	6	36
2	4	7	49
3	9	8	64
4	16	9	81
5	25	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 aumenta el valor de la variable en 1. Por consiguiente, el bucle se realiza tantas veces como letras mayúsculas hay en el alfabeto.

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 decrementa en 3, el bucle se ejecuta cuatro veces con el valor de la variable de control *i*, 9, 6, 3 y 0.

En 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 como el siguiente 128 no cumple la condición, termina el bucle.

El ejemplo 4 declara dos variables de control *i* y *j*, las inicializa a 0 y a la constante MAX. El bucle se ejecutará mientras *i* sea menor que *j*. La variable de control *i* se incrementa en 1, y a la vez *j* se decrementa en 1.



Ejemplo 6.9

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;
```

Diferentes usos de bucles for

El lenguaje C permite que:

- El valor de la variable de control se pueda modificar en valores diferentes de 1.
- Se pueda utilizar más de una variable de control.

La(s) variable(s) de control se puede(n) incrementar o decrementar en valores de tipo *int*, pero también es posible en valores de tipo *float* o *double* y en consecuencia se incrementaría o decrementaría en una cantidad decimal.

Ejemplos de incrementos/decrementos con variables de control diferentes

```
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 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 deseé. 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 mismo se convierta en *false* (*falso*) 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` de conteo, 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);
    limite++;
}
```

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 es:

```
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);
}
```

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 puede determinar por anticipado. Para estas situaciones algunos lenguajes clásicos tienen sentencias específicas 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 cuál es la condición para terminar la repetición de sentencias, supone 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 de 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;
condición     es una expresión lógica
break        termina la ejecución del bucle y transfiere el control a la sentencia siguiente al bucle
```

y la sintaxis completa:

```
for (;;)                                /* bucle */
{
    lista_sentencias1
    if (condición_terminación) break;
    lista_sentencias2
}
                                /* fin del bucle */
                                puede ser vacía, simple o compuesta.
```



Ejemplo 6.10

Bucle infinito, termina al teclear una clave.

```
#define CLAVE -999
for (;;)
{
```

```

printf("Introduzca un número, (%d) para terminar", CLAVE);
scanf("%d ", &num);
if (num == CLAVE) break;
...
}

```

Los bucles `for` vacíos

Tenga cuidado de no 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 lo tanto, después de que el bucle `for` haya terminado, se ejecuta la siguiente sentencia `puts` y se escribe "Sierra Magina".

A recordar

El bucle `for` con cuerpos vacíos puede tener algunas aplicaciones, especialmente cuando se requieren ralentizaciones o temporizaciones.

Sentencias nulas en bucles `for`

Cualquiera o todas las expresiones de un bucle `for` pueden ser nulas. Para ejecutar esta acción, se utiliza el punto y coma (`;`) para marcar la expresión vacía. Si se desea crear un bucle `for` que actúe exactamente como un bucle `while`, se deben incluir las primeras y terceras expresiones vacías. El siguiente programa contiene un bucle `for` con la expresión de inicialización y de incremento vacías.

```

#include <stdio.h>
int main()
{
    int contador = 0;
    for (; contador < 5;)
    {
        contador++;
        printf("¡Bucle!");
    }
    printf("\n Contador: %d \n", contador);
    return 0;
}

```

Ejecución

```

¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle!
Contador: 5

```

La sentencia `for` no inicializa ningún valor, pero incluye una prueba de `contador < 5`. No existe ninguna sentencia de incremento, de modo que el bucle se comporta exactamente como la sentencia siguiente:

```

while(contador < 5)
{

```

```

    contador++;
    printf("¡Bucle!");
}

```

Sentencias break y continue

La sentencia `break` termina la ejecución de un bucle, de una sentencia `switch`, en general de cualquier sentencia. El siguiente programa utiliza la sentencia `break` para salir de un bucle infinito.

```

#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;
}

```

Ejecución

```

Cuantos holas? 3
Hola!
Hola!
Hola!

```

La sentencia `continue` hace que la ejecución de un bucle vuelva a la cabecera del bucle.

El siguiente programa utiliza `continue` en un bucle para que si se cumple la condición de la sentencia `if` vuelva a la cabecera e incremente `i` en 1 (`i++`)

```

#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` valga 8.

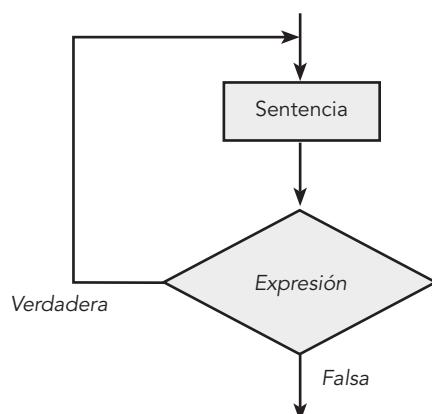
6.4 Repetición: el bucle do-while

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.



Después de cada ejecución de `sentencia` se evalúa `expresión`: si es falsa, se termina el bucle y se ejecuta la siguiente sentencia; si es verdadera, se repite el cuerpo del bucle (`la sentencia`).

Figura 6.3 Diagrama de flujo de la sentencia `do`.

**Ejemplo 6.11**

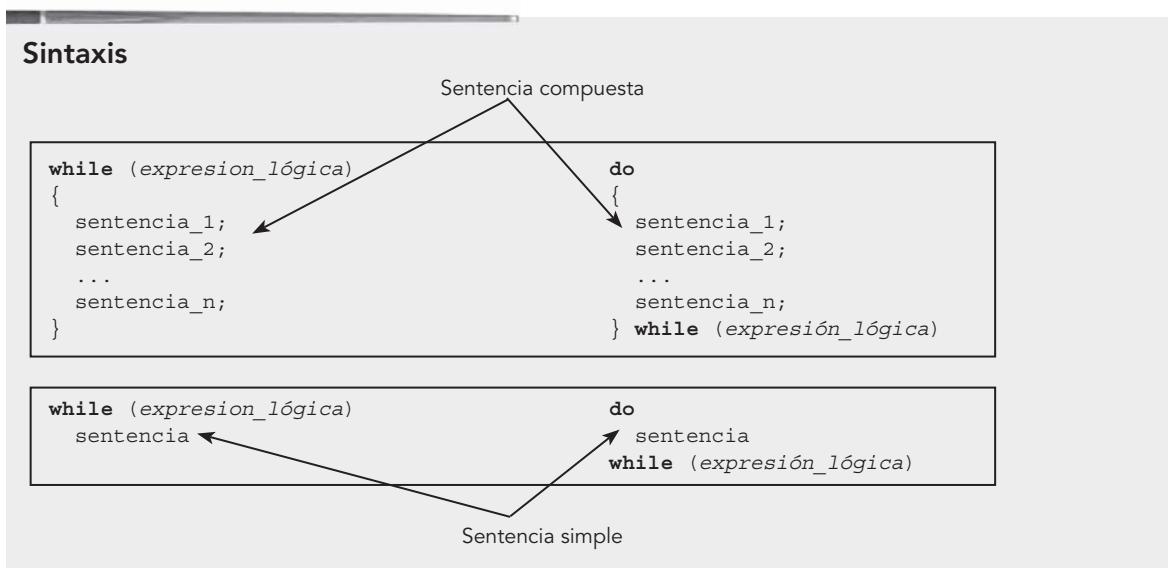
Bucle para introducir un dígito.

```
do
{
    printf("Introduzca un dígito (0-9): ");
    scanf("%c",&dígito);
} while ((dígito < '0') || ('9' < dígito));
```

- Este bucle se realiza mientras se introduzcan caracteres no dígitos y termina cuando se introduzca un carácter que sea un dígito de '0' a '9'.

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.

**Ejemplo 6.12**

Bucle do para contar de 0 a 10 (sin incluir el 10).

```
int x = 0;
do
    printf("X: %d",x++);
while (x < 10);
```

**Ejemplo 6.13**

Bucle para imprimir las letras minúsculas del alfabeto.

```
char car = 'a';
do
{
    printf("%d ",car);
    car++;
} while (car <= 'z');
```

Visualizar las potencias de 2 cuyos valores estén en el rango 1 a 1000.

Ejemplo 6.14

```
/* Realizado con while */      /* Realizado con do-while */
potencia = 1;                  potencia = 1;
while (potencia < 1000)          do
{
    printf("%d \n",potencia);    printf("%d \n",potencia);
    potencia * = 2;             potencia * = 2;
} /* fin de while */ }          { 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 verdadera; 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 cuándo se usa cada uno de los tres bucles. En C, el bucle `for` es el que se utiliza con más frecuencia 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.

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 bucles* necesita tres puntos a considerar:

1. El *cuerpo* del bucle.
2. Las sentencias de *inicialización*.
3. Las condiciones para la *terminación* del bucle.

Tabla 6.1 Formatos de los bucles.

while	El uso más frecuente es cuando la repetición no está controlada por un 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.
for	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.
do-while	Es adecuado para asegurar que al menos se ejecute el bucle una vez.

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 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%c", &numero);
    suma += numero;
    printf("¿Existen más números? (S para Si, N para No): ");
    scanf("%c", &resp);
}
```

Este método es muy tedioso para grandes listas 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, donde 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.

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 (`;`).

Muestra los valores del contador, de 0 a 4.

Ejemplo 6.15

```

/*
 Ejemplo de la sentencia nula en for.
*/
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 5; printf("i: %d\n", i++)) ;
    return 0;
}

```

Ejecución

```

i: 0
i: 1
i: 2
i: 3
i: 4

```

El bucle `for` incluye tres sentencias: la sentencia de *inicialización* establece el valor inicial del contador `i` a 0; la sentencia de *condición* comprueba `i < 5`, y la sentencia *acción* imprime el valor de `i` y lo incrementa.

11

Ejercicio 6.1

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 calculan 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 6.16

El segmento de programa siguiente visualiza una tabla de multiplicación; calcula y visualiza los productos de la forma $x * y$ para cada x en el rango de 1 a $Xultimo$ y desde cada y en el rango 1 a $Yultimo$ (donde $Xultimo$, y $Yultimo$ 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 ...
```

```
for (x = 1; x <= Xultimo; x++)
{
    for (y = 1; y <= Yultimo; y++)
    {
        int producto;
        producto = x * y;
        printf(" %d * %d = %d\n", x,y,producto); }
```

bucle externo

bucle interno

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*.

Escriba las variables de control de dos bucles anidados.

Ejemplo 6.17



Ejecución

	i	j
Externo	0	
Externo	1	
Interno		0
Externo	2	
Interno		0
Interno		1
Externo	3	
Interno		0
Interno		1
Interno		2

Ejercicio 6.2

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 ($2 * 5 - 1$). 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.

6.8 Enumeraciones

Un `enum` es un tipo definido por el programador con constantes simbólicas 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

Formato

1. `enum`
`{`
 `enumerador1, enumerador2, ... enumeradorn.`
`};`
2. `enum nombre`
`{`
 `enumerador1, enumerador2, ... enumeradorn.`
`};`

En la declaración del tipo `enum` pueden asociarse a los identificadores valores constantes en vez de la asociación que de manera predeterminada se hace (0, 1, 2 ...). Para ello se utiliza este formato:

3. `enum nombre`
`{`
 `enumerador1 = expresión_constante1,`
 `enumerador2 = expresión_constante2,`
 `...`
 `enumeradorn = expresión_constanten`
`};`

Usos típicos de `enum`.

```
enum Interruptor
{
    ENCENDIDO,
    APAGADO
};

enum Boolean
{
    FALSE,
    TRUE
};

enum
{
    ROJO, VERDE, AZUL
};
```

Ejemplo 6.18

En la última declaración se definen 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 en forma predeterminada del primer enumerador de la lista de los miembros datos es igual a 0. Obsérvese que 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, 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 `dias_semana`:

```
enum dias_semana dia;
for (dia = LUNES; dia <= DOMINGO; dia++)
{
    printf("%d ",dia);
}
```

La ejecución del bucle escribirá en pantalla los valores de las constantes del tipo enumerado: 0 1 2 3 4 5 6.

A los enumeradores se pueden asignar valores constantes o expresiones constantes durante la declaración. Por ejemplo, el tipo enumerado `Hexaedro`:

```
enum Hexaedro
{
    VERTICES = 8,
    LADOS = 12,
    CARAS = 6
}
```



Resumen

En programación es habitual tener que repetir la ejecución de una sentencia o bloques de sentencias, esta propiedad se conoce como bucle. Un **bucle** (*lazo, ciclo*) 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 el 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 ejecutan 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.

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("%lg ",n);
```

6.3 Seleccione y escriba el bucle adecuado que mejor resuelva las siguientes tareas:

- Suma de la serie $1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/50$.
- Lectura de la lista de calificaciones de un examen de historia.
- Visualizar la suma de enteros en el intervalo 11...50.

6.4 Considerar el siguiente código de programa:

```
int i = 1;
while (i <= n) {
    if ((i % n) == 0) {
        ++i;
    }
}
printf("%d \n", i);
```

- ¿Cuál es la salida si n es 0?
- ¿Cuál es la salida si n es 1?
- ¿Cuál es la salida si n es 3?

6.5 Considérese el siguiente código de programa:

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

- ¿Cuál es la salida si n es 0?
- ¿Cuál es la salida si n es 1?
- ¿Cuál es la salida si n es 3?

6.6 ¿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);
```

6.7 Escriba un programa que calcule y visualice

$$1! + 2! + 3! + \dots + (n-1)! + n!$$

donde n es un valor de un dato.

6.8 ¿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");
```

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 π (3.141592...) es muy utilizada en matemáticas. Un método sencillo de calcular su valor es:

$$\pi = 4 * \left(\frac{2}{3}\right) * \left(\frac{4}{5}\right) * \left(\frac{6}{7}\right) * \left(\frac{6}{7}\right) \dots$$

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 determine 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:

$$\begin{matrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{matrix}$$

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. Suponga 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 entre b , obteniendo el cociente q_1 y el resto r_1 .

Si $r_1 < > 0$, se divide r_1 entre b_1 , obteniendo el cociente q_2 y el resto r_2 . Si $r_2 < > 0$, se divide r_1 entre 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 pri-

mer 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 la media, la desviación media y la varianza 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`, `do-while` y `for`.

6.18 Encontrar el número mayor de una serie de números reales.

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 épsilon 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:

*

**

el número de líneas que se pueden introducir es variable.

6.27 Escribir un programa para mostrar, mediante bucles, los códigos ASCII de las letras mayúsculas y minúsculas.

6.28 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.

6.29 Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras es igual al valor del número.

6.30 Escribir un programa que visualice la siguiente salida.

1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1

6.31 Diseñar e implementar un programa que cuente el número de sus entradas que son positivos, negativos y cero.

6.32 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 2 3
1 2
1

6.33 Un carácter es un espacio en blanco si es un blanco (), 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.

6.34 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.

6.35 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.

6.36 Escribir y ejecutar un programa que invierta los dígitos de un entero positivo dado.



Funciones y recursividad

Contenido

- 7.1 Concepto de función
- 7.2 Estructura de una función
- 7.3 Prototipos de las funciones
- 7.4 Parámetros de la función
- 7.5 Funciones en línea: macros con argumentos
- 7.6 Ámbito (alcance) de una variable
- 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 utilidad
- 7.12 Visibilidad de una función
- 7.13 Compilación separada
- 7.14 Funciones recursivas
- 7.15 Recursión versus iteración
- 7.16 Recursión infinita
 - › Resumen
 - › Ejercicios
 - › Problemas

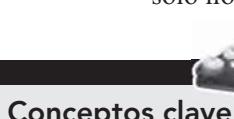
Introducción

Una *función* es un miniprograma dentro 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 clásicos como FORTRAN 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 500 funciones de bibli-



Conceptos clave

- › Biblioteca de funciones
- › Búsqueda
- › Compilación separada
- › Función
- › Iteración y recursión
- › Paso de parámetros por valor
- › Parámetros de la función
- › Paso por referencia
- › Prototipo
- › Recursión directa
- › Recursión indirecta
- › Recursividad
- › Sentencia `return`

teca, que esencialmente pertenecen a la *biblioteca estándar ANSI* (American National Standards Institute). Dado que existen tantas funciones de biblioteca, 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 la *biblioteca de funciones* del compilador o algún libro que lo incluya.

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 fracciona 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 alfabetizar 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; en el caso del programa anterior, una descomposición del programa puede ser:

```
int main()
{
    obtenercaracteres(); /* Llamada a una función que obtiene los
                           números */
    alfabetizar();        /* 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 alfabetizar()
{
    /*
        Código de C para alfabetizar 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.

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.

Una función consta de una *cabecera* que comienza con el tipo del valor devuelto por la función, seguido del nombre y argumentos de dicha función. A continuación va el *cuerpo* de la función, que es un conjunto de sentencias cuya ejecución hará que se resuelva el problema para el que está diseñada la función. Esto determina el valor particular del resultado que ha de devolverse al programa llamador. La estructura de una función en C se muestra en la figura 7.1.

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*.

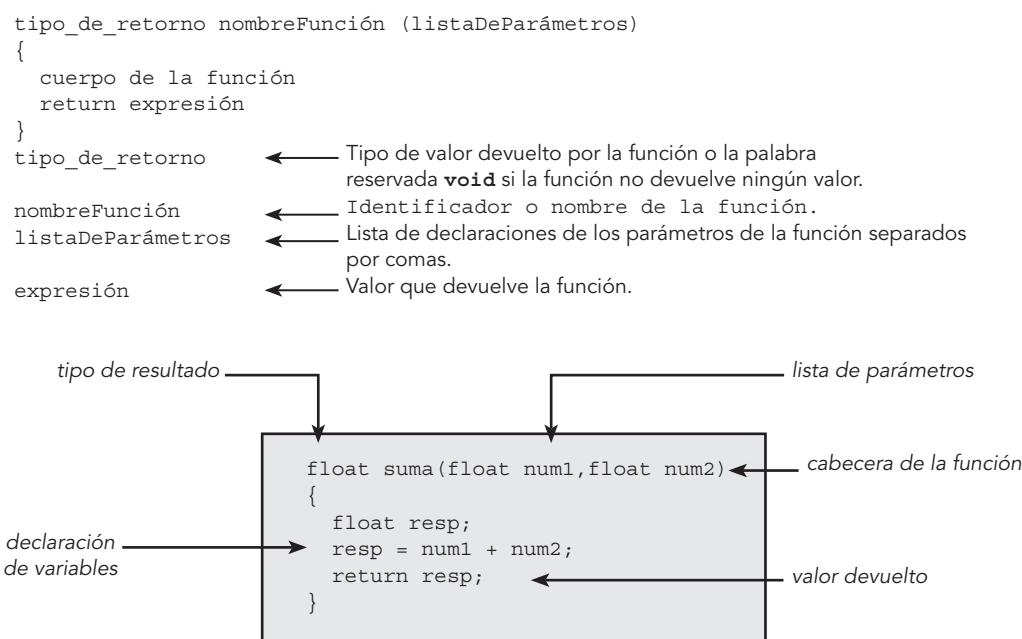


Figura 7.1 Estructura de una función.

- *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 calculado por 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 de que la ejecución de la función se haya terminado, normalmente cuando se encuentra una *sentencia return*.

Regla de programación

La sentencia `return` termina inmediatamente la función en la cual se ejecuta.

Las funciones `cuadrado()` y `suma()` muestran dos ejemplos típicos de ellas. La primera calcula y escribe los cuadrados de números enteros sucesivos a partir de un número dado (`n`), parámetro de la función, hasta obtener un cuadrado mayor de 1000. La función `suma()` calcula la suma de un número determinado (parámetro) de elementos leídos de la entrada estándar (teclado).

Obsérvese que la función `cuadrado()` no devuelve un valor, es de tipo `void`, por esa razón la sentencia final es, simplemente, `return` (se podría omitir).

```
void cuadrado(int n
{
    int 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;
}
```

La función `suma()` devuelve un valor de tipo `float`, que es la suma de los `num_elementos` datos de entrada. Por esa razón la sentencia final es `return total`.

```
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;
}
```

Ejemplo 7.1



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. 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 x, double y); /* nombre de la función media */
double MAX (int* m, int n);      /* nombre de función MAX, distinta
                                    de max */

```

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 en forma predeterminada C supone 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, como `int`, `char` o `float`, o un **apuntador¹ (puntero)** a cualquier tipo C, o un tipo `struct`.²

```

int max(int x, int y);           /* devuelve un tipo int */
double media(double* x, int n);  /* devuelve un tipo double */
float func0(int n);             /* devuelve un tipo float */
char func1(void);               /* devuelve un dato char */
int *func3(int n, int m);       /* devuelve un puntero a int */
char *func4(void);              /* devuelve un puntero a char */
int func5(void);                /* devuelve un int [es opcional] */
struct InfoPersona buscar(int n); /* devuelve una estructura */

```

Si una función no devuelve un resultado, se utiliza el tipo `void`, que se considera como un tipo de dato especial.

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);
```

Nota de programación

Si se omite un tipo de retorno para una función, como en:

```
numResultados(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 es recomendable declararla:

```
int numResultados(float Total, int longitud)
```

Resultados de una función

Una función devuelve un único valor. El resultado se muestra con una sentencia `return` cuya sintaxis es:

```

return (expresión);
return expresión;
return;

```

El valor devuelto (*expresión*) puede ser cualquier tipo de dato conocido por el lenguaje C (tipo simple, o tipo estructurado). Se pueden retornar valores múltiples devolviendo un apuntador a una estructura o un *array* (*arreglo*). 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 de la función es un apuntador. Sin embargo, si se devuelve un `int` y el tipo de retorno es un `float`, se realiza la conversión automáticamente.

¹ En el capítulo 11 se estudian los apuntadores. Es conveniente señalar que en el texto de la programación se deja puntero por cuestión de uso.

² En el capítulo 10 se estudian las estructuras o registros.

La siguiente función está mal definida, el tipo de retorno es apuntador a entero (`int*`), sin embargo devuelve un entero (`int`):

```
int* funcpuntero(void)
{
    int n;
    ...
    return n;
}
```

Una función puede tener cualquier número de sentencias `return`. Tan pronto como el programa encuentra cualquiera de las sentencias `return`, devuelve el 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`. La siguiente función no tiene tipo de retorno, tampoco tiene argumentos (parámetros):

```
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, la función principal (`main()`) 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 el 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 realice. 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, 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).

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()`.

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*.

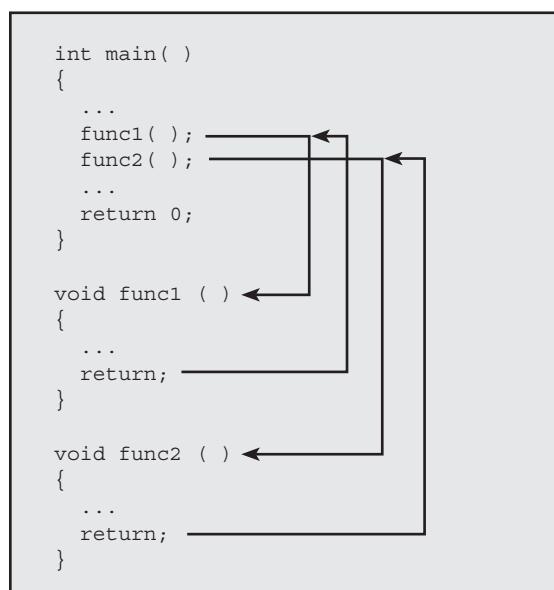


Figura 7.2 Traza de llamadas de funciones.

También se admiten paréntesis vacíos para indicar que una función no tiene argumentos (realmente significa que puede tener cualquier número de argumentos). Se recomienda utilizar `void` para indicar que una función no tiene argumentos.

```
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 el 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.

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 7.2

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;
}
```

Ejemplo 7.3

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: tipo de dato retornado, 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 sí es recomendable para que el compilador pueda hacer comprobaciones 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.

Nota de programación

C recomienda que se declare una función si es llamada antes de su definición.

Sintaxis

tipo_retorno	nombre_función	(lista_de_declaración_parámetros);
tipo_retorno		Tipo del valor devuelto por la función o palabra reservada <code>void</code> si no devuelve un valor.
nombre_función		Nombre de la función.
lista_de_declaración_parámetros		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). Palabra reservada <code>void</code> si no tiene parámetros.

Cuando un prototipo declara una función es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia. Los siguientes prototipos son válidos:

```
double FahrACelsius(double tempFahr);
int max(int x, int y);
int longitud(int h, int a);
struct persona entrad(void);
char* concatenar(char* c1, char* c2);
double intensidad(double r, double v);
```

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 es preciso tener clara la diferencia entre los conceptos *declaración* y *definición*. 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 solo 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 la 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(void); /* 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);
}

```

Declaración de una función

- Antes 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 aunque escribirlos proporciona mayor claridad.

```

char* copiar (char*, int);
char* copiar (char * buffer, int n);

```

La comprobación de tipos es una acción realizada por el compilador. El compilador conoce cuáles 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. Otro ejemplo de prototipo:

```
int procesar(int a, char b, float c, double d, char *e);
```

El compilador utiliza solo la información de los tipos de datos. Los nombres de los argumentos, aunque se requieren, 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.

```
int muestra(void);
```

Ejemplos

1. /* prototipo de la función cuadrado */

```
double cuadrado(double n);
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* nom);
void main()
{
    visualizar_nombre("Lucas El Fuerte");
}
void visualizar_nombre(char* nom)
{
    printf("Hola %s \n",nom);
}

```

Prototipos con un número no especificado de parámetros

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. Entonces, lo primero que hay que hacer es incluir dicho archivo.

```
#include <stdarg.h>
```

En el archivo está declarado el tipo va_list, un puntero (apuntador) para manejar la lista de datos pasada a la función.

```
va_list puntero;
```

La función va_start() inicializa a la variable puntero, de tal forma que referencia al primer parámetro de la lista. El prototipo que tiene:

```
void va_start(va_list puntero,ultimoijo);
```

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 es:

```
tipo va_arg(va_list puntero, tipo);
```

Donde tipo es precisamente el tipo del argumento variable que es captado en ese momento, y 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 interno preparado para siguientes llamadas. El prototipo da 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, ...)`, cuya finalidad es calcular el máximo de n argumentos de tipo `double`, donde n es el argumento fijo que se utiliza.

La función `maximo()` calcula y escribe el máximo valor de una lista variable de datos.

```
#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;
        }
    }
    printf("\t\tMáximo de la lista de %d números: %.2lf\n",n,mx);
    va_end(puntero);
}
```

7.4 Parámetros de la función

C siempre utiliza el método de *parámetros por valor* para pasar variables a funciones. Con el fin de 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 apuntador (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. Por ejemplo, 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 la siguiente llamada:

```
circulo(25, 40, vueltas*4);
```

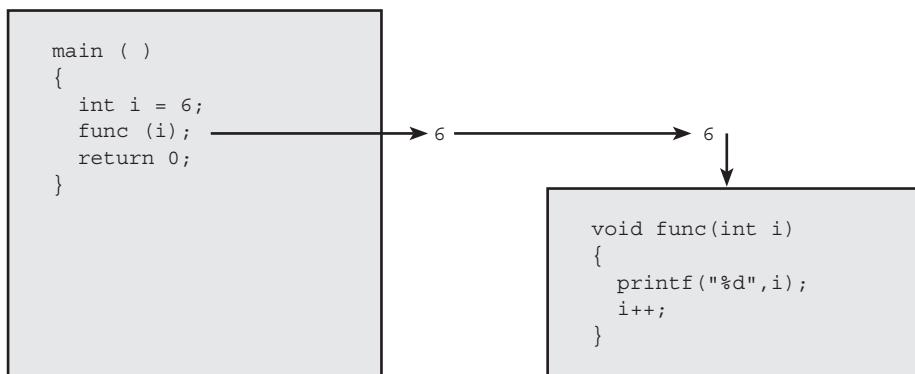


Figura 7.3 Paso de la variable *i* por valor.

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, este recibe una copia de los valores de los parámetros. Si se cambia el valor de un parámetro, 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.

Nota

El método en forma predeterminada de pasar parámetros es por valor, a menos que se pasen arreglos.³ Los arreglos se pasan siempre por dirección; se pasa la dirección o referencia al primer elemento.

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);
}

```

³ En el capítulo 8 se estudian los arreglos (*arrays*).

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
```

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, 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 apuntador (puntero).

```
float x;
int y;
entrada(&x, &y);
. .
void entrada(float* x, int* y)
```

C utiliza apuntadores (punteros) para implementar parámetros por referencia, ya que en forma pre-determinada en C el paso de parámetros es por valor.

```
/* método de paso por referencia, mediante puntero */
void intercambio(int* a, int* b)
{
    int aux = *a;
    *a = *b;
    *b = aux;
}
```

En la siguiente llamada a la función `intercambio()` esta 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 apuntador se declara poniendo el asterisco (*) antes del nombre de la variable. Las variables `p`, `r`, `q` son apuntadores a distintos tipos.

```
char* p; /* variable puntero a char */
int * r; /* variable puntero a int */
double* q; /* variable puntero a double */
```

Diferencias entre paso de variables por valor y por referencia

Las reglas que han de seguirse cuando se transmiten variables por valor y por referencia son las siguientes:

- Los parámetros valor reciben copias de los valores de los argumentos que se les pasan;
- La asignación a parámetros valor de una función nunca cambia el valor del argumento original pasado a los parámetros;
- Los parámetros para el *paso por referencia* (declarados con *, punteros) reciben la dirección de los argumentos pasados; a éstos les debe de preceder del operador &, excepto los *arrays* (*arreglos*).
- En una función, las asignaciones a parámetros referencia (apuntador) cambian los valores de los argumentos originales.

Por ejemplo, la función `potrat2()` para que cambie los contenidos de dos variables, requiere que los datos puedan ser modificados.

Paso por valor

```
double a, b;
potrat1(double x, double y)
{
...
}
```

Paso por referencia

```
double a, b;
potrat2(double* x, double* y)
{
...
}
```

Solo 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(double x, double y);
void potrat2(double* x, double* y);

void main()
{
    double a, b;
    a = 5.0; b = 1.0e2;

    potrat1(a, b);
    printf("\n a = %.1f b = %.1lf", a, b);
    potrat2(&a, &b);
    printf("\n a = %.1f b = %.1lf", a, b);
}

void potrat1(double x, double y)
{
    x = x*x;
    y = sqrt(y);
}

void potrat2(double* x, double* y)
{
    *x = (*x) * (*x);
    *y = sqrt(*y);
}
```

La ejecución del programa producirá:

```
a = 5.0 b = 100.0
a = 25.0 b = 10.0
```

Se puede observar en el programa cómo se accede a los apuntadores, con el operador * precediendo al parámetro apuntador devuelve el contenido.

Nota

Todos los parámetros en C se pasan por valor, C no tiene parámetros por referencias, hay que hacerlo con apuntadores y el operador &.

Parámetros `const` de una función

Con 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 solo 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 x, const int* y);
void f2(int x, int * const y);
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, sólo protege el valor contenido */
}
void f2(int x, int* const y)
{
    x = 10;           /* correcto */
    *y = 11;          /* correcto, ahora no se protege el contenido (*y) */
    y = &x;           /* error, protege a la variable puntero */
}
```

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:	Ítem pasado por dentro de la función	Cambia ítem	Modifica parámetros al exterior
<code>int item</code>	<i>Valor</i>	Sí	No
<code>const int item</code>	<i>Valor</i>	No	No
<code>int*item</code>	<i>Por dirección</i>	Sí	Sí
<code>const int* item</code>	<i>Por dirección</i>	<i>No su contenido</i>	No
<code>int* const item</code>	<i>Por dirección</i>	<i>No el puntero</i>	Sí

7.5 Funciones en línea, macros con argumentos

Una función normal es un bloque de código que se llama desde otra función. El compilador genera código adicional para situar la dirección de retorno en la pila interna. La dirección de retorno es:

- La sentencia desde que se invoca. Por ejemplo:
- ```
a = 2 * funcion(n,m);
```
- La dirección de la sentencia que sigue a la instrucción que llama a la función si esta llamada no forma parte de otra sentencia. Por ejemplo:

```
escribirCirculo(radio);
siguiente_sentencia;
```

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.

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 se le llama. Así la siguiente función:

```
float fesp(float x)
{
 return (x*x + 2*x -1);
}
```

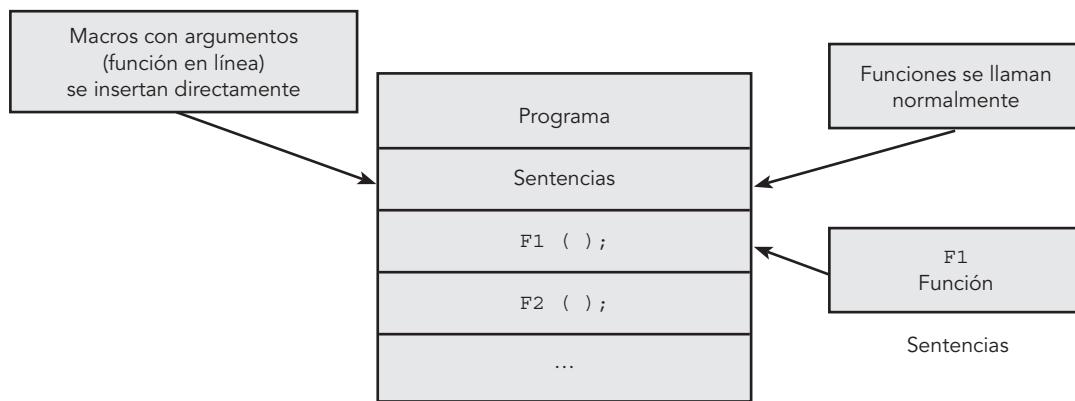


Figura 7.4 Código generado por una función fuera de línea.

puede sustituirse por la función en línea (*macro con argumentos*):

```
#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 preprocesador 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 necesita tanta memoria como la requerida 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 con una función normal, y el código de llamada suplementario es 25 bytes por cada llamada, el tamaño se incrementa en una cantidad insignificante.

### A recordar

La definición de una macro normalmente ocupa una línea. Se puede prolongar la línea con el carácter `\` al final de la línea.

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                                   |
|---------------------------------------------------------|------------------------------------------------|-----------------------------------------------|
| Macros (funciones en línea)<br>Funciones fuera de línea | Rápida de ejecutar<br>Pequeño tamaño de código | Tamaño de código grande<br>Lenta de ejecución |

## Creación de macros con argumentos

Para crear una macro con argumentos hay que utilizar la sintaxis:

```
#define NombreMacro (parámetros sin tipos) expresión_texto
```

La definición ocupará solo una línea, aunque si se necesita más texto, hay que situar una barra invertida (\) al final de la primera línea y continuar en la siguiente; en caso de ser necesarias más líneas se debe proceder de igual forma; de esa manera 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 distinto tipo. 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) de una variable

El *ámbito* o *alcance* de las variables 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 que declara a la variable en el programa determina su *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; /* m, ámbito de bloque */
 ...
}
}
```

### 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, déclarala al principio de un programa, fuera de cualquier función.

```
int g, h; /* variables globales */

int main()
{
 ...
}
```

Una variable global es visible (“*se conoce*”) 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 cuántas 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 de programación

Declare todas las variables globales 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 mismo.

## 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)
{
 ...
}
```

### A recordar

Un programa puede ocupar más de un archivo fuente. Las variables globales (de programa) son visibles en todos los archivos fuente de que consta el programa; por el contrario, las variables con ámbito en el archivo solo son visibles dentro del archivo fuente donde se definen.

## 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 */
 ...
}
```

## 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 (un *bloque* está delimitado por `{` y `}`). Una variable declarada en un bloque (variable local del bloque) sólo es visible en el interior de ese bloque. En la siguiente función, la variable `i` tiene como ámbito el bloque de la sentencia `if`:

```
float func(int j)
{
 if (j > 3)
 {
 int i;
 for (i = 0; i < 20; i++)
 func1(i);
 }
 /* aquí ya no es visible i */
}
```

Toda función determina un bloque que sigue a la *cabecera* de la función (*cuerpo* de la función). En este sentido se puede afirmar que las variables declaradas dentro de una función tienen *ámbito de bloque* de la función, o sencillamente, *ámbito de función*; no son visibles fuera del bloque. En el siguiente ejemplo, *i* es una variable local a la función, de *ámbito la función*:

```
void func1(int x)
{
 int i;
 for (i = x; i < x+10; i++)
 printf("i = %d \n", i*i);
}
```

## Variables locales

Además de tener un *ámbito* restringido, las variables locales son especiales por otra razón: existen en memoria solo 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*.

### Variables automáticas

Las variables que se declaran dentro de una función se dice que son *automáticas* (*auto*), lo cual significa que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto como se sale de dicha función. La palabra reservada *auto* es opcional.

```
auto int Total; es igual que int Total;
```

#### Nota de programación

Normalmente no se especifica (es opcional) la palabra *auto*.

### Variables externas

A veces se presenta el problema de que una función necesita utilizar una variable que *otra función* inicializa. Cómo las variables locales solo 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 con la palabra reservada *extern*. Cuando una variable se declara *externa*, se indica al compilador que la variable está definida en otro lugar.

El siguiente archivo, *extern1.c*, declara la variable global (*ámbito programa*) *f* visible en todas las funciones del archivo:

```
/* variables externas: parte 1 */
#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;
}

```

Este segundo archivo fuente, `exter2.c`, utiliza la variable *externa f (global)*:

```

/*variables externas: parte 2 */
#include <stdio.h>
void leerReal(void)
{
 extern float f; /* declara que f está definida en otro archivo */
 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.

## 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, 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 solo 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 en la memoria. Por ejemplo,

```
register int indice;
for (indice = 0; indice < 1000; indice++) ...
```

## Variables estáticas

Las variables estáticas son opuestas, en su significado, a las variables automáticas (`auto`). 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 solo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada `static`.

```

void func_uno(void)
{
 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 = 0;
 ...
 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 permite obtener la serie de números de Fibonacci. El ejercicio que se plantea es el siguiente: dado un entero  $n$ , obtener los  $n$  primeros números de la serie de Fibonacci.

La secuencia de números de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13... (cada número es la suma de los dos anteriores en la serie, excepto 0, 1, 1) se obtiene partiendo de los números 0, 1 y a partir de ellos cada número se consigue sumando los dos anteriores:

$$a_n = a_{n-1} + a_{n-2}$$

La función `fibonacci()` que se escribe a continuación, tiene dos variables `static`, `x` y `y`. Se inicializan `x` a 0 y `y` a 1; a partir de esos valores se calcula el término actual, `y`, dejando preparado `x` para la siguiente llamada. Al ser variables `static` mantienen el valor entre llamada y llamada.

```
#include <stdio.h>
long int fibonacci(void);
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(void)
{
 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 una biblioteca estándar de funciones que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con solo 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 nuestros programas se muestran a continuación encerrados entre corchetes tipo ángulo:

|            |            |            |            |
|------------|------------|------------|------------|
| <assert.h> | <ctype.h>  | <errno.h>  | <float.h>  |
| <limits.h> | <locale.h> | <math.h>   | <setjmp.h> |
| <signal.h> | <stdarg.h> | <stddef.h> | <stdio.h>  |
| <stdlib.h> | <string.h> | <time.h>   |            |

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 una 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:

- E/S estándar (para operaciones de Entrada/Salida);
- matemáticas (para operaciones matemáticas);
- rutinas estándar (para operaciones estándar de programas);
- visualizar ventana de texto;
- de conversión (rutinas de conversión de caracteres y cadenas);
- de diagnóstico (proporcionan rutinas de depuración incorporada);
- de manipulación de memoria;
- control del proceso;
- clasificación (ordenación);
- directorios;
- fecha y hora;
- de interfaz;
- diversas;
- búsqueda;
- manipulación de cadenas;
- 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 estudian 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. Casi 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.

### 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. Devuelve un valor *falso* (cero) si se pasa un carácter distinto de letra a esta función.

- `islower(c)`

Devuelve *verdadero* (distinto de cero) si `c` es una letra minúscula. Devuelve un valor *falso* (cero), si se pasa un carácter distinto de una minúscula.

**Tabla 7.3** Funciones de caracteres.

| Función             | Prueba (test) de argumento c                                                                     |
|---------------------|--------------------------------------------------------------------------------------------------|
| int isalnum(int c)  | Letra o dígito.                                                                                  |
| int isalpha(int c)  | Letra mayúscula o minúscula.                                                                     |
| int iscntrl(int c)  | Carácter de control.                                                                             |
| int isdigit(int c)  | Dígito decimal (0-9).                                                                            |
| int isgraph(int c)  | Carácter imprimible excepto <i>espacio</i> .                                                     |
| int islower(int c)  | Letra minúscula (a-z).                                                                           |
| int isprint(int c)  | Carácter imprimible, incluyendo <i>espacio</i> .                                                 |
| int ispunct(int c)  | Carácter imprimible distinto de dígito, letra y de <i>espacio</i> .                              |
| int isspace(int c)  | <i>Espacio, avance de página, nueva línea, retorno de carro, tabulación, tabulación vertical</i> |
| int isupper(int c)  | Letra mayúscula (A-Z).                                                                           |
| int isxdigit(int c) | Dígito hexadecimal.                                                                              |
| int tolower(int c)  | Devuelve la letra minúscula de c.                                                                |
| int toupper(int c)  | Devuelve la letra mayúscula de c.                                                                |

- **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.

La siguiente función comprueba que el argumento es numérico o alfabético:

- **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.

El programa solicita, y comprueba, que se teclee una letra.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
 char inicial;
 printf("¿Cuál es su primer carácter inicial?: ");
 scanf("%c%c", &inicial);
 while (!isalpha(inicial))
 {

```

```

 puts("Carácter no alfabético ");
 printf("¿Cuál es su siguiente inicial?: ");
 scanf("%c%c", &inicial);
}
puts("¡Terminado!");
return 0;
}

```

## 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:

- **iscntrl(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, devuelve *falso*.
- **isprint(c)**  
Devuelve *verdadero* si *c* es un carácter imprimible, código ASCII 21 a 127, incluyendo el *espacio*; en caso contrario, 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 el carácter *c* es un espacio, nueva línea (\n), retorno de carro (\r), tabulación (\t) o tabulación vertical (\v).

## 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.

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);

```

### Ejemplo 7.5



```

 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

Cualquier operación aritmética, por compleja que sea, es posible en un programa C. Las funciones matemáticas disponibles son las siguientes:

- funciones matemáticas de carácter general;
- trigonométricas;
- logarítmicas;
- exponentiales;
- aleatorias.

La mayoría de las funciones numéricas están en el archivo de cabecera `math.h`; las funciones de valor *absoluto* `fabs` y `labs` están definidas en `stdlib.h`, y las funciones de *división entera* `div` y `ldiv` también están en `stdlib.h`.

### Funciones matemáticas de carácter general

Las funciones matemáticas usuales en la biblioteca estándar son:

- **`ceil(x)`**  
Redondea al entero más cercano.  
Prototipo de la función: `double ceil(double x)`.
- **`fabs(x)`**  
Devuelve el valor absoluto de `x` (valor sin signo).  
Prototipo de la función: `double fabs(double x)`.
- **`floor(x)`**  
Redondea de manera predeterminada al entero más próximo.  
Prototipo de la función: `double floor(double x)`.
- **`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`.  
Prototipo de la función: `double fmod(double x, double y)`.
- **`frexp(x, exp)`**  
Devuelve la mantisa, `m` (valor sin signo en el intervalo  $[0.5, 1)$ ) del número real `x`, y asigna a `*exp` el número entero `n` tal que  $m * 2^n = x$ . En resumen, descomponen `x` en notación científica. Prototipo de la función: `double frexp(double x, int *exp)`.

- **modf(x, y)**

Descompone el número real  $x$ , en su parte decimal  $f$ , y en su parte entera  $i$ , ambos con el mismo signo que  $x$ ; de modo que devuelve  $f$  y asigna  $i$  a  $*y$ . Prototipo de la función: `double modf(double x, double *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.

Prototipo de la función: `double pow(double x, double y)`.

- **sqrt(x)**

Devuelve la raíz cuadrada de  $x$ ;  $x$  debe ser mayor o igual a cero.

Prototipo de la función: `double sqrt(double x)`.

*El programa muestra los diversos resultados que devuelven las funciones matemáticas carácter general (apartado anterior).*

**Ejemplo 7.6**

```
#include <stdio.h>
#include <math.h>
int main()
{
 int exponente;
 double d, y;
 printf("\n ceil(3.7) = %lf \t\t ceil(3.4) = %lf",
 ceil(3.7),ceil(3.4));
 printf("\n ceil(-3.7) = %lf \t\t ceil(-3.4) = %lf ",
 ceil(-3.7),ceil(-3.4));
 printf("\n floor(3.7) = %lf \t\t floor(3.4) = %lf",
 floor(3.7),floor(3.4));
 printf("\n floor(-3.7) = %lf \t\t floor(-3.4) = %lf",
 floor(-3.7),floor(-3.4));
 printf("\n fmod(5.6,2.5) = %lf \t\t fmod(-5.6,2.5) = %lf",
 fmod(5.6,2.5),fmod(-5.6,2.5));
 d = frexp(16.0, &exponente);
 printf("\n Mantisa de 16.0, m = %lf \t\t exponente = %d",
 d,exponente);
 d = modf(-17.365, &y);
 printf("\n Parte decimal de -17.365 = %lf ",d);
 printf("\n Parte entera de -17.365 = %lf ",y);
 return 0;
}
```

### Ejecución

```
ceil(3.7) = 4.000000 ceil(3.4) = 3.000000
ceil(-3.7) = -3.000000 ceil (-3.4) = -2.000000
floor(3.7) = 3.000000 floor(3.4) = 3.000000
floor(-3.7) = -4.000000 floor (-3.4) = -4.000000
fmod (5.6,2.5) = 0.600000 fmod (-5.6,2.5) = -0.600000
Mantisa de 16.0, m = 1.000000 exponente = 4
Parte decimal de -17.365 = -0.365000
Parte entera de -17.365 = -17.000000
```

## 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`.

Prototipo de la función: `double acos(double x)`.

- **asin (x)**

Calcula el arco seno del argumento `x`. El argumento `x` debe estar entre `-1` y `1`.

Prototipo de la función: `double asin(double x)`.

- **atan (x)**

Calcula el arco tangente del argumento `x`.

Prototipo de la función: `double atan(double x)`.

- **atan2 (x, y)**

Calcula el arco tangente de `x` dividido entre `y`.

Prototipo de la función: `double atan2(double x, double y)`.

- **cos (x)**

Calcula el coseno del ángulo `x`; `x` se expresa en radianes.

Prototipo de la función: `double cos(double x)`.

- **sin (x)**

Calcula el seno del ángulo `x`; `x` se expresa en radianes.

Prototipo de la función: `double sin(double x)`.

- **tan (x)**

Devuelve la tangente del ángulo `x`; `x` se expresa en radianes.

Prototipo de la función: `double tan(double x)`.

- **cosh (x)**

Calcula el coseno hiperbólico de `x`.

Prototipo de la función: `double cosh(double x)`.

- **sinh (x)**

Calcula el seno hiperbólico de `x`.

Prototipo de la función: `double sinh(double x)`.

- **tanh (x)**

Calcula la tangente hiperbólica de `x`.

Prototipo de la función: `double tanh(double x)`.

### Regla de programación

Si necesita pasar un ángulo expresando en *grados* a *radianes* para poder utilizarlos con las funciones trigonométrica, multiplique los grados por `pi/180`, donde `pi = 3.14159`.

## Funciones logarítmicas y exponenciales

Las funciones logarítmicas y exponenciales suelen ser utilizadas con frecuencia no sólo en matemáticas, sino también en el mundo de las empresas y los negocios. Estas funciones requieren también el archivo de inclusión `math.h`.

- **exp (x)**

Calcula el exponencial  $e^x$ , donde  $e$  es la base de logaritmos naturales de valor 2.718282. Prototipo de la función: `double exp (double x)`.

- **log (x)**

La función calcula el logaritmo natural (logaritmo en base  $e$ ) del argumento  $x$ ;  $x$  ha de ser positivo. Prototipo de la función: `double log (double x)`.

- **log10 (x)**

Calcula el logaritmo decimal del argumento  $x$ ;  $x$  ha de ser positivo. Prototipo de la función: `double log10 (double x)`.

## 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 de C son: **rand**, **random** (no ANSI-C), **randomize** (no ANSI-C) y **srand**. Estas funciones se encuentran 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 de incluir dicho archivo en la parte superior de su programa.

*Prototipo de la función: `int rand (void)`.*

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()**.

- **srand(semilla)**

El programa genera 10 números aleatorios. Cada vez que se ejecute el programa la secuencia de números generada será la misma.

### Ejemplo 7.7

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
 int i;
 for (i = 1; i <= 10; i++)
 printf("%d ",rand());
 return 0;
}
```

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.

*Prototipo de la función: `void srand(unsigned int semilla)`.*

- **randomize**

Los compiladores de C generalmente incorporan la macro `randomize` para inicializar el generador de números aleatorios con una semilla 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` debe incluirse en el programa. La macro no retorna valor.

El programador de C puede incorporar su propia macro `randomize`. A continuación se escribe una posible definición que llama a `srand()` con la semilla que devuelve la función `time()`:

```
#include <stdlib.h>
#include <time.h>
#define randomize (srand(time(NULL)))
```

- **random(num)**

Al igual que `randomize`, la macro `random` generalmente viene incorporada en los compiladores de C. 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`. Una definición de `random` que puede incorporar a sus programas es la siguiente:

```
#define random(num) (rand()%(num))
```

En el ejemplo 7.8 se define esta macro y se utiliza para generar números aleatorios en el rango  $[0, 1000]$ .



### Ejemplo 7.8

Programa para encontrar el mayor de 10 números aleatorios entre 0 y 1000

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

#define randomize (srand(time(NULL)))
#define random(num) (rand()%(num))
#define TOPE 1000

#define MAX(x,y) ((x)>(y)?(x):(y))

int main(void)
{
 int mx,i;
 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;
}
```

## 7.11 Funciones de utilidad

C incluye 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)**

Devuelven el valor absoluto del argumento `n`.

Su prototipo:

```
int abs(int n)
long labs(long n)
```

- **atof(cad)**

Convierte los dígitos de la cadena *cad* (ignora *espacios* iniciales) a número real.

Por ejemplo, `atof("-5.85")` devuelve el número real `-5.85`.

*Prototipo de la función:* `double atof(const char *cad)`.

- **atoi(cad), atol(cad)**

Convierte los dígitos de la cadena *cad* (ignora *espacios* iniciales) a número entero y entero largo respectivamente. Por ejemplo, `atoi("580")` devuelve el número entero 580.

Prototipo de las funciones:

```
int atoi(const char *cad);
long atol(const char *cad);
```

- **div(num, 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<sup>4</sup> `div_t`. La definición de la estructura es:

```
typedef struct
{
 int quot; /* cociente */
 int rem; /* resto */
} div_t;
```

*Prototipo de la función:* `div_t div(int num, int denom);`

El siguiente ejemplo calcula y visualiza el cociente y el resto de la división de dos enteros.

**Ejemplo 7.9**

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

int main(void)
{
 div_t resultado;
 resultado = div(19, 4);
 printf("Cociente %d", resultado.quot);
 printf("Resto %d", resultado.rem);
 return 0;
}
```

- **ldiv(num, denom)**

Calcula el cociente y resto de *num* dividido entre *denom*, y almacena los resultados de *quot* y *rem*, miembros de tipo `long` de la estructura `ldiv_t`. La definición de la estructura es:

```
typedef struct
{
 long int quot; /* cociente */
 long int rem; /* resto */
} ldiv_t;
```

Por ejemplo:

```
resultado = ldiv(1600L, 40L);
```

<sup>4</sup> Estructura es un tipo de dato que se estudia en el capítulo 10.

## 7.12 Visibilidad de una función

El *ámbito* de un elemento es su visibilidad desde otras partes del programa y la *duración* es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuándo se crea y cuándo se hace disponible. El ámbito de un elemento en C depende de dónde se sitúe la definición y de los modificadores que lo 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 7.5 resume el modo en que se ve afectado el ámbito por la posición en el archivo fuente.

Existen dos tipos 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:

- *Las variables globales tienen duración estática en forma predeterminada.* 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 desde cualquier función, a continuación del punto de definición.
- *Las variables globales están disponibles, de manera predeterminada, a otros archivos fuente.* Esta operación se denomina *enlace externo*.

## 7.13 Compilación separada

Hasta este momento, casi todos los ejemplos que se han expuesto en el capítulo se encontraban en un solo 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

`prog_demo.c`

Las variables globales declaradas en este nivel tienen ámbito global. Son válidas para todas las funciones de este archivo fuente. Disponible en otros archivos fuente a menos que se utilice la palabra reservada *static*.

`function_a()`

Las variables declaradas en este nivel son locales y tienen clase de almacenamiento *auto* al salir de la función, a menos que se utilice la palabra reservada *static*. Visible solo a esta función.

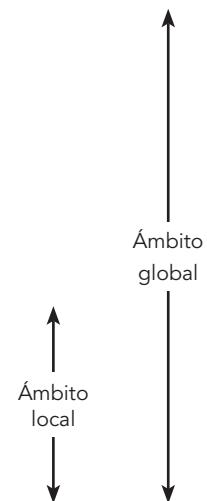


Figura 7.5 Ámbito de variable local y global.

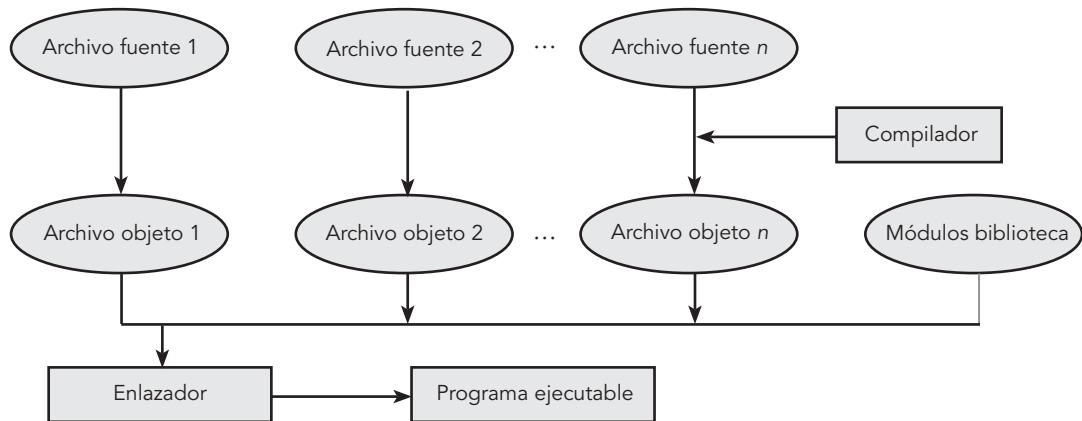


Figura 7.6 Compilación separada.

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 7.6 muestra cómo el enlazador puede construir un programa ejecutable, utilizando módulos objetos, cada uno de los cuales se obtiene compilando un módulo fuente.

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 de manera predeterminada. Si lo desea, por razones de legibilidad, puede utilizar la palabra reservada `extern` con un prototipo de función.

Se puede desear restringir la visibilidad de una función, haciéndola visible sólo a otras funciones en un archivo fuente, utilizando la palabra reservada `static` con la cabecera de la función y la sentencia del prototipo de función. Tales funciones no serán públicas al enlazador, de modo que otros módulos no tendrán acceso a ellas. Una razón para hacer esto es 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.

La palabra reservada `static`, tanto para variables globales como para funciones, es útil para evitar conflictos de nombres y prevenir su uso accidental. 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`).

### Consejo de programación

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” con relación al archivo en que está definida.

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), int g(void);
 extern int n; /* Declaración de n (no definición) */
 f(8);

```

### Ejemplo 7.10



```

n++;
g();
puts("Fin de 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.

### A recordar

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.

## 7.14 Funciones recursivas

Una función recursiva es aquella que se invoca a sí misma en forma directa o indirecta. En *recursión directa* el código de la función *f()* contiene una sentencia que invoca a *f()*, mientras que en *recursión indirecta* *f()* invoca a la función *g()*, que invoca a su vez a la función *p()*, y así sucesivamente hasta que se invoca de nuevo a la función *f()*.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva de *f(n)* debe incluir un componente base (*condición de salida*) en el que *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

Debe existir una “*forma de salir*” de la secuencia de llamadas recursivas. Así en la función de Fibonacci la condición de salida o base es *f(n) = n* para *n ≤ 1*.

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{para } n > 1$$

*F<sub>0</sub>* = 0 y *F<sub>1</sub>* = 1 constituyen el *componente base* o condiciones de salida y

*F<sub>n</sub> = F<sub>n-1</sub> + F<sub>n-2</sub>* es el componente recursivo.

En la definición recursiva del producto de dos números naturales, *a \* b*; para *b = 1*, *a \* b = a* constituye el *componente base* y el componente recursivo *a \* b = a + a \* (b - 1)*.

Un método recursivo correcto debe incluir un componente base o condición de salida ya que en caso contrario se produce una recursión infinita.

### A recordar

El diseño de una función recursiva debe contemplar un *caso base* o *condición de salida*, que se resuelve con una sentencia simple, para dejar de hacer llamadas recursivas.

Escribir, un programa que visualice los  $n$  primeros términos de la serie de Fibonacci, en donde  $n$  es un número requerido al usuario.

### Ejemplo 7.11



El *componente base* es la condición de salida; ahora se escribe la *función principal* que consiste en introducir un entero,  $n$ , mayor que 1, llamar iterativamente a la función `fibonacci( )` y escribir el valor que devuelve.

```
#include <stdio.h>
long fibonacci(int n);
int main()
{
 int n, k;
 do {
 printf("Introduzca el número de términos: ");
 scanf("%d", &n);
 }while (n <= 1);

 puts("\tSerie números de fibonacci");
 printf("0 1 "); /* componente base de la serie */

 for (k = 2; k <= n; k++)
 {
 printf("%ld%c", fibonacci(k), (k%7==0 ?'\n':' '));
 }
 return 0;
}
long fibonacci(int n)
{
 if (n == 0 || n == 1)
 return n;
 else
 return fibonacci(n - 2) + fibonacci (n - 1);
}
```

### Recursividad indirecta: funciones mutuamente recursivas

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función. El programa `alfabeto.c` visualiza el alfabeto utilizando recursión mutua o indirecta.

```
#include <stdio.h>
void funcionA(char c);
void funcionB(char c);

int main()
{
 printf("\n");
 funcionA('Z');
 printf("\n");
 return 0;
}
void funcionA(char c)
{
 if (c > 'A')
 funcionB(c);
 printf("%c", c);
}
void funcionB(char c)
{
 funcionA(--c);
}
```

El programa principal llama a la función recursiva `funcionA()` con el argumento 'z' (la última letra del alfabeto). Esta examina su parámetro *c*, si *c* está en orden alfabético después que 'A', llama a `funcionB()`, que inmediatamente invoca a `funcionA()` pasándole un parámetro predecesor de *c*. Esta acción hace que `funcionA()` vuelva a examinar *c*, y nuevamente llamar a `funcionA()` hasta que *c* sea igual a 'A' (*caso base o condición de salida*). En este momento, la recursión termina ejecutando `printf()` veintiséis veces y visualizando el alfabeto, carácter a carácter.



### Ejemplo 7.12

Determinar si un número entero positivo es par o impar, con dos funciones que se llaman mutuamente.

Para resolver el ejemplo se escriben dos funciones, `par()` e `impar()` que se llaman *mutuamente*. El programa principal invoca a `par()` pasando el entero *n* que se quiere examinar si es par o impar. Desde la función `par()` se invoca a la función `impar()` pasando *n-1*, y lo mismo desde `impar()` a `par()`. El *caso base* es *n==0*, si ocurre en el método `par()` es que el entero es par pues el número de llamadas ha sido múltiplo de 2; y de igual forma, si el *caso base* ocurre en la función `impar()` el entero es impar por lo que `par()` debe devolver `false(0)`. El programa utiliza la función `strcpy()` de la biblioteca `string.h` para copiar la cadena que informa del resultado.

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

int par(int n);
int impar(int n);

int main()
{
 int n;
 char rslt[12];

 do {
 printf("Introduzca un número: ");
 scanf("%d", &n);
 }while (n < 0);

 if (par(n))
 strcpy(rslt, " es par");
 else
 strcpy(rslt, " es impar");
 printf("\t %d %s", n, rslt);
 return 0;
}

int par(int n)
{
 if (n == 0)
 return 1;
 else
 return impar(n-1);
}

int impar(int n)
{
 if (n == 0)
 return 0; /* no es par */
 else
 return par(n-1);
}
```

## Condición de terminación de la recursión

Cuando se implementa una función recursiva es preciso considerar una *condición de terminación (caso base)*, ya que en caso contrario la función continuaría indefinidamente llamándose a sí misma y llegaría un momento en que la memoria se agotaría. En consecuencia, será necesario establecer en cualquier función recursiva la *condición de parada* de las llamadas recursivas y evitar indefinidamente las llamadas. Así, por ejemplo, en el caso de la función `factorial()`, definida anteriormente, la condición de salida (*caso base*) se da cuando el número sea 1 o 0, ya que en ambos casos el factorial es 1.

En el caso de la recursión mutua entre `par()` e `impar()` la condición de salida es que `n==0`. Además hay que considerar que cada llamada recursiva a la función se acerca a la *condición de salida*, esto se observa claramente en los ejemplos anteriores; las llamadas a `par()` o `impar()` siempre se hacen disminuyendo `n` en 1 y así se puede asegurar que `n` llegará a ser 0. En la función factorial ocurre lo mismo, cada llamada recursiva se hace con el valor `n-1`, esto asegura que el argumento llegará a ser el *caso base*, es decir, `n == 0`.

## 7.15 Recursión versus iteración

En las secciones anteriores se han estudiado varias funciones que se pueden implementar fácilmente o bien de modo recursivo o bien de modo iterativo. En esta sección compararemos los dos enfoques y examinaremos las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas a funciones. La iteración y recursión implican cada una un test de terminación (*condición de salida*). La iteración termina cuando la condición del bucle no se cumple mientras que la recursión termina cuando se reconoce un *caso base*, es decir, se alcanza la *condición de salida*.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de llamadas a funciones y en consecuencia se necesita un tiempo suplementario para realizar cada llamada.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce que se realice una nueva creación y copia de las variables de la función; esto puede consumir una considerable cantidad de memoria e incrementar el tiempo de ejecución. Por el contrario, la iteración se produce dentro de una función de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria; es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

### Nota de ejecución

Cualquier problema que se puede resolver de modo recursivo, se puede resolver también en forma iterativa (no recursivamente). Un enfoque recursivo se elige de manera normal con preferencia a un enfoque iterativo cuando el enfoque recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

### Consejo de programación

Evite utilizar recursividad en situaciones de rendimiento crítico o exigencia de mucho tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional.

### A recordar

Si una solución de un problema se puede expresar iterativa o recursivamente con igual facilidad, es preferible la solución iterativa, ya que se ejecuta más rápidamente y utiliza menos memoria.



### Ejemplo 7.13

Dado un número natural  $n$ , obtener la suma de los dígitos de que consta. De este modo se ofrece un ejemplo claro de comparación entre la resolución de modo iterativo y de modo recursivo.

Si se tiene por ejemplo  $n = 259$  la suma de los dígitos se puede expresar:

```
suma = suma(n/10) + modulo (n,10) para n > 9
suma = n para n ≤ 9 (caso base)
```

Por ejemplo, para  $n = 259$ ,

```
suma = suma(259/10) + modulo (259,10) → 2 + 5 + 9 = 16
 ↓
suma = suma(25/10) + modulo (25,10) → 2 + 5 ↑
 ↓
suma = suma(2/10) + modulo (2,10) → 2 ↑
```

### Solución recursiva

El caso base ( $n \leq 9$ ) se resuelve directamente. El componente recursivo consiste en una autollamada con argumento ( $n/10$ ).

```
int sumaRecursiva(int n)
{
 if (n <= 9)
 return n;
 else
 return sumaRecursiva(n/10) + n%10;
}
```

### Solución iterativa

La solución iterativa se construye con un bucle mientras  $n > 9$ , repitiendo la acción de acumular el módulo de 10 y hacer  $n$  10 veces menor ( $n/10$ ). La condición de salida del bucle es el *caso base* de la solución recursiva.

```
int sumaIterativa(int n)
{
 int suma = 0;
 while (n > 9)
 {
 suma = suma + n%10;
 n /= 10;
 }
 return (suma+n);
}
```

### Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva solo cuando una solución iterativa sencilla no sea posible.
2. Utilícese una solución recursiva solo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a las funciones.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el costo extra (en tiempo y memoria) de la ejecución de un programa recursivo.

### Nota de programación

Una función recursiva que tiene la llamada recursiva como última sentencia (*recursión final*) puede transformarse fácilmente en iterativa reemplazando la llamada mediante un bucle cuya condición verifica el caso base.

## 7.16 Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el *caso base o condición de salida*.

En realidad la **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicha función se ejecutará hasta que la computadora agota la memoria disponible y se produce una terminación anormal del programa.

El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (condición de salida o caso base).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

Deducir cuál es la condición de salida del método `mcd()` que calcula el mayor denominador común de dos números enteros  $b_1$  y  $b_2$  (el `mcd`, máximo común divisor, es el entero mayor que divide a ambos números). Codificar la función recursiva `mcd(m,n)` y un programa que llame a la función con números de entrada.

Supongamos dos números 6 y 124; el procedimiento clásico de cálculo del `mcd` es la obtención de divisiones sucesivas entre ambos números (124 entre 6) si el resto no es 0, se divide el número menor (6, en el ejemplo) por el resto (4, en el ejemplo) y así sucesivamente hasta que el resto sea 0.

$$\begin{array}{r} 124 \\ 04 \end{array} \left| \begin{array}{r} 6 \\ 20 \end{array} \right.$$

$$\begin{array}{r} 6 \\ 2 \end{array} \left| \begin{array}{r} 4 \\ 1 \end{array} \right.$$

$$\begin{array}{r} 4 \\ 0 \end{array} \left| \begin{array}{r} 2 \\ 2 \end{array} \right. \quad (mcd = 2)$$

$$\begin{array}{r} 20 \\ 124 \\ \hline 4 \end{array} \left| \begin{array}{r} 1 \\ 6 \\ 4 \\ 2 \\ \hline 2 \\ 0 \end{array} \right. \quad mcd = 2$$

### Ejemplo 7.14

En el caso de 124 y 6, el `mcd` es 2. En consecuencia la *condición de salida* es que el resto sea cero. El algoritmo del `mcd` entre dos números  $m$  y  $n$  es:

- `mcd(m, n)` es  $n$  si  $n \leq m$  y  $n$  divide a  $m$  (el resto es cero)
- `mcd(m, n)` es `mcd(n, m)` si  $m < n$
- `mcd(m, n)` es `mcd(n, resto de  $m$  dividido entre  $n$ )` en caso contrario.

Los pasos anteriores significan: el `mcd` es  $n$  si  $n$  es el número más pequeño y  $n$  divide a  $m$ . Si  $m$  es el número más pequeño, entonces la determinación del `mcd` debe ejecutarse con los argumentos transpuestos. Por último si  $n$  no divide a  $m$ , el resto se obtiene encontrando el `mcd` de  $n$  y el resto de  $m$  dividido entre  $n$ .

El programa que incorpora la función recursiva `mcd()` tiene como datos de entrada los dos números enteros.

```
#include <stdio.h>
int mcd(int m, int n);
int main()
{
```

```

int m,n;
do {
 printf("Introduzca dos enteros positivos: ");
 scanf("%d %d", &m, &n);
}while ((n <= 0) || (m <= 0));

printf("\nEl máximo común divisor es %d: ", mcd(m,n));
return 0;
}
int mcd(int m, int n)
{
 if (n <= m && m%n == 0)
 return n;
 else if (m < n)
 return mcd(n, m);
 else
 return mcd(n, m%n);
}

```

Al ejecutarse el programa se produce la siguiente salida:

Introduzca dos enteros positivos: 1265 2970  
El máximo común divisor es: 55



## Resumen

Las funciones son la base de la construcción de programas en C. Se utilizan para subdividir problemas grandes en tareas más pequeñas. El encapsulamiento de las características en funciones hace más fácil mantener los programas. 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 apuntadores;
- el modificador `const` se utiliza cuando se desea que los *parámetros de la función* sean valores de solo 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;
- etcétera.

Tenga cuidado con 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.
- El 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 registro* se pueden utilizar cuando se desea aumentar la velocidad de procesamiento de ciertas variables.

Una función es recursiva si tiene una o más sentencias que son llamadas a sí mismas. La recursividad puede ser directa o indirecta, esta última ocurre cuando la función `f()` llama a `p()` y esta a su vez llama a `f()`. Ciertos

problemas matemáticos tienen una definición recursiva, como por ejemplo el factorial de un número; la resolución en una computadora se realiza fácilmente mediante funciones recursivas; la recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos. Los aspectos más importantes que se deben tener en cuenta en el diseño y construcción de funciones recursivas son los siguientes:

- Un algoritmo recursivo correspondiente con una función normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema en los que este se soluciona sin ninguna llamada recursiva sino con una sentencia simple. De otro modo, una función recursiva debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, es el *caso base*, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante una función recursiva. De igual modo aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Toda función recursiva puede ser transformada en otra función con esquema iterativo.

- Las funciones con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de *overflow*.
- Cuando se codifica una función recursiva se debe comprobar siempre que tiene una condición de terminación, es decir que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de una función recursiva es correcto se deben cumplir las siguientes tres condiciones:
  1. No existir recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y esta conducir a otra, y así sucesivamente; pero cada llamada debe aproximarse más a la condición de terminación.
  2. Para la condición de terminación, la función devuelve el valor correcto para ese caso.
  3. Una función puede tener más de una llamada recursiva; cada llamada debe alcanzar la condición de terminación y devolver el valor que le corresponde al caso base.

## 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 *Dígito* 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 una función.

7.8 Escribir un programa que calcule los N primeros números naturales *primos*.

7.9 Convierta la siguiente función iterativa en recursiva. La función calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando la serie

$$1 + 1/1 ! + 1/2 ! + \dots + 1/n !$$

hasta que los términos adicionales no afecten a la aproximación

```
double loge(void)
{
 double enl, delta, fact;
 int n;
 enl = fact = delta = 1.0;
 n = 1;
 do
 {
 enl += delta;
 n++;
 }
```

```

fact *= n;
delta = 1.0 / fact;
} while (enl != enl + delta);
return enl;
}

```

7.10 Explique por qué la siguiente función puede producir un valor incorrecto cuando se ejecute:

```

long factorial (long n)
{
 if (n == 0 || n == 1)
 return 1;
 else
 return n * factorial (--n);
}

```

7.11 ¿Cuál es la secuencia numérica generada por la función recursiva  $f()$  en el listado siguiente si la llamada es  $f(5)$ ?

```

long f(int n)
{
 if (n == 0 || n == 1)
 return 1;
 else
 return 3 * f(n - 2) + 2 * f(n - 1);
}

```

7.12 Escribir una función recursiva de prototipo `int vocales(char * cd)` para calcular el número de vocales de una cadena.

7.13 Proporcionar funciones recursivas que representen los siguientes conceptos:

- El producto de dos números naturales.
- El conjunto de permutaciones de una lista de números.

7.14 Suponer que la función  $G$  está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \frac{1}{G(x-y+1)} & \text{si } x \geq y \\ 2x-y & \text{si } x < y \end{cases}$$

donde  $x$  y  $y$  son enteros positivos. Encontrar el valor de:

- $G(8, 6)$ ;
- $G(100, 10)$ .

7.15 Escribir una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

$$\begin{aligned} A(m, n) &= n + 1 & \text{si } m = 0 \\ A(m, n) &= A(m - 1, 1) & \text{si } m > 0, y n = 0 \\ A(m, n) &= A(m - 1, A(m, n - 1)) & \text{si } m > 0, y n > 0 \end{aligned}$$

7.16 ¿Cuál es la secuencia numérica generada por la función recursiva siguiente, si la llamada es  $f(8)$ ?

```

long f(int n)
{
 if (n == 0 || n == 1)
 return 1;
 else if (n % 2 == 0)
 return 2 + f(n - 1);
 else
 return 3 + f(n - 2);
}

```

7.17 ¿Cuál es la secuencia numérica generada por la función recursiva siguiente?

```

int f(int n)
{
 if (n == 0)
 return 1;
 else if (n == 1)
 return 2;
 else
 return 2*f(n - 2) + f(n - 1);
}

```

7.18 El elemento mayor de un arreglo (*array*) entero de  $n$ -elementos se puede calcular recursivamente. Suponiendo que la función:

```
int max(int x, int y);
```

devuelve el mayor de dos enteros  $x$  y  $y$ . Definir la función:

```
int maxarray(int a[], int n);
```

que utiliza recursión para devolver el elemento mayor de  $a[]$

*Condición de parada*  $n == 1$

*Incremento recursivo:*  $\text{maxarray} = \text{max}(\text{max}(a[0] \dots a[n-2]), a[n-1])$

7.19 Escribir una función recursiva,

```
int product(int v[], int b);
```

que calcule el producto de los elementos del *array*  $v[]$  mayores que  $b$ .

7.20 El ejercicio 7.14. tiene una función recursiva, escribir de nuevo la función eliminando la recursividad.

## Problemas

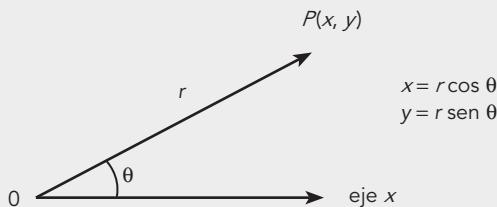
- 7.1 Escribir un programa que solicite del usuario un carácter y que lo sitúe en el centro de la pantalla. El usuario debe poner a continuación desplazar el carácter pulsando las letras A (arriba), B (abajo), I (izquierda), D (derecha) y F (fin) para terminar.
- 7.2 Escribir una función que reciba una cadena de caracteres y la devuelva en forma inversa (“hola” 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 un 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 *mcd* (máximo común divisor) de dos números por el algoritmo de Euclides. (Dividir  $a$  entre  $b$ , se obtiene el co-

ciente  $q$  y el resto  $r$  si es cero  $b$  es el *mcd*, si no se divide  $b$  entre  $r$ , y así sucesivamente hasta encontrar un resto cero; el último divisor es el *mcd*). La función *mcd* ( ) 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.

$$\frac{m!}{n!} = \frac{m!}{n!(m-n)!} \quad \text{donde}$$

$$m! = \begin{cases} 1 & \text{si } m = 0 \\ 1.2.3 \cdots m & \text{si } m > 0 \end{cases}$$

- 7.12 Escribir una función que permita calcular la serie:

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n * (n+1) * (2n+1)}{6}$$

- 7.13 Escribir un programa que lea dos números  $x$  y  $n$  y en una función calcule la suma de la progresión geométrica:

$$1 + x + x^2 + x^3 + \cdots + x^n$$

- 7.14 Escribir un programa que encuentre el valor mayor, el valor menor y la suma de los datos de la 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:

$$x^5 (e^{2x} - 1)$$

- 7.16 Escribir una función con dos parámetros,  $x$  y  $n$ , que devuelva lo siguiente:

$$\begin{aligned} x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2} & \quad \text{si } x \geq 0 \\ \frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} & \quad \text{si } x < 0 \end{aligned}$$

- 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.

$$\text{Área} = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{donde}$$

$$p = \frac{a+b+c}{2}$$

- 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 de 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  |

Compruebe su programa para los siguientes datos:  
LXXXVI (86), CCCXIX (319), MCCLIV (1254)

- 7.20 Escribir 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 muestra el valor de todas las funciones trigonométricas correspondientes al mismo.



# Arreglos (arrays), listas y tablas. Cadenas

## Contenido

- 8.1 Arreglos (arrays)
- 8.2 Inicialización de un arreglo (array)
- 8.3 Arreglos multidimensionales
- 8.4 Utilización de arreglos como parámetros
- 8.5 Concepto de cadena
- 8.6 Lectura de cadenas
- 8.7 La biblioteca `string.h`
- 8.8 Arreglos y cadenas como parámetros de funciones

## 8.9 Asignación de cadenas

- 8.10 Longitud y concatenación de cadenas
- 8.11 Comparación de cadenas
- 8.12 Conversión de cadenas a números
  - › Resumen
  - › Ejercicios
  - › Problemas

## Conceptos clave

- › Apuntador a `char`
- › Arreglo (array)
- › Arreglo de caracteres
- › Arreglos bidimensionales
- › Arreglos como parámetros
- › Arreglos multidimensionales
- › Biblioteca `string.h`
- › Cadena
- › Cadena de texto
- › Cadena vacía
- › Carácter nulo
- › Comparación
- › Concatenación
- › Conversión
- › Declaración de un arreglo
- › Literal de cadena
- › Lista, tabla
- › Valores índice o subíndice
- › Variable cadena

## Introducción

En capítulos anteriores se han descrito las características de los tipos de datos básicos o simples (carácter, entero y coma flotante). Asimismo, ha aprendido a definir y utilizar constantes simbólicas utilizando `const`, `#define` y el tipo `enum`. Numerosos problemas de programación exigen procesar una colección de valores relacionados entre sí. El procesamiento de estos conjuntos de datos se realiza con la estructura de datos básica **arreglo (array)**, concepto matemático de vector y matriz.

En este capítulo aprenderá el concepto y tratamiento de los arreglos. Un arreglo almacena muchos elementos del mismo tipo, como 20 enteros, 50 números de coma flotante o 15 caracteres. El arreglo es muy trascendental 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 arreglo, se puede crear una variable que contenga un grupo de caracteres.

Asimismo, en este capítulo se estudiará el tema de las cadenas. Una cadena se considera como un arreglo unidimensional de tipo `char` o `unsigned char`. Los temas que se analizarán son los siguientes: 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); inversión de los caracteres de una cadena.

## 8.1 Arreglos (arrays)<sup>1</sup>

Un arreglo, lista o tabla es una secuencia de datos del mismo tipo. Los datos se llaman elementos del arreglo y se numeran consecutivamente 0, 1, 2, 3, etc. El tipo de elementos almacenados en el arreglo puede ser cualquier tipo de dato de C, incluyendo estructuras definidas por el usuario, como se describirá más tarde. Normalmente el arreglo se utiliza para almacenar tipos como `char`, `int` o `float`.

Un arreglo 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 17 comunidades autónomas españolas. Cada item del arreglo se denomina *elemento*.

Los elementos de un arreglo se numeran, como ya se ha comentado, consecutivamente 0, 1, 2, 3... Estos números se denominan *valores índice* o *subíndice* del arreglo. El término "subíndice" se utiliza ya que se especifica igual que en matemáticas, como una secuencia como  $a_0, a_1, a_2, \dots$ . Estos números localizan la posición del elemento dentro del arreglo, proporcionando acceso directo al arreglo.

Si el nombre del arreglo 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 arreglo tiene  $n$  elementos, sus nombres son `a[0], a[1], \dots, a[n-1]`. La figura 8.1 representa el arreglo `a` de seis elementos; la descripción del mencionado arreglo es la siguiente:

El arreglo `a` tiene 6 elementos: `a[0]` contiene 25.1, `a[1]`, 34.2, `a[2]`, 5.25, `a[3]`, 7.45, `a[4]`, 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 arreglo se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

|                |      |      |      |      |      |      |
|----------------|------|------|------|------|------|------|
| <code>a</code> | 25.1 | 34.2 | 5.25 | 7.45 | 6.09 | 7.54 |
|                | 0    | 1    | 2    | 3    | 4    | 5    |

Figura 8.1 Arreglo (array) de seis elementos.

En C los índices de un arreglo siempre tienen como límite inferior 0, como índice superior el tamaño del arreglo menos 1.

### Declaración de un arreglo o array

Al igual que con cualquier tipo de variable, se debe declarar un arreglo antes de utilizarlo. Un arreglo se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el tamaño o longitud del arreglo. Para indicar al compilador el tamaño o longitud del arreglo se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La sintaxis para declarar un arreglo de una dimensión determinada es:

```
tipo nombreArray [numeroDeElementos];
```

Por ejemplo, para crear un arreglo (lista) de diez variables enteras, se escribe:

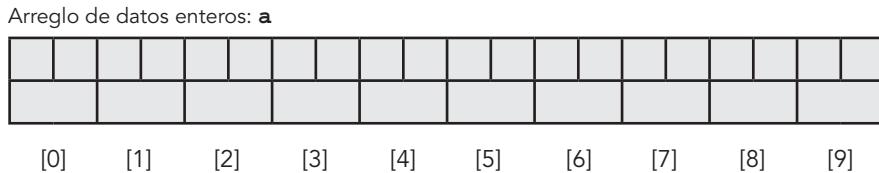
```
int numeros[10];
```

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 arreglo de diez enteros ocupa 20 bytes de memoria. La figura 8.2 muestra el esquema de un arreglo de diez elementos; cada elemento puede tener su propio valor.

Se puede acceder a cada elemento del arreglo utilizando un índice en el nombre del arreglo. Por ejemplo,

```
printf("%d \n", numeros[4]);
```

<sup>1</sup> En muchos países de Latinoamérica, el término inglés *array* se traduce por **arreglo**, mientras que en España, normalmente se conserva el término en inglés.



Un arreglo de enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza dos bytes. Se accede a cada elemento de arreglo mediante un índice que comienza en cero. Así, el elemento quinto (a [4]) del arreglo ocupa los bytes 9º. y 10º.

**Figura 8.2** Almacenamiento de un arreglo en memoria.

visualiza el valor del elemento 5 del arreglo. Los arreglos siempre comienzan en el elemento 0. Así pues, el arreglo `numeros` contiene los siguientes elementos individuales:

|                         |                         |                         |                         |
|-------------------------|-------------------------|-------------------------|-------------------------|
| <code>numeros[0]</code> | <code>numeros[1]</code> | <code>numeros[2]</code> | <code>numeros[3]</code> |
| <code>numeros[4]</code> | <code>numeros[5]</code> | <code>numeros[6]</code> | <code>numeros[7]</code> |
| <code>numeros[8]</code> | <code>numeros[9]</code> |                         |                         |

Si por ejemplo, se quiere crear un arreglo 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 arreglo 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.

## Subíndices de un arreglo

El índice de un arreglo se denomina, con frecuencia, *subíndice del arreglo*. El término procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

|                                  |                   |                         |
|----------------------------------|-------------------|-------------------------|
| <code>numeros<sub>0</sub></code> | <i>equivale a</i> | <code>numeros[0]</code> |
| <code>numeros<sub>3</sub></code> | <i>equivale a</i> | <code>numeros[3]</code> |

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 arreglo 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 arreglos y apuntadores.

### Ejemplos

```
int edad[5];
Arreglo edad contiene 5 elementos: el primero, edad[0]
y el último, edad[4].
int pesos[25], longitudes[100];
Declara 2 arreglos de enteros.
float salarios[25];
Declara un arreglo de 25 elementos float.
double temperaturas[50];
Declara un arreglo de 50 elementos double.
char letras[15];
Declara un arreglo de caracteres.
#define MX 120
char buffer[MX+1];
Declara un arreglo de caracteres de tamaño MX+1, el pri-
mer elemento es buffer[0] y el último buffer[MX].
```

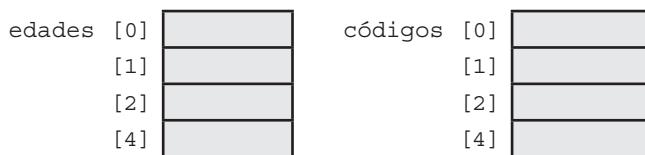


Figura 8.3 Almacenamiento en memoria de arreglos.

En los programas se pueden referenciar elementos del arreglo 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]
```

### Almacenamiento en memoria de los arreglos (arrays)

Los elementos de los arreglos se almacenan en bloques contiguos. Así, por ejemplo, los arreglos,

```
int edades[5];
char códigos[5];
```

se representan gráficamente en memoria en la figura 8.3.

#### A recordar

Todos los subíndices de los arreglos comienzan con 0. Esta forma de indexar se denomina indexación basada en cero.

#### Precaución

C permite asignar valores fuera de rango a los subíndices. Se debe tener cuidado con no hacer esta acción, debido a que sobrescribirían datos o código.

Los arreglos de caracteres funcionan de igual forma que los numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte. Así, por ejemplo, un arreglo llamado `nombre` se puede representar en la figura 8.4.

#### A recordar

En las cadenas de caracteres el sistema siempre inserta un último carácter (nulo) para indicar fin de cadena.

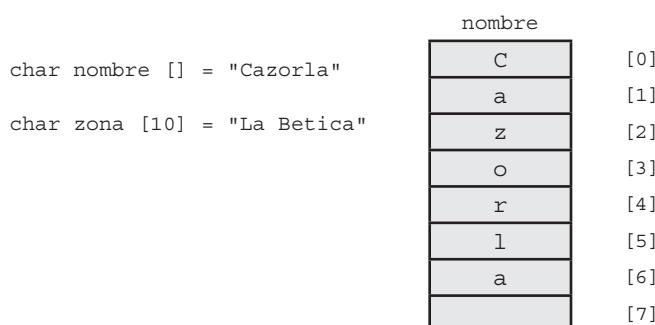


Figura 8.4 Almacenamiento de un arreglo de caracteres en memoria.

## El tamaño de los arreglos

El operador `sizeof` devuelve el número de bytes necesarios para contener su argumento. Si se usa `sizeof` para solicitar el tamaño de un arreglo, esta función devuelve el número de bytes reservados para el arreglo completo.

Por ejemplo, suponiendo que se ha declarado un arreglo de enteros de 100 elementos denominado `edades`; si se desea conocer el tamaño del arreglo, se puede utilizar una sentencia similar a:

```
n = sizeof(edades);
```

donde `n` tomará el valor 200 en el supuesto de que el tipo `int` ocupe 2 bytes.

## Verificación del rango del índice de un arreglo

C, al contrario que otros lenguajes de programación, por ejemplo, Pascal, no verifica el valor del índice de la variable que representa al arreglo. Así, por ejemplo, en Pascal si se define un arreglo `a` con índices 0 a 5, entonces `a[6]` hará que el programa se "rompa" en tiempo de ejecución.

Paso a una función un arreglo y como segundo argumento el tamaño o número de elementos.

### Ejemplo 8.1

```
double suma(const double a[], const int n)
{
 int i;
 double S = 0.0;
 for (i = 0; i < n; i++)
 S += a[i];
 return S;
}
```

## 8.2 Inicialización de un arreglo (array)

Se deben asignar valores a los elementos del arreglo antes de utilizarlos, como se asignan valores a variables. Para asignar valores a cada elemento del arreglo 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 arreglo contiene muchos elementos. El método utilizado, normalmente, es inicializar el arreglo completo en una sola sentencia.

Cuando se inicializa un arreglo, el tamaño de este 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 arreglo `numeros` tiene 6 elementos, `n` tiene 3 elementos y el arreglo `c` tiene 4 elementos.

En C los arreglos 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 arreglo de caracteres (una variable cadena) con una constante cadena.

```
char s[] = "Puesta del Sol";
```

### Precaución

C puede dejar los corchetes de un arreglo vacíos sólo cuando se asignan valores iniciales al arreglo, como

```
int cuenta [] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos a `cuenta`.

El método de inicializar arreglos mediante valores constantes después de su definición es adecuado cuando el número de elementos del arreglo es pequeño. Por ejemplo, para inicializar un arreglo (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 FEB 28
#define MAR 31
...
int meses[12] = {ENE, FEB, MAR, ABR, MAY, JUN,
 JUL, AGO, SEP, OCT, NOV, DIC};
```

Pueden asignarse valores a un arreglo utilizando un bucle `for` o `while/do-while`, y este suele ser el sistema más empleado normalmente. Por ejemplo, para inicializar todos los valores del arreglo `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 arreglo `numeros` se inicializa y establece a cero.



### Ejemplo 8.2

El siguiente programa 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 arreglos se inicializan a 0 en forma predeterminada. Por ello, la ejecución del siguiente programa visualiza 0 para los 10 valores del arreglo:

```
int lista[10];
int main()
{
 int j;
 for (j = 0; j <= 9; j++)
 printf("\n lista[%d] = %d", j, lista[j]);
 return 0;
}
```

Así, por ejemplo, en

```
int Notas[5];
void main()
{
 static char Nombres[5]; /* inicializado al carácter nulo */
}
```

El arreglo de enteros se ha definido globalmente y el arreglo de caracteres se ha definido como un arreglo local estático de `main()`.

### A recordar

Si se define un arreglo globalmente o un arreglo estático y no se proporciona ningún valor de inicialización, el compilador inicializará el arreglo con un valor en forma predeterminada (cero para arreglos de elementos enteros y reales, coma flotante, y carácter nulo para arreglos de caracteres).

Rellenar los elementos de un arreglo con números reales positivos procedentes del teclado.

### Ejemplo 8.3

```
#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.

Visualizar el arreglo `muestra` después de introducir datos en el mismo, separándolos con el tabulador.

### Ejemplo 8.4

```
#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 leidos del teclado: ");
for (i = 0, i < MAX; ++i)
 printf("%f\t", muestra[i]);
}

```

### 8.3 Arreglos multidimensionales

Los arreglos vistos anteriormente se conocen como arreglos *unidimensionales* (una sola dimensión) y se caracterizan por tener un solo subíndice. Estos arreglos se conocen también por el término *listas*. Los arreglos *multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arreglos más usuales son los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear arreglos de tantas dimensiones como requieran sus aplicaciones, esto es, tres, cuatro o más dimensiones.

Un arreglo de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (figura 8.5).

Obsérvese que en el arreglo bidimensional de la figura 8.5 si las filas se etiquetan de 0 a  $m$  y las columnas de 0 a  $n$ , el número de elementos que tendrá el arreglo será el resultado del producto  $(m + 1) \cdot (n + 1)$ . El sistema para localizar un elemento será a través de las coordenadas representadas por su número de fila y su número de columna  $(a, b)$ . La sintaxis para la declaración de un arreglo 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][30];
int matriz[4][2];
```

#### Precaución

Al contrario que otros lenguajes, C requiere que cada dimensión esté encerrada entre corchetes. La sentencia

```
int equipos [4, 30]
```

no es válida.

|     | 0 | 1 | 2 | 3 | ... | $n$ |
|-----|---|---|---|---|-----|-----|
| 0   |   |   |   |   |     |     |
| 1   |   |   |   |   |     |     |
| 2   |   |   |   |   |     |     |
| 3   |   |   |   |   |     |     |
| 4   |   |   |   |   |     |     |
| ... |   |   |   |   |     |     |
| $m$ |   |   |   |   |     |     |

Figura 8.5 Estructura de un arreglo o array de dos dimensiones.

Un arreglo de dos dimensiones en realidad es un arreglo de arreglos. Es decir, es un arreglo unidimensional, y cada elemento no es un valor entero, o de coma flotante o carácter, sino que cada elemento es otro arreglo.

Los elementos de los arreglos se almacenan en memoria de modo que el subíndice más próximo al nombre del arreglo 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 arreglo, `int tabla[4][2]`, suponiendo que cada entero ocupa 2 bytes.

**Tabla 8.1** Un arreglo bidimensional.

| Elemento     | Posición relativa de memoria |
|--------------|------------------------------|
| tabla [0][0] | 0                            |
| tabla [0][1] | 2                            |
| tabla [1][0] | 4                            |
| tabla [1][1] | 6                            |
| tabla [2][0] | 8                            |
| tabla [2][1] | 10                           |
| tabla [3][0] | 12                           |
| tabla [3][1] | 14                           |

## Inicialización de arreglos multidimensionales

Los arreglos 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 más 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}
 };
2. int tabla2[3][4] = {
 {1, 2, 3, 4},
 {5, 6, 7, 8},
 {9, 10, 11, 12}
 };
```

| Tabla [2] [3] |   | 0  | 1  | 2  | Columna |
|---------------|---|----|----|----|---------|
| Fila          | 0 | 51 | 52 | 53 |         |
|               | 1 | 54 | 55 | 56 |         |

| Tabla [3] [4] |   | 0 | 1  | 2  | 3  | Columna |
|---------------|---|---|----|----|----|---------|
| Fila          | 0 | 1 | 2  | 3  | 4  |         |
|               | 1 | 5 | 6  | 7  | 8  |         |
|               | 2 | 9 | 10 | 11 | 12 |         |

**Figura 8.6** Tablas de dos dimensiones.

### Consejo de programación

Los arreglos multidimensionales (a menos que sean globales) no se inicializan a valores específicos a menos que se les asigne 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 arreglo multidimensional, utilice una sentencia como ésta:

```
flota ventas [3] [4] = {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.};
```

### Acceso a los elementos de los arreglos bidimensionales

Se puede acceder a los elementos de *arreglos bidimensionales* de igual forma que a los elementos de un arreglo 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> [indice fila] [indice columna] = valor elemento;
```

extracción de elementos

```
<variable> = <nombre array> [indice fila] [indice columna];
```

Algunos ejemplos de asignación de valores:

```
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];
```

### Lectura y escritura de elementos de arreglos bidimensionales

Las funciones de entrada o salida se aplican de igual forma a los elementos de un arreglo 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])
```

|       |         |
|-------|---------|
| tabla | [0] [0] |
|       | [0] [1] |
|       | [0] [2] |
|       | [0] [3] |
|       | [1] [0] |
|       | [1] [1] |
|       | [1] [2] |
|       | [1] [3] |
|       | [2] [0] |
|       | [2] [1] |
|       | [2] [2] |
|       | [2] [3] |

Figura 8.7 Almacenamiento en memoria de tabla [3] [4].

```

 puts("VERDADERO");
else
 puts("FALSO");

```

## Acceso a elementos mediante bucles

Se puede acceder a los elementos de arreglos bidimensionales mediante bucles anidados, el bucle externo para el acceso a las filas y el bucle interno para las columnas. Su sintaxis es:

```

int IndiceFila, IndiceCol;
for (IndiceFila = 0; IndiceFila < NumFilas; ++IndiceFila)
 for (IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
 Procesar elemento[IndiceFila] [IndiceCol];

```

Definir una tabla de discos, llenar la tabla con datos de entrada y mostrarla por pantalla.

### Ejemplo 8.5

```

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 Precio euros: %.1f \n", discos[fila][col]);
 }
}

```

### Ejercicio 8.1

Lectura y visualización de un arreglo de dos dimensiones.

La función leer () lee un arreglo (una tabla) de dos dimensiones y la función visualizar () presenta la tabla en la pantalla.

```

#include <stdio.h>
/* prototipos funcionales*/
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, 5 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, 5 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 |

### Arreglo 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 arreglo de tres dimensiones es imaginar un cubo tal como se muestra en la figura 8.8.

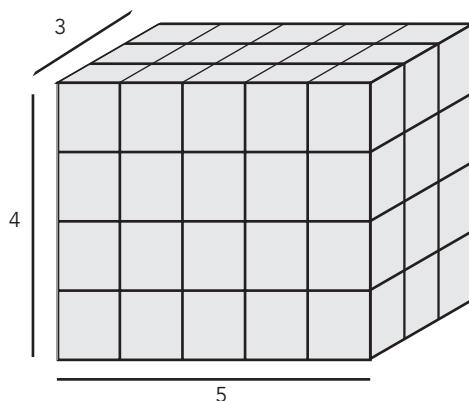


Figura 8.8 Un arreglo de tres dimensiones ( $4 \times 5 \times 3$ ).

Un arreglo tridimensional se puede considerar como un conjunto de arreglos 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 arreglo tridimensional `equipos` es:

```
int equipos[3][15][10];
```

Un ejemplo típico de un arreglo de tres dimensiones es el modelo *libro*, en el que cada página del libro es un arreglo bidimensional construido por filas y columnas. Así, por ejemplo, cada página tiene 45 líneas que forman las filas del arreglo y 80 caracteres por línea, que forman las columnas del arreglo. Por consiguiente, si el libro tiene 500 páginas, existirán 500 planos y el número de elementos será  $500 \times 80 \times 45 = 1.800.000$ .

## Proceso de un arreglo de tres dimensiones

El arreglo *libro* tiene tres dimensiones `[PAGINAS] [LINEAS] [COLUMNAS]`, que definen el tamaño del arreglo. El tipo de datos del arreglo 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 arreglo.

```
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
#define randomize (srand (time (NULL)))
#define random(num) (rand ()% (num))
void gen_mat(int a[][N], int n);
int simetrica(int a[][N], int n);
void escribe_mat(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 gen_mat(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 escribe_mat(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' : ' '));
 }
}

```

## 8.4 Utilización de arreglos como parámetros

En C *todos* los arreglos se pasan por referencia (dirección). Esto significa que cuando se llama a una función y se utiliza un arreglo como parámetro, se debe tener cuidado de no modificar los arreglos en la función llamada. C trata automáticamente la llamada a la función como si hubiera situado el operador de dirección & delante del nombre del arreglo. La figura 8.9 ayuda a comprender el mecanismo. Dadas las declaraciones.

```
#define MAX 100
double datos[MAX];
```

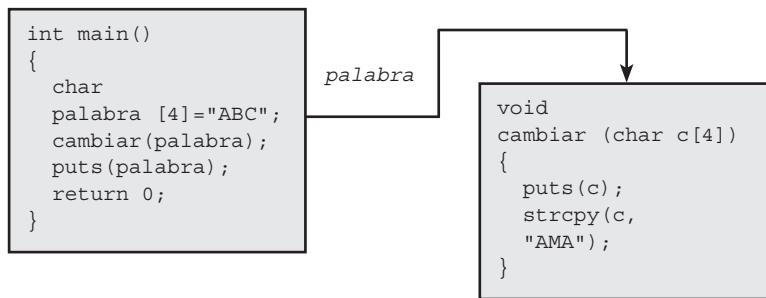


Figura 8.9 Paso de un arreglo por dirección.

se puede declarar una función que acepte un arreglo de valores `double` como parámetro. La función `SumaDeDatos()` puede tener el prototipo:

```
double SumaDeDatos(double datos[MAX]);
```

Incluso es mejor si se dejan los corchetes vacíos y se añade un segundo parámetro que indica el tamaño del arreglo:

```
double SumaDeDatos(double datos[], int n);
```

A la función `SumaDeDatos` se la llamará con argumentos de tipo arreglo junto con un entero `n`, que informa a la función sobre cuántos valores contiene el arreglo. Por ejemplo, esta sentencia visualiza la suma de valores de los datos del arreglo:

```
printf("\nSuma = %lf", SumaDeDatos(datos, MAX));
```

La función `SumaDeDatos` no es difícil de escribir. Un simple bucle `for` suma los elementos del arreglo 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 arreglo a una función incluye el tipo de elemento del arreglo y su nombre. El ejemplo 8.6 incluye dos funciones que procesan arreglos. En ambas listas de parámetros, el arreglo `a [ ]` se declara de esta forma:

```
double a[]
```

El número real de elementos se pasa mediante una variable entera independiente. Cuando se pasa un arreglo a una función, se pasa realmente solo la dirección de la celda de memoria donde comienza el arreglo. Este valor se representa por el nombre del arreglo `a`. La función puede cambiar entonces el contenido del arreglo accediendo directamente a las celdas de memoria en donde se almacenan los elementos del arreglo. Así, aunque el nombre del arreglo se pasa por valor, sus elementos se pueden cambiar como si se hubieran pasado por referencia.

Paso de arreglo a funciones. En el ejemplo se lee un arreglo y se escribe.

El arreglo tiene un tamaño máximo, `L`, aunque el número real de elementos es determinado en la función `leerArray()`. El segundo argumento es, por lo tanto, un apuntador<sup>2</sup> para así poder transmitir por referencia y obtener dicho dato de la función.

```
#include <stdio.h>
#define L 100
```

### Ejemplo 8.6



<sup>2</sup>Los apuntadores se estudian en el capítulo 11.

```

void leerArray(double a[], int* num);
void imprimirArray (const double [], int n);
int main()
{
 double a[L];
 int n;
 leerArray(a, &n);
 printf("El arreglo 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 arreglo tiene tres elementos:

0: 31.31
1: 15.25
2: 44.77

```



### Ejemplo 8.7

Escribir una función que calcule el máximo de los primeros  $n$  elementos de un arreglo 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;
}

```

## Precauciones

Cuando se utiliza una variable arreglo como argumento, la función receptora puede no conocer cuántos elementos existen en el arreglo. Sin su conocimiento una función no puede utilizar el arreglo. Aunque la variable arreglo puede apuntar a su comienzo, no proporciona ninguna indicación de dónde termina el arreglo.

La función `SumaDeEnteros()` suma los valores de todos los elementos de un arreglo y devuelve el total.

```
int SumaDeEnteros(int ArrayEnteros[])
{
 ...
}
int main()
{
 int lista[5] = {10, 11, 12, 13, 14};
 SumaDeEnteros (lista);
 ...
}
```

Aunque `SumaDeEnteros()` conoce dónde comienza el arreglo, no anota cuántos elementos hay en el arreglo; en consecuencia, no sabe cuántos elementos hay que sumar.

### Consejo de programación

Se pueden utilizar dos métodos alternativos para permitir que una función conozca el número de argumentos asociados con un arreglo que se pasa como argumento de una función.

- situar un valor de señal al final del arreglo, que indique a la función que se debe detener el proceso en ese momento;
- pasar un segundo argumento que indica el número de elementos del arreglo.

Todas las cadenas utilizan el primer método ya que terminan en nulo. Una segunda alternativa es pasar el número de elementos del arreglo siempre que se pasa este como un argumento. El arreglo 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()` cuántos elementos se procesarán en el arreglo `ArrayEnteros`. Este método suele ser el utilizado para arreglos de elementos que no son caracteres.

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++)
```

### Ejemplo 8.8



```

 scanf ("%d", &Items[i]);
 printf ("Total = %d \n", SumaDeEnteros(Items, 10));
 printf ("Valor máximo: %d \n", maximo(Items, 10));
 return 0;
 }
 int SumaDeEnteros (cons 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;
 }
}

```

### Paso de cadenas como parámetros

La técnica de pasar arreglos como parámetros se utiliza para pasar cadenas de caracteres (véase apartado 8.5) a funciones. Las cadenas terminadas en nulo utilizan el primer método dado anteriormente para controlar el tamaño de un arreglo. Las cadenas son arreglos de caracteres. Cuando una cadena se pasa a una función, como `strlen()`, la función conoce que se ha llegado al final del arreglo cuando lee un valor de 0 en un elemento del arreglo.

Las cadenas utilizan siempre un 0 para indicar que es el último elemento del arreglo de caracteres. Este 0 es el carácter nulo del código de caracteres ASCII.

El siguiente ejemplo de una función 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 `convierte_mayus()` recibe una cadena, un arreglo de caracteres cuyo último carácter es el nulo (0). El bucle termina cuando se alcance el fin de la 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 Concepto de cadena

Una *cadena* (también llamada *constante de cadena* o *literal de cadena*) es un tipo de dato compuesto, un arreglo de caracteres (`char`), terminado por un carácter *nulo* (`'\0'`), `NULL` (figura 8.10). Un ejemplo es:

`"ABC".`

Cuando la cadena aparece dentro de un programa se verá como si se almacenaran cuatro elementos: '`A'`', '`B'`', '`C'` y '`\0`'. En consecuencia, se considerará que la cadena "ABC" es un arreglo de cuatro elementos de tipo `char`. El valor real de esta cadena es la dirección de su primer carácter y su tipo es un apuntador a `char`. Aplicando el operador `*` a un apuntador a `char` se obtiene el carácter que forma su contenido; es posible también utilizar aritmética de apuntadores con cadenas:

```
*"ABC" es igual a 'A'
*("ABC" + 1) es igual a 'B'
*("ABC" + 2) es igual a 'C'
*("ABC" + 3) es igual a '\0'
```

De igual forma, utilizando el subíndice del arreglo se puede escribir:

```
"ABC" [0] es igual a 'A'
"ABC" [1] es igual a 'B'
"ABC" [2] es igual a 'C'
"ABC" [3] es igual a '\0'
```

### A recordar

El número total de caracteres de una cadena en C es siempre igual a la longitud de la cadena más 1.

### Ejemplos

1. `char cad[ ] = "Lupiana";`  
cad tiene ocho caracteres; '`L`', '`u`', '`p`', '`i`', '`a`', '`n`', '`a`' y '`\0`'
2. `printf("%s", cad);`  
el sistema copiará caracteres de `cad` a `stdout` (pantalla) hasta que el carácter `NULL`, '`\0`', se encuentre.
3. `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 suficientemente grande para contener la entrada.

### A recordar

Las funciones declaradas en el archivo de cabecera `<string.h>` se utilizan para manipular cadenas.

## Declaración de variables de cadena

Las cadenas se declaran como los restantes tipos de arreglos. 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 *bit de signo* y se puede propagar a la posición de mayor orden (peso) del nuevo tipo.

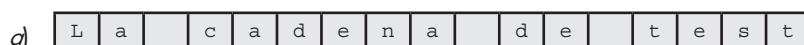


Figura 8.10 a) Arreglo de caracteres; b) Cadena de caracteres.

Observe que el tamaño de la cadena ha de incluir el carácter '\0'. En consecuencia, para definir un arreglo de caracteres que contenga la cadena "ABCDEF", escriba,

```
char UnaCadena[7];
```

### Nota de programación

A veces se puede encontrar una declaración como ésta:

```
char *s;
```

¿Es s realmente una cadena? No, no es. Es un apuntador a un carácter (el primer carácter de una cadena), que todavía no tiene memoria asignada.

## Inicialización de variables de cadena

Todos los tipos de arreglos requieren una *inicialización* (iniciación) que consiste en una lista de valores encerrados entre comillas. Por ejemplo:

```
char texto[81] = "Esto es una cadena";
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 solo 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 arreglo se trata como un valor de dirección, como un apuntador constante. ¿Cómo se puede inicializar una cadena fuera de la declaración? Normalmente se hace con funciones de copia, de la biblioteca *string.h*; la más utilizada es *strcpy()* que posteriormente se estudia.



### Ejemplo 8.9

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 manda imprimir, no escribe nada.

## 8.6 Lectura de cadenas

La entrada de datos se realiza, habitualmente, con la función `scanf( )`; para entrada de cadenas se utiliza el código `%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 arreglo de caracteres de 30 elementos. Suponga que introduce la entrada `Pepe Martolles`, cuando ejecuta el programa se visualizará en pantalla `Pepe`. Es decir, la palabra `Martolles` 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 y termina al leer el carácter de fin de línea.

### Función `gets( )`

Para entrada de cadenas desde el teclado se utiliza, normalmente, la función `gets( )`. Ésta lee caracteres, incluyendo blancos de separación entre palabras, hasta encontrar el carácter de *fin de línea*, o bien el carácter *fin de fichero* (archivo). Entonces, el carácter que utiliza `gets( )` para delimitar cadenas es el fin de línea (retorno de carro) en vez del espacio en blanco que es el utilizado por la función `scanf( )`.

El argumento de `gets( )` ha de ser una variable cadena de suficiente espacio para que puedan guardarse todos los caracteres tecleados (hay que recordar que en las cadenas se inserta automáticamente el carácter nulo, ‘\0’, en la última posición), el carácter de fin de línea no se incluye en la variable cadena. Por ejemplo, con las siguientes sentencias se puede leer una cadena de un máximo de 20 caracteres:

```
char bf[21];
gets(bf);
```

Si al ejecutar el programa que tuviera esas sentencias se teclea: `En Pradoluengo los perros ladran.` La cadena en la variable `bf` no va a terminar con el carácter nulo debido a que se han tecleado más de 20 caracteres, esto provocará resultados incorrectos en la ejecución del programa.

La función devuelve un apuntador (ver capítulo 11) a la cadena leída y almacenada en su argumento; si ha habido error, o bien la línea tecleada comienza con el carácter *fin de fichero* o *archivo*, devuelve `NULL`. El prototipo de la función que se encuentra en `stdio.h` es el siguiente:

```
char* gets(char *s);
```

Se desea leer líneas de texto, máximo de 80 caracteres, y contar el número de palabras que tiene cada línea.

Cada línea se lee llamando a la función `gets( )`, con un argumento que pueda almacenar el máximo de caracteres de una línea. Por consiguiente se declara la variable: `char cad[81]`, que será el argumento de `gets( )`. Con el fin de simplificar, se supone que las palabras se separan con un espacio; entonces para contar las palabras se recorre el arreglo `cad` contando el número de espacios, la longitud de la cadena se determina con una llamada a `strlen( )`. El número de palabras será el número de espacios (blancos) contados,

### Ejemplo 8.10



más uno ya que la última palabra no termina con un espacio sino con el retorno de carro. La ejecución termina tecleando al inicio de una línea ^Z (tecla Ctrl “control” y z); entonces la función `gets( )` devuelve NULL y termina el bucle.

```
#include <stdio.h>
#include <string.h>
void main()
{
 char cad[81], *a;
 int i, n;
 puts("Introduce líneas, separando las palabras con blancos.\n");
 a = gets(cad);
 while (a != NULL)
 {
 n = 0;
 for (i = 0; i < strlen(cad); i++)
 if (cad[i] == ' ') n++;
 /* también se accede a los char con *(cad+i) */
 if (i > 0) ++n;
 printf("Número de palabras: %d\n", n);
 a = gets(cad);
 }
}
```

### Ejecución

```
Introduce líneas, separando las palabras con blancos.
Puedes ver el futuro con pesimismo u optimismo
Número de palabras: 8
Siempre con la vista clara y adelante
Número de palabras: 7
^Z
```

### Precaución

Si el primer carácter de la línea de texto es el carácter fin de línea, la función `gets (cad)` asigna a `cad` una cadena nula.



### Ejemplo 8.11

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];
 char CodigoPostal[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);
/* Visualizar 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`'.

### 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`).

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;
}

```

### Ejemplo 8.12

#### Nota

La salida del bucle es con `Control-Z`, como único carácter de la última línea.

### 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 8.3

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>`.

La variable `pre` contiene el carácter previo, entonces si este es un blanco o fin de línea, el carácter siguiente es el de una nueva palabra y se convierte a mayúscula.

```
#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
```

## Función `puts( )`

La función `puts( )` visualiza una cadena de caracteres, incluyendo el carácter fin de línea. Es la función opuesta 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.h`:

```
int puts(const char *s);
```



### Ejercicio 8.4

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, lo hace sin la última palabra de la línea anterior.

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. La longitud de la cadena se obtiene con una llamada a `strlen( )` de la biblioteca `string.h`.

```
#include <stdio.h>
#include <string.h>
void sgtepal(char* p);
void main()
{
 char linea[81];
 printf("\n\tIntroduce una linea de caracteres.\n");
 gets(linea);
 while (*linea)
 {
 puts(linea);
 linea++;
 }
}
```

```

 sgtepalc(linea);
 }
}

void sgtepalc(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

```

## 8.7 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 proveedores 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 apuntador a `char` o se puede especificar el nombre de una variable arreglo `char`. Cuando se pasa un arreglo a una función, C pasa automáticamente la dirección del arreglo `char`. La tabla 8.2 resume algunas de las funciones de cadena más usuales.

### Utilización del modificador `const` con cadenas

Las funciones de cadena declaradas en `<string.h>`, recogidas en la tabla 8.2 y algunas otras, incluyen la palabra reservada `const`. Generalmente los argumentos apuntador a `char` precedidos de `const` son de entrada, en caso contrario de salida. Por ejemplo, el segundo parámetro fuente de `strcpy` representa el área fuente; se utiliza solo para copiar caracteres de ella, de modo que esta 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. Por contra, el primer parámetro *destino* de `strcpy` es el área de destino, la cual se sobreescribirá 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.

## 8.8 Arreglos y cadenas como parámetros de funciones

En la llamada a una función los arreglos y cadenas siempre se pasa la dirección del arreglo, un apuntador al primer elemento del arreglo. En la función, las referencias a los elementos individuales se hacen por indirección de la dirección del arreglo o cadena.

**Tabla 8.2** Funciones de cadena de <string.h>

| Función           | Cabecera de la función y prototipo                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strcat</b>     | char *strcat(char *destino, const char *fuente);<br>Añade la cadena <i>fuente</i> al final de <i>destino</i> , <i>concatena</i> . Devuelve la cadena <i>destino</i> .                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>strchr( )</b>  | char* strchr(const char* s1, int ch);<br>Devuelve un puntero a la primera ocurrencia de <i>ch</i> en <i>s1</i> . Devuelve <b>NULL</b> si <i>ch</i> no está en <i>s1</i> .                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>strcmp( )</b>  | int strcmp(const char *s1, const char *s2);<br>Compara alfabéticamente la cadena <i>s1</i> con <i>s2</i> y devuelve:<br>0 si <i>s1</i> = <i>s2</i><br>< 0 si <i>s1</i> < <i>s2</i><br>> 0 si <i>s1</i> > <i>s2</i>                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>strcpy( )</b>  | char *strcpy(char *destino, const char *fuente);<br>Copia la cadena <i>fuente</i> a la cadena <i>destino</i> . Devuelve la cadena <i>destino</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>strcspn( )</b> | size_t strcspn(const char* s1, const char* s2);<br>Devuelve la longitud de la subcadena más larga de <i>s1</i> que comienza con el carácter <i>s1[0]</i> y no contiene ninguno de los caracteres de la cadena <i>s2</i> .                                                                                                                                                                                                                                                                                                                                                                           |
| <b>strlen( )</b>  | size_t strlen (const char *s)<br>Devuelve la longitud de la cadena <i>s</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>strncpy( )</b> | char* strncpy (char* destino, const char* fuente, size_t n);<br>Copia <i>n</i> caracteres de la cadena <i>fuente</i> en <i>destino</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>strncat( )</b> | char* strncat(char* s1, const char*s2, size_t n);<br>Añade los primeros <i>n</i> caracteres de <i>s2</i> a <i>s1</i> . Devuelve <i>s1</i> . Si <i>n</i> >= strlen( <i>s2</i> ), entonces strncat( <i>s1</i> , <i>s2</i> , <i>n</i> ) tiene el mismo efecto que strcat( <i>s1</i> , <i>s2</i> ).                                                                                                                                                                                                                                                                                                     |
| <b>strncmp( )</b> | int strncmp(const char* s1, const char* s2, size_t n);<br>Compara alfabéticamente los <i>n</i> primeros caracteres de <i>s1</i> con los de <i>s2</i> . Si <i>n</i> es mayor que el número de caracteres de una cadena se comporta como strcmp( ). Si <i>n</i> ≥ strlen( <i>s2</i> ), entonces strncmp( <i>s1</i> , <i>s2</i> , <i>n</i> ) y strcmp( <i>s1</i> , <i>s2</i> ) tienen el mismo efecto.                                                                                                                                                                                                 |
| <b>strpbrk( )</b> | char* strpbrk(const char* s1, const char* s2);<br>Devuelve la dirección de la primera ocurrencia en <i>s1</i> de cualquiera de los caracteres de <i>s2</i> . Devuelve <b>NULL</b> si ninguno de los caracteres de <i>s2</i> aparece en <i>s1</i> .                                                                                                                                                                                                                                                                                                                                                  |
| <b>strrchr( )</b> | char* strrchr(const char* s, int c);<br>Devuelve un puntero a la última ocurrencia de <i>c</i> en <i>s</i> . Devuelve <b>NULL</b> si <i>c</i> no está en <i>s</i> . La búsqueda la hace en sentido inverso, desde el final de la cadena al primer carácter, hasta que encuentra el carácter <i>c</i> .                                                                                                                                                                                                                                                                                              |
| <b>strspn( )</b>  | size_t strspn(const char* s1, const char* s2);<br>Devuelve el número máximo de caracteres pertenecientes a la cadena <i>s2</i> que consecutivamente están en la cadena <i>s1</i> .                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>strstr( )</b>  | char* strstr(const char *s1, const char *s2);<br>Busca la cadena <i>s2</i> en <i>s1</i> y devuelve un puntero a los caracteres donde se encuentra <i>s2</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>strtok( )</b>  | char* strtok(char* s1, const char* s2);<br>Analiza la cadena <i>s1</i> en <i>tokens</i> (componentes léxicos), estos delimitados por caracteres de la cadena <i>s2</i> . La llamada inicial a strtok( <i>s1</i> , <i>s2</i> ) devuelve la dirección del primer <i>token</i> y sitúa <b>NULL</b> al final del <i>token</i> . Después de la llamada inicial, cada llamada sucesiva a strtok( <b>NULL</b> , <i>s2</i> ) devuelve un puntero al siguiente <i>token</i> encontrado en <i>s1</i> . Estas llamadas cambian la cadena <i>s1</i> , reemplazando cada separador con el carácter <b>NULL</b> . |

### Ejercicio 8.5

El programa siguiente extrae *n* caracteres de una cadena introducida por el usuario.

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 como tercero, el número de caracteres a extraer. Se utilizan los apuntadores para pasar arreglos a la función.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.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("Número 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 extraída \"%s\"", s2);
}

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 arreglo, sino como apuntadores de tipo *char*. En la línea,

```
*dest++ = *fuente++;
```

los apuntadores 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.

## 8.9 Asignación de cadenas

C soporta dos métodos para asignar cadenas. Uno de ellos, ya visto anteriormente, cuando se inicializan las variables de cadena. La sintaxis utilizada:

```
char VarCadena[LongCadena] = ConstanteCadena;
char *VarCadena = ConstanteCadena;
```

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 8.13

Una vez definido un arreglo 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`).

### 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 de caracteres de *fuente* si `num` es menor que su longitud; si `num` es mayor rellena con caracteres NULL.



### Ejemplo 8.14

Estas sentencias copian 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 de programación

Los apuntadores 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".

## 8.10 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.

### 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 definido en `string.h`.

```
char cad[] = "1234567890";
unsigned i;
i = strlen(cad);
```

Estas sentencias asignan 10 a la variable `i`.

Este programa muestra por pantalla la longitud de varias cadenas.

### Ejemplo 8.15

```
#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 cadena: ");
 gets(bufer);
 printf("strlen(%s) = %d", bufer, strlen(bufer));
}
```

#### Ejecución

```
strlen (IJKLMN) = 6
strlen ("") = 0
Introduzca una cadena: Sierra de Horche
strlen (Sierra de Horche) = 16
```

### 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 apuntador a la cadena destino. Su prototipo es:

```
char* strcat(char* destino, const char* fuente);
```

Copia una constante cadena y a continuación concatena con otra cadena.

### Ejemplo 8.16

```
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 apuntador a la cadena destino. Su prototipo es:

```
char* strncat(char* destino, const char* fuente, size_t num)
```

y cuando se invoca con una llamada 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, que termina el proceso.



### Ejemplo 8.17

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( )` comprueban 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 resultados extraños por no haber espacio
para la concatenación s1 con s2 */
```

## 8.11 Comparación de cadenas

Dado que las cadenas son arreglos 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( )`, y `strncmp( )`.

### 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 primera cadena es menor que la segunda cadena; o un valor mayor que cero si la primera cadena es mayor que la segunda (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 | <code>cad1</code> | es menor que | <code>cad2</code> |
| = 0 | si | <code>cad1</code> | es igual a   | <code>cad2</code> |
| > 0 | si | <code>cad1</code> | es mayor que | <code>cad2</code> |



### Ejemplo 8.18

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 `cad1[i]` o `cad2[i]`

|          |              |         |
|----------|--------------|---------|
| Waterloo | es menor que | Windows |
| Mortimer | es mayor que | Mortim  |
| Jertru   | es igual que | Jertru  |

### Función `strncmp( )`

La función `strncmp( )` compara los `num` caracteres más 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 |

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". Si se comparan 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".

### Ejemplo 8.19

## 8.12 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, y también las funciones `strtod( )`, `strtol( )` y `strtoul( )` para conversiones más complejas. Estas funciones se incluyen en la biblioteca `stdlib.h`, por lo que ha de incluir en su programa la directiva:

```
#include <stdlib.h>.
```

### 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( )**.

### A recordar

Si la cadena no se puede convertir, **atoi( )** devuelve cero.



#### Ejemplo 8.20

Convierte los dígitos de una cadena en un valor entero.

```
char *cadena = "453423";
int valor;
valor = atoi(cadena);
```

### 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 los 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 8.21

Convertir los dígitos de una cadena a un número de tipo `double`.

```
char *cadena = "545.7345";
double valor;
valor = atof(cadena);
```

### Función **atol( )**

La función **atol( )** convierte una cadena a un valor entero 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]

Una cadena que tiene dígitos consecutivos se convierte en entero largo.

### Ejemplo 8.22

```
char *cadena = "45743212";
long valor;
valor = atol(cadena);
```

## Función `strtol( )` y `strtoul( )`

La utilidad de estas dos funciones radica en que convierten los dígitos de una cadena, en cualquier sistema de numeración (base), a entero (long) o a entero sin signo (unsigned long). Por ejemplo:

```
char *c = "432342";
long n;
n = strtol(c, (char**)NULL, 8);
```

En la llamada anterior, el valor devuelto por la función será 144610. La función ha leído los dígitos 432342 como número en base 8 y lo ha transformado a su valor decimal (144610).

El primer argumento, en la llamada a ambas funciones, es la cadena a analizar. El segundo argumento, de tipo (char\*\*) , es de salida; si este es distinto de NULL contiene el apuntador al carácter de la cadena en el que ha finalizado el número. El tercer argumento es la base de numeración de los dígitos leídos, la transformación a número entero (long, o bien unsigned long) se realiza según esa base de numeración.

En este otro ejemplo se convierten los dos números de la cadena, en base 10, a su equivalente valor decimal. La llamada a la función se hace con el tercer argumento 0 (sería equivalente poner 10) para que interprete los dígitos de la cadena como decimales.

```
char *c = " -49 2332";
char **pc = (char**)malloc(1);
long n1;
unsigned long n2;
n1 = strtol(c,pc,0);
printf(" n1 = %ld\n",n1);
printf(" cadena actual %s\n",*pc);
c = *pc;
n2 = strtoul(c,pc,10);
printf("n2 = %lu",n2);
```

Al ejecutar las sentencias anteriores se obtienen estos resultados:

```
n1 = -49
cadena actual 2332
n2 = 2332
```

También se puede observar que ambas funciones ignoran los blancos por la izquierda. Además, si el tercer argumento es 0 y los dos primeros caracteres del número son 0x (o bien, 0X) la función interpreta que son dígitos hexadecimales; y si el primer carácter es 0 considera que son dígitos en octal. Por ejemplo:

```
strtol(" -0x11", (char**)NULL, 0);
strtol(" 012", (char**)NULL, 0);
```

devuelven -17 y 10, respectivamente.

Por último, si se produce un error de *overflow* o *underflow* ambas funciones asignan a la variable `errno` al valor `ERANGE` (se encuentran en `errno.h`) y devuelven la constante máximo entero o mínimo entero, respectivamente (`LONG_MAX`, `LONG_MIN`).

El prototipo de las funciones se encuentra en `stdio.h`; es el siguiente:

```
long strtol(const char* c, char** pc, int base);
unsigned long strtoul(const char* c, char** pc, int base);
```

### A recordar

La llamada a la función `atoi` (cadena) es equivalente a:

```
(int)strtol(cadena, (char**) NULL, 10)
```

La llamada a la función `atol` (cadena) es equivalente a:

```
strtol(cadena, (char**) NULL, 10)
```

### Función `strtod( )`

La función `strtod( )` convierte los dígitos de una cadena en un número real de tipo `double`. El primer argumento, en la llamada, es la cadena; el segundo argumento es de salida, al cual la función asigna un apuntador al carácter de la cadena con el que terminó de formarse el número real. Por ejemplo, `strtod (" -0.5347e+3 88", pc)` devuelve como número real  $-0.5347 \times 10^3$ ; la función salta los blancos por la izquierda y lee los caracteres  $-0.5347e+3$ , termina en el carácter blanco (espacio); el apuntador a ese carácter delimitador es asignado al argumento `pc`. Este segundo argumento ha de ser de tipo `char**`; para obtener el apuntador se accede a `*pc`. La utilidad de este segundo argumento está en que a partir de él se puede leer otro número de la cadena. Si no resulta de interés obtener el apuntador al carácter delimitador, se llama a la función con el segundo argumento `NULL`, por ejemplo `strtod(" -343.553 04322", (char**)NULL)`.

El siguiente programa muestra cómo obtener todos los números reales (tipo `double`) de una cadena. Se puede observar el uso de la variable `errno` (archivo `errno.h`); la función asigna a `errno` la constante `ERANGE` si se produce un error por *overflow* al convertir la cadena, y entonces devuelve la constante `HUGE_VAL`.

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

void main(void)
{
 char*c = "333.55553 444444.2 3e+1221";
 char **a;
 double v=0 ;
 a = (char**)malloc(1);

 v = strtod(c, a);
 if (errno != 0)
 {
 printf("Error \"%d\" al convertir la cadena.",errno);
 exit(-1);
 }
 printf("c = [%s], v = %lf\n",c,v);
 while ((*a) != '\0')
 {
 c = *a;
 v = strtod(c,a);
 if (errno != 0)
 {
 printf("Error \"%d\" al convertir la cadena.",errno);
 exit(-1);
 }
 printf("c = [%s], v = %lf\n",c,v);
 }
}
```

## Ejecución

```
c = [333.55553 444444.2 3e+1221] , v = 333.555530
c = [444444.2 3e+1221] , v = 444444.200000
Error "34" al convertir la cadena.
```

El prototipo de la función que se encuentra en `stdio.h`, es el siguiente:

```
double strtod(const char* c, char** pc);
```

## A recordar

La llamada a la función `atof(cadena)` es equivalente a:

```
strtod(cadena, char**) NULL)
```

## 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()`.

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 se visualiza,

```
Metros cuadrados: 1980756
Nombre de la finca:
```

## Ejemplo 8.23



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 opciones, la primera:

```
printf("Metros cuadrados: "); scanf("%ld%*c", &k);
printf("Nombre de la finca: "); gets(cad);
```

La segunda opción 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);
```

## Precaución

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.



## Resumen

En este capítulo se analizan los arreglos (*arrays*), tipos agregados de C. 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 arreglo (*array*), es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.
- Existen arreglos de una dimensión, de dos dimensiones... y multidimensionales.
- En C los arreglos se definen especificando el tipo de dato del elemento, el nombre del arreglo y el tamaño de cada dimensión del arreglo. Para acceder a los elementos del arreglo 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 arreglos 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 arreglos multidimensionales.

```
int ventas_totales[12][50];
```

En este capítulo también 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 arreglos 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 destaca

## Ejercicios

Para los ejercicios 8.1. a 8.6, 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]);
```

can las funciones que soportan asignación, concatenación, conversión 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* ( ) y *strncmp* ( ) permiten realizar diversos tipos de comparaciones. La función *strcmp* ( ) realiza una comparación de dos cadenas. La función *strncmp* ( ) es una variante de la función *strcmp* ( ), que utiliza un número especificado de caracteres al comparar cadenas.
- Las funciones *strchr* ( ), *strspn* ( ), *strcspn* ( ) y *strpbrk* ( ) permiten buscar 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.

.....

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 arreglo

```

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 arreglo

```

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 arreglo de dos dimensiones,  $m \times n$ .

8.10 Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.

8.11 Escribir una función que obtenga una cadena del dispositivo de entrada, de igual forma que `char* gets(char*)`. Utilizar para ello `getchar()`.

8.12 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 apuntador a la cadena leída, o `EOF` si se alcanzó el fin de fichero (archivo). El prototipo de la función debe ser:

```
char[] lee_linea(char[] c, int n);
```

8.13 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.

8.14 Escribir una función para transformar un número entero en una cadena de caracteres formada por los dígitos del número entero.

8.15 Escribir una función para transformar un número real en una cadena de caracteres que sea la representación decimal del número real.



## 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  |    |    |
|   |   | 1  | 1  |    |
|   | 1 | 2  | 1  |    |
|   | 1 | 3  | 3  | 1  |
| 1 | 4 | 6  | 4  | 1  |
| 1 | 5 | 10 | 10 | 5  |
| 1 | 6 | 15 | 20 | 15 |
|   |   |    |    | 6  |
|   |   |    |    | 1  |

En el triángulo de Pascal cada número es suma de los dos números situados encima de él. Este problema se debe resolver utilizando un arreglo 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, etcétera.
- 8.4 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.

- 8.5 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 la 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.

- 8.6 El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona la palabra y el otro jugador trata de averiguar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia:* almacenar una lista de palabras en un arreglo y seleccionar palabras aleatoriamente.
- 8.7 Escribir un programa que lea las dimensiones de una matriz, la visualice y a continuación encuentre el mayor y menor elemento de la matriz y sus posiciones.
- 8.8 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.
- $$\text{Varianza} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$
- 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.
- 8.9 Escribir un programa para leer una matriz A y formar la matriz transpuesta de A. El programa debe escribir ambas matrices.
- 8.10 Escribir una función que acepte como parámetro un arreglo que puede contener elementos duplicados. La función debe sustituir cada valor repetido por -5 y devolver al punto donde fue llamado el arreglo modificado y el número de entradas modificadas.
- 8.11 Los resultados de las últimas elecciones a alcalde en el pueblo x han sido los siguientes:

| Distrito | Candidato | Candidato | Candidato | Candidato |
|----------|-----------|-----------|-----------|-----------|
|          | A         | B         | C         | 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 de 50% de los datos, el programa imprimirá un mensaje declarándole ganador.
  - d) Si ningún candidato recibe más de 50% 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.
- 8.12 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).
- 8.13 Escribir un programa en el que se genere aleatoriamente un arreglo 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 arreglo original y el arreglo con la distribución indicada.
- 8.14 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.
- 8.15 Se quiere leer un texto con un máximo de 30 líneas. Se quiere que el texto se muestre de tal forma que aparezcan las líneas en orden alfabético.

8.16 Se sabe que en las líneas que forman un texto hay valores numéricos enteros, representan los kg de papas 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.

8.17 Escribir un programa que lea una cadena clave y un texto con un máximo de 50 líneas. El programa debe eliminar las líneas que contengan la clave.

8.18 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.

8.19 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 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 formatear el texto para que todas las líneas tengan la longitud de la máxima línea.

8.20 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.



# Algoritmos de ordenación y búsqueda

## Contenido

- 9.1 Ordenación
- 9.2 Ordenación por burbuja
- 9.3 Ordenación por selección
- 9.4 Ordenación por inserción
- 9.5 Ordenación rápida (quicksort)

## 9.6 Búsqueda en listas: búsqueda secuencial y binaria

- › Resumen
- › Ejercicios
- › Problemas

## Introducción

Muchas actividades humanas requieren que diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; los anuarios o directorios telefónicos se ordenan por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado. Los estudiantes de una clase en la universidad se ordenan por sus apellidos o por los números de expediente. Por esta circunstancia una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la ordenación.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. El capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C. De igual modo se estudiará el análisis de los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

En el capítulo se analizarán métodos básicos y avanzados empleados en programas profesionales.

## Conceptos clave

- › Búsqueda
- › Complejidad del algoritmo
- › Intercambio
- › Ordenación por inserción
- › Ordenación por selección
- › Ordenación por burbuja
- › Ordenación rápida

## 9.1 Ordenación

La **ordenación** o **clasificación** de datos (*sort* en inglés) es una operación consistente en disponer un conjunto (estructura) de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un cam-

po número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres. Los elementos numéricos se pueden organizar en orden creciente o decreciente de acuerdo con el valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*.

Una colección de datos (*estructura*) puede ser almacenada en un *archivo*, un arreglo (*vector* o *tabla*), un arreglo de *registros*, una lista enlazada o un *árbol*. Cuando los datos están almacenados en un arreglo, una lista enlazada o un árbol, se denomina *ordenación interna*. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*.

Se dice que una *lista* está *ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esa clave. Se dice que la lista está en *orden ascendente* si:

$i < j$  implica que  $k[i] \leq k[j]$

y se dice que está en *orden descendente* si:

$i > j$  implica que  $k[i] \leq k[j]$

para todos los elementos de la lista. Por ejemplo, para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave *k*, donde  $k[i]$  es el nombre del abonado (apellidos, nombre).

4 5 14 21 32 45 orden ascendente  
75 70 35 16 14 12 orden descendente

Zacarias Rodriguez Martinez Lopez Garcia orden descendente

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo se sabe cuál es el mejor algoritmo? La *eficiencia* es el factor que mide la calidad y rendimiento de un algoritmo. En el caso de la operación de ordenación, se suelen seguir dos criterios a la hora de decidir qué algoritmo, de entre los que resuelven la ordenación, es el más eficiente: 1) *tiempo menor de ejecución en computadora*; 2) *menor número de instrucciones*. Sin embargo, no siempre es fácil efectuar estas medidas: puede no disponerse de instrucciones para medida de tiempo, aunque no sea este el caso del lenguaje C, y las instrucciones pueden variar dependiendo del lenguaje y del propio estilo del programador. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza. Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el número de comparaciones entre elementos efectuados. El algoritmo de ordenación *A* será más eficiente que el *B*, si requiere menor número de comparaciones. Así, en el caso de ordenar los elementos de un vector, el número de comparaciones será *función* del número de elementos (*n*) del vector (*arreglo*). Por consiguiente, se puede expresar el número de comparaciones en términos de *n* (por ejemplo,  $n + 4$ ), o bien,  $n^2$  en lugar de números enteros (por ejemplo, 325).

En todos los métodos de este capítulo, normalmente, para comodidad del lector, se utiliza el orden ascendente sobre vectores o listas (arreglos unidimensionales).

Los métodos de ordenación se suelen dividir en dos grandes grupos:

- *directos*: burbuja, selección, inserción;
- *indirectos* (avanzados) *shell*, *ordenación rápida*, ordenación por mezcla, *radixsort*.

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

### A recordar

Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según que los elementos a ordenar estén en la memoria principal o en la memoria externa.

## 9.2 Ordenación por burbuja

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprender y programar; por el contrario, es el menos eficiente y por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina *ordenación por burbuja* u *ordenación por hundimiento* debido a que los valores más pequeños “*burbujean*” gradualmente (*suben*) hacia la cima o parte superior del arreglo (*array*) de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del arreglo. La técnica consiste en hacer varias pasadas a través del arreglo. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el arreglo.

### Algoritmo de la burbuja

En el caso de un arreglo (lista) con  $n$  elementos, la ordenación por burbuja requiere hasta  $n-1$  pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha “*burbujeado*” hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista  $A[n-1]$  está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

- En la pasada 0 se comparan elementos adyacentes

$(A[0], A[1]), (A[1], A[2]), (A[2], A[3]), \dots (A[n-2], A[n-1])$

Se realizan  $n-1$  comparaciones, por cada pareja  $(A[i], A[i+1])$  se intercambian los valores si  $A[i+1] < A[i]$ .

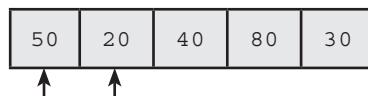
Al final de la pasada, el elemento mayor de la lista está situado en  $A[n-1]$ .

- En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en  $A[n-2]$ .
- El proceso termina con la pasada  $n-1$ , en la que el elemento más pequeño se almacena en  $A[0]$ .

El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada  $n-1$ , o bien antes. Si en una pasada no se produce intercambio alguno entre elementos del arreglo es porque ya está ordenado, entonces no son necesarias más pasadas.

El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un arreglo de 5 elementos ( $A = 50, 20, 40, 80, 30$ ) donde se introduce una variable `interruptor` para detectar si se ha producido intercambio en la pasada.

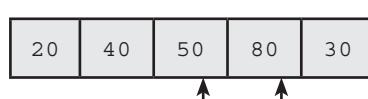
Pasada 0



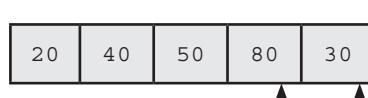
Intercambio 50 y 20



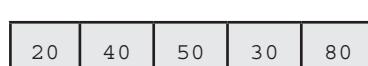
Intercambio 50 y 40



50 y 80 ordenados

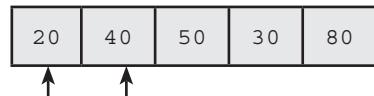


Intercambio 80 y 30

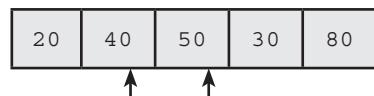


Elemento mayor es 80  
interruptor = TRUE

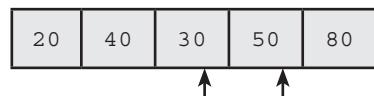
Pasada 1



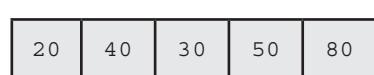
20 y 40 ordenados



40 y 50 ordenados



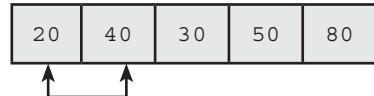
Se intercambian 50 y 30



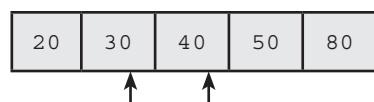
- 50 y 80 elementos mayores ordenados
- interruptor = TRUE

En la pasada 2, solo se hacen dos comparaciones y se produce un intercambio.

Pasada 2



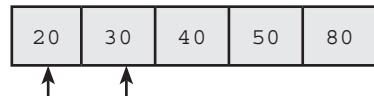
20 y 40 ordenados



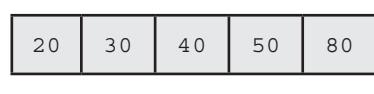
- Se intercambian 40 y 30
- interruptor = TRUE

En la pasada 3, se hace una única comparación de 20 y 30 y no se produce intercambio.

Pasada 3



20 y 30 ordenados



- Lista ordenada
- interruptor = FALSE

En consecuencia, el algoritmo de ordenación por burbuja mejorado contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas (al principio de la primera pasada todavía no se ha producido ningún intercambio, por lo tanto la variable interruptor se pone a valor falso (0); el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de interruptor a verdadero (1)).

El algoritmo terminará, bien cuando se finalice la última pasada ( $n-1$ ) o bien cuando el valor del interruptor sea falso (0), es decir no se haya hecho ningún intercambio. La condición para realizar una nueva pasada se define en la expresión lógica:

(pasada < n-1) && interruptor

### Codificación del algoritmo de la burbuja

La función `ordBurbuja()` implementa el algoritmo de ordenación por burbuja. Tiene dos argumentos, el arreglo que se va a ordenar crecientemente, y el número de elementos  $n$ . En la codificación se supone que los elementos son de tipo entero largo.

```
void ordBurbuja(long a[], int n)
{
```

```

int interruptor = 1;
int pasada, j;
for (pasada = 0; pasada < n-1 && interruptor; pasada++)
{
 /* bucle externo controla la cantidad de pasadas */
 interruptor = 0;
 for (j = 0; j < n-pasada-1; j++)
 if (a[j] > a[j+1])
 {
 /* elementos desordenados, es necesario intercambio */
 long aux;
 a[j] = a[j+1];
 a[j+1] = aux;
 }
 }
}

```

Una modificación del algoritmo anterior puede ser utilizar, en lugar de una variable bandera `interruptor`, una variable `indiceIntercambio` que se inicie a 0 (cero) al principio de cada pasada y se establezca al índice del último intercambio, de modo que cuando al terminar la pasada el valor de `indiceIntercambio` siga siendo 0 implicará que no se ha producido ningún intercambio (o bien que el intercambio ha sido con el primer elemento) y, por consiguiente, la lista estará ordenada. En caso de no ser 0, el valor de `indiceIntercambio` representa el índice del arreglo a partir del cual los elementos están ordenados. La codificación en C de esta alternativa sería:

```

/*
 Ordenación por burbuja: array (arreglo) de n elementos
 Se realizan una serie de pasadas mientras indiceIntercambio > 0
*/
void ordBurbuja2 (long a[], int n)
{
 int i, j;
 int indiceIntercambio;
 /* i es el índice del último elemento de la sublista */
 i = n-1;
 /* el proceso continúa hasta que no haya intercambios */
 while (i > 0)
 {
 indiceIntercambio = 0;
 /* explorar la sublista a[0] a a[i] */
 for (j = 0; j < i; j++)
 /* intercambiar pareja y actualizar IndiceIntercambio */
 if (a[j+1] < a[j])
 {
 long aux = a[j];
 a[j] = a[j+1];
 a[j+1] = aux;
 indiceIntercambio = j;
 }
 /* i se pone al valor del índice del último intercambio */
 i = indiceIntercambio;
 }
}

```

### Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación por burbuja? Dependerá de la versión utilizada. En la versión más simple se hacen  $n-1$  pasadas y  $n-1$  comparaciones en cada pasada. Por consiguiente, el número de comparaciones es  $(n-1) * (n-1) = n^2 - 2n + 1$ , es decir la complejidad es  $O(n^2)$ .

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables `interruptor` o `indiceIntercambio`, se tendrá entonces una eficiencia diferente para cada algoritmo. En el mejor de los casos, la ordenación por burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y por lo tanto su complejidad es  $0(n)$ . En el peor caso se requieren  $(n-i-1)$  comparaciones y  $(n-i-1)$  intercambios. La ordenación completa requiere  $\frac{n(n-1)}{2}$  comparaciones y un número similar de intercambios. La complejidad para el peor caso es  $0(n^2)$  comparaciones y  $0(n^2)$  intercambios.

De cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar, entonces, que el número medio de pasadas  $k$  sea  $0(n)$  y el número total de comparaciones es  $0(n^2)$ .

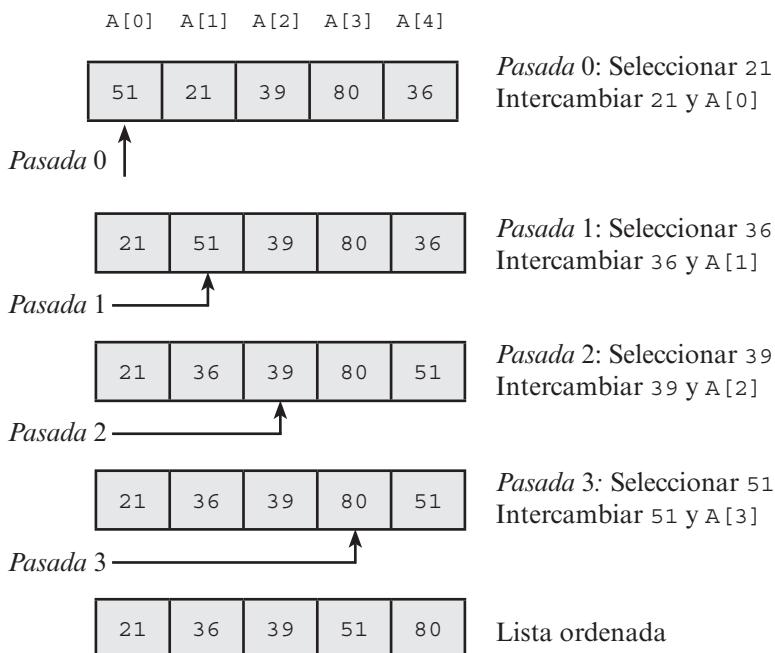
### 9.3 Ordenación por selección

Considérese el algoritmo para ordenar un arreglo  $A$  de enteros en orden ascendente, esto es del número más pequeño al mayor. Si el arreglo  $A$  tiene  $n$  elementos, se trata de ordenar los valores del arreglo de modo que el dato contenido en  $A[0]$  sea el valor más pequeño, el valor almacenado en  $A[1]$  el siguiente más pequeño, y así hasta  $A[n-1]$  que ha de contener el elemento de mayor valor. El algoritmo de selección se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista,  $A[0]$  en la primera pasada. En síntesis, se busca el elemento más pequeño de la lista y se intercambia con  $A[0]$ , primer elemento de la lista.

$A[0] \ A[1] \ A[2] \dots A[n-1]$

Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista  $A[1], A[2] \dots A[n-1]$  permanece desordenado. La siguiente pasada busca en esta lista desordenada y selecciona el elemento más pequeño, que se almacena entonces en la posición  $A[1]$ . De este modo los elementos  $A[0]$  y  $A[1]$  están ordenados y la sublista  $A[2], A[3] \dots A[n-1]$  desordenada; entonces, se selecciona el elemento más pequeño y se intercambia con  $A[2]$ . El proceso continúa  $n-1$  pasadas; en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el arreglo completo ha quedado ordenado.

Un ejemplo práctico ayudará a la comprensión del algoritmo. Consideremos un arreglo  $A$  con 5 valores enteros 51, 21, 39, 80, 36:



## Algoritmo de selección

Los pasos del algoritmo son:

1. Seleccionar el elemento más pequeño de la lista A. Intercambiárselo con el primer elemento A[0]. Ahora la entrada más pequeña está en la primera posición del vector.
2. Considerar las posiciones de la lista A[1], A[2], A[3]... seleccionar el elemento más pequeño e intercambiárselo con A[1]. Ahora las dos primeras entradas de A están en orden.
3. Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista, intercambiándolos adecuadamente.

## Codificación del algoritmo de selección

La función `ordSeleccion()` ordena una lista o vector de números reales de  $n$  elementos. En la pasada  $i$ , el proceso de selección explora la sublista  $A[i]$  a  $A[n-1]$  y fija el índice del elemento más pequeño. Después de terminar la exploración, los elementos  $A[i]$  y  $A[indiceMenor]$  intercambian las posiciones.

```
/*
ordenar un array (arreglo) de n elementos de tipo double
utilizando el algoritmo de ordenación por selección
*/
void ordSeleccion (double a[], int n)
{
 int indiceMenor, i, j;
 /* ordenar a[0]..a[n-2] y a[n-1] en cada pasada */
 for (i = 0; i < n-1; i++)
 {
 /* comienzo de la exploración en índice i */
 indiceMenor = i;
 /* j explora la sublista a[i+1]..a[n-1] */
 for (j = i+1; j < n; j++)
 if (a[j] < a[indiceMenor])
 indiceMenor = j;
 /* sitúa el elemento más pequeño en a[i] */
 if (i != indiceMenor)
 {
 double aux = a[i];
 a[i] = a[indiceMenor];
 a[indiceMenor] = aux;
 }
 }
}
```

## 9.4 Ordenación por inserción

El método de *ordenación por inserción* es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado. Así el proceso en el caso de la lista de enteros  $A = 50, 20, 40, 80, 30$  se muestra en la figura 9.1.

## Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento  $A[0]$  se considera ordenado es decir, la lista inicial consta de un elemento.
2. Se inserta  $A[1]$  en la posición correcta delante o detrás de  $A[0]$ , dependiendo de que sea menor o mayor.

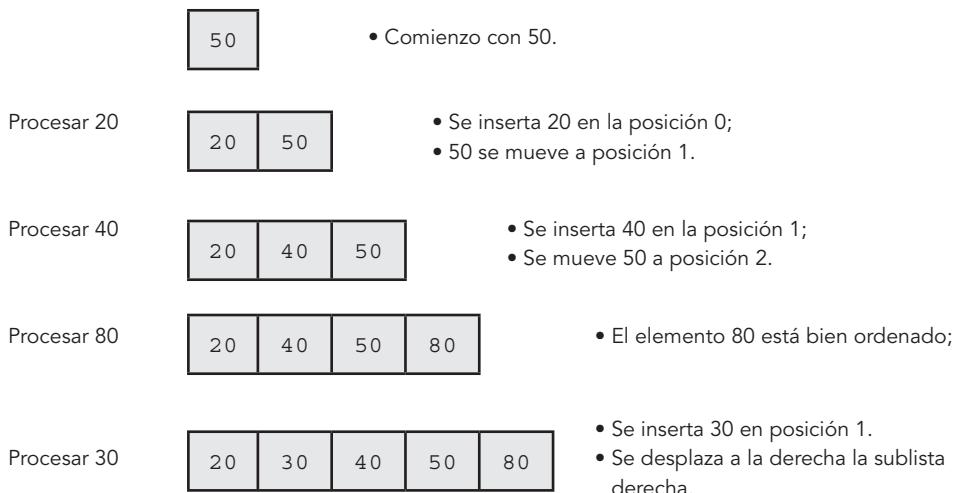


Figura 9.1 Método de ordenación por inserción.

3. Por cada bucle o iteración  $i$  (desde  $i = 1$  hasta  $n-1$ ) se explora la sublista  $A[i-1] \dots A[0]$  buscando la posición correcta de inserción; a la vez se mueve hacia la derecha en la sublista una posición todos los elementos mayores que el elemento a insertar  $A[i]$ , para dejar vacía esa posición.
4. Insertar el elemento en la posición correcta.

### Codificación del algoritmo de inserción

El método `ordInsercion()` tiene dos argumentos, el arreglo `a[]` que se va a ordenar crecientemente y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero.

```
void ordInsercion (int a[], int n)
{
 int i, j;
 int aux;
 for (i = 1; i < n; i++)
 /* indice j explora la sublista a[i-1]...a[0] buscando la
 posición correcta del elemento destino, lo asigna a a[j] */
 {
 j = i;
 aux = a[i];
 /* se localiza el punto de inserción explorando hacia abajo */
 while (j > 0 && aux < a[j-1])
 {
 /* desplazar elementos hacia arriba para hacer espacio */
 a[j] = a[j-1];
 j--;
 }
 a[j] = aux;
 }
}
```

La complejidad del algoritmo es cuadrática,  $O(n^2)$ , debido a que todo el proceso se controla con dos bucles anidados que en el peor de los casos realizan  $n-1$  iteraciones.

## 9.5 Ordenación rápida (quicksort)

El algoritmo conocido como *quicksort* (ordenación rápida) fue creado por Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar

que aplica la técnica “divide y vence”. El método es, posiblemente, el más pequeño en código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

El método se basa en dividir los  $n$  elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición *izquierda*, un elemento *central* denominado *pivote* o elemento de partición, y una partición *derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublistas (partición izquierda) son menores que todos los elementos de la segunda sublistas (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede escoger cualquier elemento de la lista como pivote, por ejemplo el primer elemento de la lista. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que divida la lista exactamente por la mitad, de acuerdo con el tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones “pobres” de pivotes.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra todos los elementos (claves) mayores o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan en forma recursiva utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublistas, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o “particionado” recursivo de la lista hasta que todas las sublistas constan de solo un elemento.

### A recordar

El algoritmo de ordenación *quicksort* sigue la estrategia típica de los algoritmos recursivos “divide y vence”. El problema de tamaño  $n$  (ordenar una lista de  $n$  elementos) se divide en dos subproblemas o tareas, cada una de las cuales se vuelve a dividir en dos tareas, cada vez de menor tamaño. Como cada tarea realiza las mismas acciones, el algoritmo se expresa recursivamente. El *caso base* (condición de parada) es conseguir una tarea (sublista) de tamaño 1.



### Ejemplo 9.1

Se ordena una lista de números enteros aplicando el algoritmo *quicksort*, como pivote se elige el primer elemento de la lista.

#### 1. Lista original

5 2 1 8 3 9 7

#### pivote elegido

5

↑  
sublista izquierda 1 (elementos menores que 5)

2 1 3

sublista derecha 1 (elementos mayores o iguales a 5)

8 9 7

#### 2. El algoritmo se aplica a la sublista izquierda:

Sublista izquierda 1

2 1 3

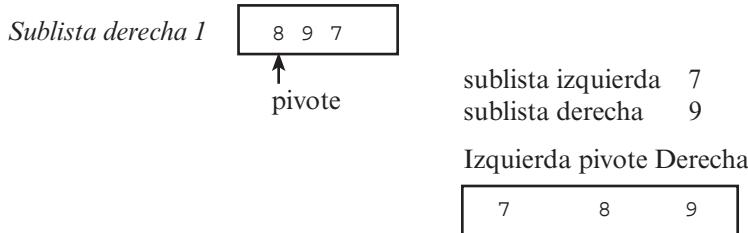
↑  
pivot

sublista izquierda 1  
sublista derecha 3

Izquierda pivot Derecha

1 2 3

3. El algoritmo se aplica a la sublista derecha:



4. *Lista ordenada final:*

Sublista izquierda      pivot      Sublista derecha

|       |   |       |
|-------|---|-------|
| 1 2 3 | 5 | 7 8 9 |
|-------|---|-------|

### A recordar

El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido aunque la estrategia de partición suele ser similar.

## Algoritmo quicksort

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar la forma de situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. En el ejemplo 9.2 se elige como pivote el elemento central de la lista actual.

Se ordena una lista de números enteros aplicando el algoritmo *quicksort*. Como pivote se elige el elemento central de la lista.

### Ejemplo 9.2

Lista original: 8 1 4 9 6 3 5 2 7 0

pivote (elemento central) 6

La etapa 2 requiere mover todos los elementos menores que el pivote a la parte izquierda del arreglo y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice  $i$ , que se inicializa a la posición más baja (inferior), buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice  $j$  inicializado a la posición más alta (superior).

El índice  $i$  se detiene en el elemento 8 (mayor que el pivote) y el índice  $j$  se detiene en el elemento 0 (menor que el pivote)

|         |   |           |
|---------|---|-----------|
| 8 1 4 9 | 6 | 3 5 2 7 0 |
| ↑<br>→i |   | ↑<br>j←   |

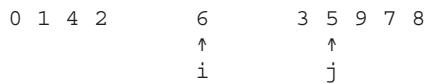
Ahora se intercambian 8 y 0 para que estos dos elementos se sitúen correctamente en cada sublista; se incrementa el índice  $i$  y se decrementa  $j$  para seguir los intercambios.

|         |   |           |
|---------|---|-----------|
| 0 1 4 9 | 6 | 3 5 2 7 8 |
| ↑<br>i  |   | ↑<br>j    |

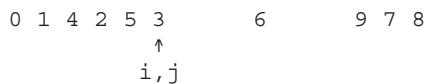
A medida que el algoritmo continúa,  $i$  se detiene en el elemento mayor, 9, y  $j$  se detiene en el elemento menor, 2.



Se intercambian los elementos mientras que  $i$  y  $j$  no se *crucen*. En el momento en que se cruzan los índices se detiene el recorrido. En el caso anterior se intercambian 9 y 2.



Continúa la exploración y ahora el índice  $i$  se detiene en el elemento 6 (que es el pivote) y el índice  $j$  se detiene en el elemento menor 5. Se intercambian 6 y 5, se incrementa  $i$  y se decrementa  $j$ .



Los índices tienen actualmente los valores  $i = 5$ ,  $j = 5$ . Continúa la exploración hasta que  $i > j$ , acaba con  $i = 6$ ,  $j = 5$ .



En esta posición los índices  $i$  y  $j$  han cruzado posiciones en el arreglo. En este caso se detiene la búsqueda y no se realiza ningún *intercambio* ya que el elemento al que accede  $j$  está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:

*sublista\_izquierda    pivot    sublista\_derecha*



El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote, en principio puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el pivote como el elemento central o próximo al central de la lista. La figura 9.2. muestra las operaciones del algoritmo para ordenar la lista  $a[]$  de  $n$  elementos enteros.

Pasos que sigue el algoritmo *quicksort*:

- Seleccionar el elemento central de  $a[0:n-1]$  como pivote.
- Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.
- *Ordenar* la partición *izquierda* utilizando *quicksort* recursivamente.
- *Ordenar* la partición *derecha* utilizando *quicksort* recursivamente.
- La solución es partición *izquierda* seguida por el pivote y a continuación partición *derecha*.

### Codificación del algoritmo *quicksort*

La función *quicksort()* refleja el algoritmo citado anteriormente; a esta función se la llama pasando como argumento el arreglo  $a[]$  y los índices que la delimitan 0 y  $n-1$  (índice inferior y superior). La llamada a la función:

```
quicksort(a, 0, n-1);
```

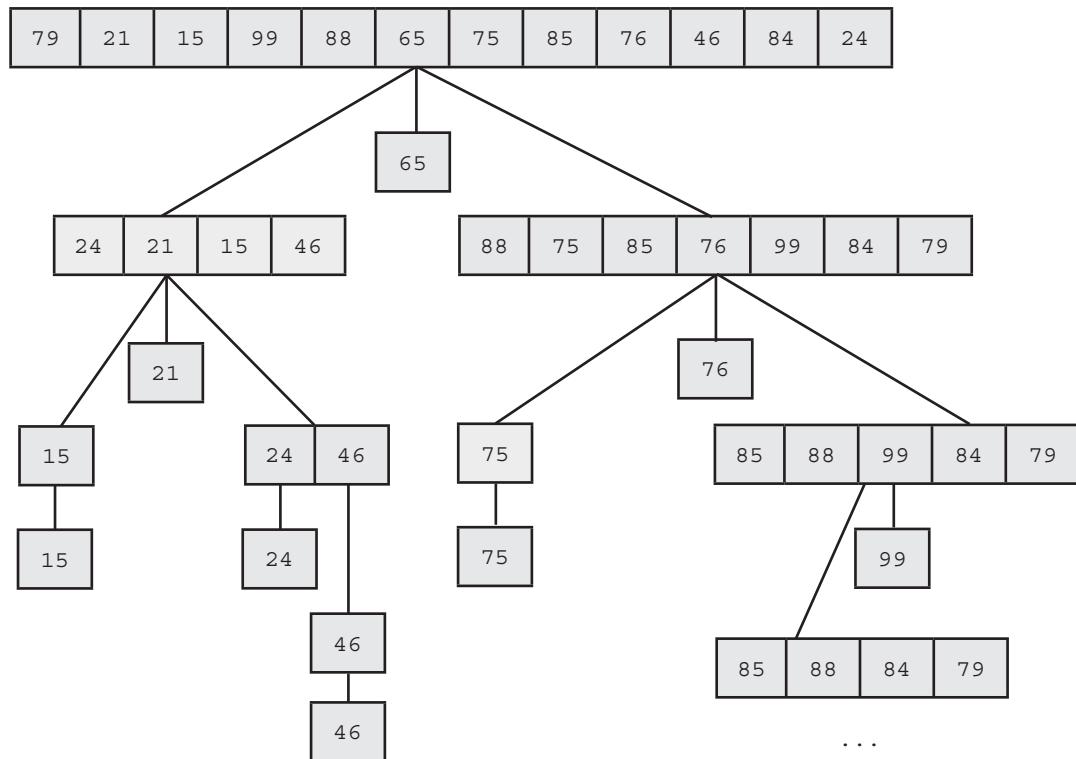
Y la codificación recursiva de la función:

```
void quicksort(double a[], int primero, int ultimo)
{
```

```

int i, j, central;
double pivot;
central = (primero + ultimo)/2;
pivot = a[central];
i = primero;
j = ultimo;
do {
 while (a[i] < pivot) i++;
 while (a[j] > pivot) j--;
 if (i <= j)
 {
 double tmp;
 tmp = a[i];
 a[i] = a[j];
 a[j] = tmp; /* intercambia a[i] con a[j] */
 i++;
 j--;
 }
}while (i <= j);
if (primero < j)
 quicksort(a, primero, j); /* mismo proceso con sublista izquierda */
if (i < ultimo)
 quicksort(a, i, ultimo); /* mismo proceso con sublista derecha */
}

```



Izquierda: 24, 21, 15, 46

Pivote: 65

Derecha: 88, 75, 85, 76, 99, 84, 79

Figura 9.2 Ordenación rápida eligiendo como pivote el elemento central.

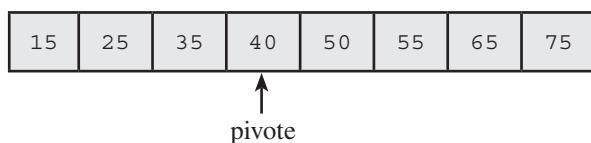
## Análisis del algoritmo *quicksort*

El análisis general de la eficiencia de *quicksort* es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que  $n$  (número de elementos de la lista) es una potencia de 2,  $n = 2^k$  ( $k = \log_2 n$ ). Además, supongamos que el pivote es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos sublistas aproximadamente iguales.

En la primera exploración o recorrido se hacen  $n-1$  comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño  $n/2$ . En la siguiente fase, el proceso de cada sublista requiere aproximadamente  $n/2$  comparaciones. Las comparaciones totales de esta fase son  $2(n/2) = n$ . La siguiente fase procesa cuatro sublistas que requieren un total de  $4(n/4)$  comparaciones, etc. Al final, el proceso de división termina después de  $k$  pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$\begin{aligned} n + 2(n/2) + 4(n/4) + \dots + n(n/n) &= n + n + \dots + n \\ &= n * k = n * \log_2 n \end{aligned}$$

Para una lista normal la complejidad de *quicksort* es  $O(n \log_2 n)$ . El caso ideal que se ha examinado se efectúa en realidad cuando la lista (el arreglo) está ordenado en orden ascendente. En este caso el pivote es precisamente el centro de cada sublista.



Si el arreglo está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada y el algoritmo tiene la complejidad  $O(n \log_2 n)$ .

El escenario del caso peor de *quicksort* ocurre cuando el pivote cae en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay  $n$  comparaciones y la sublista grande contiene  $n-1$  elementos. En el siguiente recorrido, la sublista mayor requiere  $n-1$  comparaciones y produce una sublista de  $n-2$  elementos, etc. El número total de comparaciones es:

$$n + n-1 + n-2 + \dots + 2 = (n-1)(n+2)/2$$

Entonces la complejidad es  $O(n^2)$ . En general el algoritmo de ordenación tiene como complejidad media  $O(n \log_2 n)$  y es posiblemente el algoritmo más rápido. La tabla 9.1 muestra las complejidades de los algoritmos empleados en los métodos de ordenación más importante.

Tabla 9.1 Comparación de la complejidad en los métodos de ordenación.

| Método    | Complejidad  |
|-----------|--------------|
| Burbuja   | $n^2$        |
| Inserción | $n^2$        |
| Selección | $n^2$        |
| Montículo | $n \log_2 n$ |
| Fusión    | $n \log_2 n$ |
| Shell     | $n \log_2 n$ |
| Quicksort | $n \log_2 n$ |

### Consejo de programación

Se recomienda utilizar los algoritmos de inserción o selección para ordenar listas pequeñas. Para listas grandes utilizar el algoritmo *quicksort*.

## 9.6 Búsqueda en listas: búsqueda secuencial y binaria

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arreglos y registros, y por ello será necesario determinar si un arreglo contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un arreglo se denomina *búsqueda*. Las técnicas de búsqueda más utilizadas son: *búsqueda lineal* o *secuencial*, la técnica más sencilla y *búsqueda binaria* o *dicotómica*, la técnica más eficiente. El algoritmo de búsqueda secuencial recorre el arreglo desde el primer elemento hasta encontrar la clave buscada, en esta sección se desarrolla la técnica de búsqueda binaria de forma iterativa.

### Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la indagación de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta, pero normalmente no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con “J” y se está en la “L” se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa uno en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

Se desea buscar si el elemento 225 se encuentra en el conjunto de datos siguiente:

|       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|
| a [0] | a [1] | a [2] | a [3] | a [4] | a [5] | a [6] | a [7] |
| 13    | 44    | 75    | 100   | 120   | 275   | 325   | 510   |

### Ejemplo 9.3

El punto central de la lista es el elemento `a [3]` (100). El valor que se busca es 225 que es mayor que 100; por consiguiente la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es decir, en la sublista:

|       |       |       |       |
|-------|-------|-------|-------|
| a [4] | a [5] | a [6] | a [7] |
| 120   | 275   | 325   | 510   |

Ahora el elemento mitad de esta sublista `a [5]` (275). El valor buscado, 225, es menor que 275 y la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir en la sublista de un único elemento:

|       |
|-------|
| a [4] |
| 120   |

El elemento mitad de esta sublista es el propio elemento `a [4]` (120). Al ser 225 mayor que 120 la búsqueda debe continuar en una sublista vacía. Este ejemplo termina indicando que no se ha encontrado la clave en la lista.

### Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está almacenada como un arreglo, donde los índices de la lista son `bajo = 0` y `alto = n-1`, donde `n` es el número de elementos del arreglo, los pasos a seguir son:

1. Calcular el índice del punto central del arreglo:

central = (bajo + alto)/2 (división entera)

2. Comparar el valor de este elemento central con la clave

- Si  $a[\text{central}] < \text{clave}$ , la nueva sublista de búsqueda está en el rango

bajo = central+1 .. alto

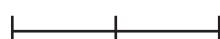
- Si  $\text{clave} < a[\text{central}]$ , la nueva sublista de búsqueda está en el rango

bajo..central-1



bajo      central-1 = alto      bajo = central+1      alto

clave



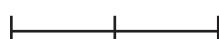
bajo      central      alto  
Clave encontrada

clave



bajo      central      alto  
Búsqueda lista inferior

clave



bajo      central      alto  
Búsqueda lista superior

Figura 9.3 Búsqueda binaria de un elemento.



#### Ejemplo 9.4

Sea el arreglo de enteros A (-8, 4, 5, 9, 12, 18, 25, 40, 60), buscar la clave:  
= 40.

1.  $a[0] \ a[1] \ a[2] \ a[3] \ a[4] \ a[5] \ a[6] \ a[7] \ a[8]$

|    |   |   |   |    |    |    |    |    |
|----|---|---|---|----|----|----|----|----|
| -8 | 4 | 5 | 9 | 12 | 18 | 25 | 40 | 60 |
|----|---|---|---|----|----|----|----|----|

bajo = 0  
alto = 8

↑  
central

$$\text{central} = \frac{\text{bajo} + \text{alto}}{2} = \frac{0 + 8}{2} = 4$$

clave (40) > a[4] (12)

2. Buscar en sublista derecha

|    |    |    |    |
|----|----|----|----|
| 18 | 25 | 40 | 60 |
|----|----|----|----|

bajo = 5  
alto = 8

↑

$$\text{central} = \frac{\text{bajo} + \text{alto}}{2} = \frac{5 + 8}{2} = 6 \text{ (división entera)}$$

clave (40) > a[6] (25)

3. Buscar en sublista derecha

|    |    |
|----|----|
| 40 | 60 |
|----|----|

bajo = 7  
alto = 8

$$\text{central} = \frac{\text{bajo} + \text{alto}}{2} = \frac{7 + 8}{2} = 7$$

clave (40) = a[7] (40) búsqueda con éxito

### Codificación del algoritmo de búsqueda binaria:

```

/*
 búsqueda binaria.
 devuelve el índice del elemento buscado, o bien -1 en caso de fallo
*/
int busquedaBin(int lista[], int n, int clave)
{
 int central, bajo, alto;
 int valorCentral;
 bajo = 0;
 alto = n-1;
 while (bajo <= alto)
 {
 central = (bajo + alto)/2; /* índice de elemento central */
 valorCentral = lista[central]; /* valor del índice central */
 if (clave == valorCentral)
 return central; /* encontrado, devuelve posición */
 else if (clave < valorCentral)
 alto = central -1; /* ir a sublistas inferior */
 else
 bajo = central + 1; /* ir a sublistas superior */
 }
 return -1; /* elemento no encontrado */
}

```

### Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación, cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trate de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

#### Complejidad de la búsqueda secuencial

La búsqueda secuencial recorre la lista desde el primer elemento, su complejidad distingue entre el comportamiento en el peor y mejor caso. El *mejor caso* se da cuando aparece una coincidencia en el primer elemento de la lista y en esa circunstancia el tiempo de ejecución es  $O(1)$ . El peor caso se produce cuando el elemento no está en la lista o se encuentra al final. Esto requiere buscar en todos los  $n$  elementos, lo que implica una complejidad lineal, dependiente de  $n$ ,  $O(n)$ .

El caso medio requiere un poco de razonamiento probabilista. Para el caso de una lista aleatoria es probable que una coincidencia ocurra en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central  $n/2$ . El elemento central ocurre después de  $n/2$  comparaciones, que define el costo esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es  $O(n)$ .

#### Análisis de la búsqueda binaria

El mejor caso de la búsqueda binaria se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es  $O(1)$  dado que solo se realiza una prueba de comparación de igualdad. La complejidad del *peor caso* es  $O(\log_2 n)$  que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El peor caso se produce cuando se debe continuar la búsqueda y llegar a una sublistas de longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublistas por un factor de 2. El tamaño de las sublistas es:

$n \ n/2 \ n/4 \ n/8 \dots 1$

La división de sublistas requiere  $m$  iteraciones, en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

$$n \ n/2 \ n/2^2 \ n/2^3 \ n/2^4 \ \dots \ n/2^m$$

donde  $n/2^m = 1$

Tomando logaritmos en base 2 en la expresión anterior quedará:

$$n = 2^m$$

$$m = \log_2 n$$

Por esa razón la complejidad del peor caso es  $O(\log_2 n)$ . Cada iteración requiere una operación de comparación:

$$\text{Total de comparaciones} \approx 1 + \log_2 n$$

#### Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista como representa su complejidad  $O(n)$ .

Sin embargo, en el caso de la búsqueda binaria, el número máximo de comparaciones es  $1 + \log_2 n$ . Si por ejemplo, la lista tiene  $2^{10} = 1024$  elementos, el total de comparaciones será  $1 + \log_2 2^{10}$ ; es decir, 11. Para una lista de 2048 elementos y teniendo presente que  $2^{11} = 2048$  implicará que el número máximo de elementos examinados (comparaciones) en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número  $m$  más pequeño para una lista de 1 000 000, tal que:

$$2^m \geq 1\,000\,000$$

Es decir  $2^{19} = 524\,288$ ,  $2^{20} = 1\,048\,576$  y por lo tanto el número de elementos examinados (en el peor de los casos) es 21.

La tabla 9.2 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsqueda secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial y cuyos resultados de tiempo vienen dados por las funciones de complejidad  $O(\log_2 n)$  y  $O(n)$  de las búsquedas binaria y secuencial, respectivamente.

#### A recordar

La búsqueda secuencial se aplica para localizar una clave en una lista no ordenada. Para aplicar el algoritmo de búsqueda binaria la lista, o vector, donde se busca debe estar ordenado. La complejidad de la búsqueda binaria es logarítmica,  $O(\log n)$ , más eficiente que la búsqueda secuencial que tiene complejidad lineal,  $O(n)$ .

**Tabla 9.2** Comparación de las búsquedas binaria y secuencial.

| Números de elementos examinados |                  |                     |
|---------------------------------|------------------|---------------------|
| Tamaño de la lista              | Búsqueda binaria | Búsqueda secuencial |
| 1                               | 1                | 1                   |
| 10                              | 4                | 10                  |
| 1 000                           | 11               | 1 000               |
| 5 000                           | 14               | 5 000               |
| 100 000                         | 18               | 100 000             |
| 1 000 000                       | 21               | 1 000 000           |



## Resumen

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*.
- Los datos se pueden ordenar en forma ascendente o descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración*.
- Los algoritmos de ordenación básicos son:
  - Selección.
  - Inserción.
  - Burbuja.
- Los algoritmos de ordenación más avanzados son:
  - *Shell*.
  - *Quicksort*.
  - *Heapsort* (por montículos).
  - *Mergesort*.
- La eficiencia de los algoritmos de la burbuja, inserción y selección es  $O(n^2)$ .
- La eficiencia del algoritmo *quicksort* es  $O(n \log n)$ .
- La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista.
- Existen dos métodos básicos de búsqueda en arreglos: **búsqueda secuencial** y **binaria**.
- La **búsqueda secuencial** se utiliza normalmente cuando el arreglo no está ordenado. Comienza en el principio del arreglo y busca hasta que se encuentra el dato requerido o se llega al final de la lista.
- Si un arreglo está ordenado se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria**.
- La eficiencia de una búsqueda secuencial es  $O(n)$ .
- La eficiencia de una búsqueda binaria es  $O(\log n)$ .



## Ejercicios

9.1 ¿Cuál es la diferencia entre ordenación por selección y ordenación por inserción?

9.2 Se desea eliminar todos los números duplicados de una lista o vector (*array*). Por ejemplo, si el arreglo toma los valores:

4    7    11    4    9    5    11    7    3    5

ha de cambiarse

4    7    11    9    5    3

Escribir una función que elimine los elementos duplicados de un arreglo.

9.3 Escribir una función que elimine los elementos duplicados de un arreglo ordenado. ¿Cuál es la eficiencia de esta función? Compare la eficiencia con la que tiene la función del ejercicio 9.2.

9.4 Un arreglo contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción.

¿Cuál será el valor de los elementos del arreglo después de tres pasadas más del algoritmo?

|   |    |   |    |    |    |    |    |
|---|----|---|----|----|----|----|----|
| 3 | 13 | 8 | 25 | 45 | 23 | 98 | 58 |
|---|----|---|----|----|----|----|----|

9.5 Dada la siguiente lista:

|    |   |    |    |    |    |
|----|---|----|----|----|----|
| 47 | 3 | 21 | 32 | 56 | 92 |
|----|---|----|----|----|----|

Después de dos pasadas de un algoritmo de ordenación, el arreglo ha quedado dispuesto así:

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 3 | 21 | 47 | 32 | 56 | 92 |
|---|----|----|----|----|----|

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

9.6 Un arreglo contiene los elementos indicados más abajo. Utilizando el algoritmo de ordenación *burbuja* encuentre las pasadas y los intercambios que se realizan para su ordenación.

|   |    |    |   |    |    |    |    |    |   |
|---|----|----|---|----|----|----|----|----|---|
| 8 | 43 | 17 | 6 | 40 | 16 | 18 | 97 | 11 | 7 |
|---|----|----|---|----|----|----|----|----|---|

9.7 Partiendo del mismo arreglo que en el ejercicio 9.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación *quicksort* para su ordenación.

9.8 Un arreglo de registros se quiere ordenar según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: día, mes y año de 2, 2 y 4 dígitos, respectivamente. Adaptar el método de selección a esta ordenación.

- 9.9 Un arreglo contiene los elementos indicados a continuación. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

|   |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|
| 8 | 13 | 17 | 26 | 44 | 56 | 88 | 97 |
|---|----|----|----|----|----|----|----|

Igual búsqueda pero para el número 20.

- 9.10 Escribir la función de ordenación correspondiente al método *inserción* para poner en orden alfabético una lista de  $n$  nombres.

- 9.11 Escribir una función de búsqueda binaria aplicada a un arreglo ordenado descendente.

- 9.12 Supongamos que se tiene una secuencia de  $n$  números que deben ser clasificados:

- Si se utiliza el método de *selección*, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
  - ¿ya está clasificado?
  - ¿está en orden inverso?
- Realizar los mismos cálculos si se utiliza el algoritmo *quicksort*.

## Problemas

- 9.1 Un método de ordenación muy simple, pero no muy eficiente, de elementos  $x_1, x_2, x_3, \dots, x_n$  en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño de la lista  $x_1$  a  $x_n$ ; intercambiarlo con  $x_1$ .

Paso 2: Localizar el elemento más pequeño de la lista  $x_2$  a  $x_n$ ; intercambiarlo con  $x_2$ .

Paso 3: Localizar el elemento más pequeño de la lista  $x_3$  a  $x_n$ ; intercambiarlo con  $x_n$ .

En el último paso, los dos últimos elementos se comparan e intercambian, si es necesario, y la ordenación se termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

- 9.2 Dado un vector  $x$  de  $n$  elementos reales, donde  $n$  es impar, diseñar una función que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que el valor y la otra mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que dé prueba de la función.

- 9.3 Se trata de resolver el siguiente problema escolar. Dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales.

**Nota:** utilizar como algoritmo de ordenación el método de la *burbuja*.

- 9.4 Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1 000 nombres y números de

teléfono de un archivo que contiene los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

- 9.5 Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2000 enteros aleatorios en el rango 0...1 999 y a continuación se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

- 9.6 Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier acción hecha con *Maestro* [ $i$ ] debe hacerse con *Esclavo* [ $i$ ]. Después de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea.

**Nota:** utilizar como algoritmo de ordenación el método *Quicksort*.

- 9.7 Cada línea de un archivo de datos contiene información sobre la compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y a continuación se visualiza. La información debe ser ordenada por venta de mayor a menor y visualizada de nuevo.

- 9.8 Se desea realizar un programa que realice las siguientes tareas:

- a) Generar, aleatoriamente, una lista de 999 números reales en el rango de 0 a 2000.
- b) Ordenar en modo creciente por el método de la burbuja.
- c) Ordenar en modo creciente por el método de *selección*.
- d) Buscar si existe el número  $x$  (leído del teclado) en la lista. Aplicar la búsqueda binaria.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

- t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.

- t2. Tiempo que se emplearía en *ordenar* la lista ya dispuesta.
- t3. Tiempo empleado en ordenar la lista establecida en forma inversa.

- 9.9 Se leen dos listas de números enteros, A y B de 100 y 60 elementos, respectivamente. Se desea resolver las siguientes tareas:

- a) Ordenar, aplicando el método de inserción, cada una de las listas A y B.
- b) Crear una lista C por intercalación o mezcla de las listas A y B.
- c) Visualizar la lista C ordenada.



# Estructuras y uniones

## Contenido

- 10.1 Estructuras
- 10.2 Acceso a estructuras
- 10.3 Sinónimo de un tipo de dato: `typedef`
- 10.4 Estructuras anidadas
- 10.5 Arreglos de estructuras
- 10.6 Arreglos como miembros

## 10.7 Utilización de estructuras como parámetros

- 10.8 Uniones
- 10.9 Tamaño de estructuras y uniones
  - › Resumen
  - › Ejercicios
  - › Problemas

## 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 el arreglo o la estructura, pero sí necesario en algunos casos.

Con los campos de bits se puede trabajar con un bit o un grupo de bits de una palabra entera. Un `typedef`, de hecho, no es un nuevo tipo de dato sino simplemente un sinónimo de un tipo existente.

### 10.1 Estructuras

#### Conceptos clave

- › Estructura
- › Estructuras anidadas
- › `sizeof`
- › `struct`
- › `typedef`
- › `union`

Los arreglos 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 arreglo; sin embargo, si se necesita una lista de información de clientes que contenga elementos tales como el nombre, la edad, la dirección, el número de la cuenta, etc., los arreglos no son adecuados. La solución a este problema es utilizar un *tipo de dato registro*, denominada estructura en C.

Los componentes individuales de una estructura se llaman *miembros*. Cada miembro (elemento) de una estructura puede contener valores de un tipo diferente de datos. Por ejemplo, se puede utilizar una estructura para almacenar distintos 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*.

### A recordar

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 de miembro. Por ejemplo, suponga 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 del miembro  | Tipo de dato                       |
|---------------------|------------------------------------|
| Título              | Arreglo de caracteres de tamaño 30 |
| Artista             | Arreglo de caracteres de tamaño 25 |
| Número de canciones | Entero                             |
| Precio              | Coma flotante                      |
| Fecha de compra     | Arreglo de caracteres de tamaño 8  |

La figura 10.1 contiene la estructura CD; muestra gráficamente los tipos de datos dentro de la estructura. Observe que cada miembro es un tipo de dato diferente.

### 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 miembro1> <nombre miembro1>
 <tipo de dato miembro2> <nombre miembro2>
 ...
 <tipo de dato miembron> <nombre miembron>
};
```

|                     |                                   |
|---------------------|-----------------------------------|
| Título              | Ay, ay, ay, cómo se aleja el sol. |
| Artista             | No me pisces las sandalias.       |
| Número de canciones | 10                                |
| Precio              | 2222.25                           |
| Fecha de compra     | 8-10-1999                         |

Figura 10.1 Representación gráfica de una estructura CD.

La declaración de la estructura CD es:

```
struct 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 vendedor[30];
 unsigned int codigo;
 int inids_articulos;
 float precio_unit;
};
```

### 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>`. Sin embargo, cada definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo con el 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,
2. Listando el tipo de la estructura creado seguido 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;
} libro1, libro2, libro3;

2. struct info_libro libro1, 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 v1, s1, c1;
```

## 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 `libro1`. Por consiguiente, sería necesario realizar las siguientes sentencias:

```
libro3 = libro1;
libro4 = libro1;
libro5 = libro1;
```

De modo alternativo se puede escribir

```
libro5 = libro4 = libro3 = libro1;
```

## Inicialización de una declaración de estructuras

Se puede inicializar una estructura de dos formas. 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 es:

```
struct <tipo> <nombre variable estructura> =
{ valor miembro_1,
```

```

 valor miembro2,
 ...
 valor miembron
 };
 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:

```

structleccion_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 v1 = {
 "Salvador Rapido",
 29,
 'V',
 "Senior",
 "Independiente",
 0.0
};

```

## 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 ejemplo 10.1 ilustra el uso del operador `sizeof` para determinar el tamaño de una estructura.



### Ejemplo 10.1

Se declara una estructura para representar a una persona y se determina el tamaño.

```

#include <stdio.h>
struct persona
{
 char nombre[30];
 int edad;
 float altura;
 float peso;
};
void main()

```

```

{
 struct persona mar;
 printf("Sizeof(persona) : %d \n", sizeof(mar));
}

```

### Ejecución

```
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> |               | 40             |

## 10.2 Acceso a estructuras

Se accede a una estructura, bien para almacenar información en la estructura o bien para recuperar la información de esta. Se puede acceder a los miembros de una estructura de una de estas dos formas:

1. utilizando el operador punto (.)
2. utilizando el operador apuntador ->.

### 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 leen 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)
 strcpy(cr.categoría, "Juvenil");
else if (cr.edad <= 40)
 strcpy(cr.categoría, "Senior");
else
 strcpy(cr.categoría, "Veterano");

```

Acceso a una estructura de datos mediante el operador apuntador

El operador apuntador,<sup>1</sup> `->`, sirve para acceder a los datos de la estructura a partir de un apuntador. Para utilizar este operador se debe definir primero una variable apuntador para apuntar a la estructura. A continuación, utilice simplemente el operador apuntador para apuntar a un miembro dado.

La asignación de datos a estructuras utilizando el operador apuntador (puntero) tiene el formato:

```
<puntero estructura> -> <nombre miembro> = datos;
```

Así, por ejemplo, una estructura estudiante:

```

struct estudiante
{
 char Nombre[41];
 int Num_Estudiante;
 int Anyo_de_matricula;
 float Nota;
};

```

Se puede definir `ptr_est` como un apuntador 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_Estudiante = 3425;
ptr_est -> Nota = 8.5;

```

### Consejo de programación

Antes de acceder a los miembros de una estructura con un variable apuntador y el operador `->`, es preciso crear espacio de almacenamiento en memoria; por ejemplo, con la función `malloc()`.

### 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 (apuntador). 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:

<sup>1</sup> En el capítulo 11 se estudian los apuntadores.

```

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);
/* cálculo del módulo */
z.modulo = sqrt(z.pr*z.pr + z.pi*z.pi);

```

## Recuperación de información de una estructura

Se recupera información de una estructura utilizando el operador de asignación o una sentencia de salida (`printf()`, `puts()`, ...). Para acceder a los miembros se utiliza el operador punto o el operador flecha (apuntador). El formato general toma una de estas dos formas:

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);

```

## 10.3 Sinónimo de un tipo de datos: `typedef`

La sentencia `typedef` permite al 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;
```

A partir de la sentencia anterior, `Longitud` se puede utilizar como un tipo de dato, en este ejemplo sinónimo de `double`. La función `Distancia()`, escrita a continuación, es de tipo `Longitud`:

```

Longitud Distancia (const Punto& p, const Punto& p2)
{
 ...
 Longitud longitud = sqrt(r-cua);
 return longitud;
}

```

Otros ejemplos:

```
typedef char* String;
typedef const char* string;
```

A continuación se pueden hacer las siguientes declaraciones con la palabra `String` o `string`:

```
String nombre = "Jesus Lopez Arollo";
string concatena(string apelli, string apell2);
```

### Sintaxis

```
typedef tipo_dato_definido nuevo_nombre
```

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];
```

### Ejemplo

Declaración del tipo de dato `racional` y asociación a `Racional`.

```
typedef struct rational
{
 int numerador;
 int denominador;
} Racional;
```

Ahora se puede declarar la estructura `numero` utilizando el tipo `complex` y el tipo `rational`:

```
struct numero
{
 complex a;
 Racional r;
};
```

### A recordar

La ventaja de `typedef` es que permite dar nombres de tipos de datos más acordes con aquello que representan en una determinada aplicación.

## 10.4 Estructuras anidadas

Una estructura puede contener otras estructuras llamadas *estructuras anidadas*, las cuales ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes solo 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 cliente
{
 char nombre_cliente[30];
 char direccion[25];
 char ciudad[20];
 char provincia[20];
 long int cod_postal;
 double saldo;
};

```

Estas estructuras contienen 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;
};

```

Una 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 cliente
{
 char nombre_cliente[30];
 struct info_dir direccion_clien;
 double saldo;
};

```

La figura 10.2 muestra gráficamente las estructuras anidadas `empleado` y `cliente`.

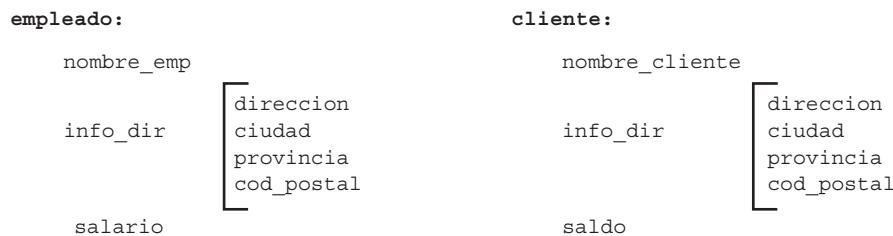


Figura 10.2 Estructuras anidadas.

## 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 llevar a cabo 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 demás campos. La representación del tipo de operación se hace con una variable de tipo enumerado. A continuación se declaran estos tipos, se escribe una función que lee una operación financiera y devuelve la operación leída.

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

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

struct tiempo
{
 unsigned int horas, minutos;
};

enum tipo_operacion {deposito, retirada, aldia, estado};

typedef struct fecha Fecha;
typedef struct tiempo Tiempo;
typedef enum tipo_operacion TipOperacion;

struct registro_operacion
{
 long numero_cuenta;
 float cantidad;
 TipOperacion operacion;
 Fecha f;
 Tiempo t;
};

typedef struct registro_operacion RegistrOperacion;
RegistrOperacion entrada(void);

int main()
{
 RegistrOperacion w;
 w = entrada();
 printf("\n Operación realizada\n\n");
 printf("\t%ld\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;
}

RegistrOperacion entrada(void)
{
 int x, y, z;
 RegistrOperacion una;
 printf("\nNúmero de cuenta: ");
 scanf("%ld", &una.numero_cuenta);
 puts("\tTipo de operación");
 puts("Deposito(0)");
```

```

 puts("Retirada de fondos(1)");
 puts("Puesta al dia(2)");
 puts("Estado de la cuenta(3)");
 scanf("%d",&una.operacion);

 printf("\nFecha (dia mes año): ");
 scanf("%d %d %d",&una.f.dia,&una.f.mes,&una.f.anyo);

 printf("Hora de la operacion(hora minuto): ");
 scanf("%d %d",&una.t.horas,&una.t.minutos);

 return una;
}

```

### A recordar

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`.

```
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}};
```

## 10.5 Arreglos de estructuras

Se puede crear un arreglo (array) de estructuras tal como se crea un arreglo de otros tipos. Los arreglos 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 arreglos proporcionan un medio práctico de almacenar diversos valores del mismo tipo, los arreglos de estructuras le permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras.

Muchos programadores de C utilizan arreglos de estructuras como un método para almacenar datos en un archivo de disco. Se pueden introducir y calcular sus datos de disco en arreglos de estructuras y a continuación almacenar esas estructuras en memoria. Los arreglos de estructuras proporcionan también un medio de guardar datos que se leen del disco.

La declaración de un arreglo de estructuras `info_libro` se puede hacer de un modo similar a cualquier arreglo, es decir,

```
struct info_libro libros[100];
```

asigna un arreglo de 100 elementos denominado `libros`. Para acceder a los miembros de cada uno de los elementos estructura se utiliza una notación de arreglo. 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 arreglo 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", 1997};

```

En el siguiente ejemplo se declara una estructura que representa a un número racional, un arreglo de números racionales es inicializado con valores al azar.

```
struct racional
{
 int N,
 int D;
};

struct racional rs[4] = { 1,2, 2,3, -4,7, 0,1};
```

## 10.6 Arreglos como miembros

Los miembros de las estructuras pueden ser de cualquier tipo y en particular arreglos. En este caso, será preciso extremar las precauciones cuando se accede a los elementos individuales del arreglo.

Considérese la siguiente definición de estructura. Esta sentencia declara un arreglo 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 10.2

Una librería desea catalogar su inventario de libros. El siguiente programa crea un arreglo de 100 estructuras, donde cada estructura contiene diversos tipos de variables, incluyendo arreglos.

```
#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;
};

typedef struct inventario inventario;
int main()
{
 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?\n");
gets(libro[total].autor);

printf("¿Cuántos libros existen?: ");
scanf("%d",&libro[total].num);

printf("¿Cuántos ejemplares están pedidos?: ");
scanf("%d%c",&libro[total].pedido);
printf("¿Cuál es el precio de venta?: ");
gets(b);
libro[total].precio_venta = atof(b); /* conversión a real */

printf("\n ¿Hay más libros? (S/N) ");
scanf("%c%c",&resp);

resp = toupper(resp); /* convierte a mayúsculas */
if (resp == 'S')
{
 total++;
 continue;
}
} while (resp == 'S');
return 0;
}

```

## 10.7 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`, en el paso por valor, puede ser prohibitivo. En tales casos, se debe considerar el método de pasar la dirección de la estructura.

El ejemplo 10.3 muestra el paso de una estructura por dirección y por valor a una función, así como la declaración de argumento de la función en cada caso.

Solicitar los datos de entrada de una estructura que represente a una persona y mostrarlos por pantalla. Con una función de entrada y otra de salida.

### Ejemplo 10.3

```

#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 entrad_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);
}

```

### A recordar

Si se desea pasar una estructura por referencia, necesita situar un operador de referencia & antes de la variable estructura en la llamada a la función. El parámetro correspondiente debe de ser tipo apuntador a la estructura. El acceso a miembros dato de la estructura se realiza a partir de un apuntador y requiere el uso del selector ->.

## 10.8 Uniones

Las uniones son similares a las estructuras en cuanto que agrupan a una serie de variables, pero la forma de almacenamiento es distinta y, por consiguiente, efectos diferentes. Una estructura (struct) permite almacenar variables relacionadas juntas y almacenadas en posiciones contiguas en memoria. Las uniones se declaran 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 analizando el tamaño de cada variable de la unión; el mayor de los tamaños de las variables 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 al de la variable que ocupe más memoria. 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 y ocupa 8 bytes.

```
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 diversas 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 arreglos de caracteres, tales como los 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 solo 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 apuntador a unión. Así:

```
cadenas.ayuda;
cadenas.mensaje_error;
pc -> mensaje_error;
```

## 10.9 Tamaño de estructuras y uniones

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 es:

```

sizeof(tipo_estructura):15
sizeof(tipo_union): 8
sizeof(monedas): 2

```



## Resumen

Una estructura permite que los miembros dato de los mismos o diferentes tipos se encapsulen en una única implementación, al contrario que los arreglos 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 solo 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 regempleado;
```



## Ejercicios

10.1 Encontrar 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];

```

10.2 Declarar un tipo de datos para representar las estaciones del año.

10.3 Escribir una función que devuelva la estación del año que se ha leído del teclado. La función debe ser del tipo declarado en el ejercicio 10.2.

10.4 Declarar un tipo de dato para representar una cuenta bancaria.

10.5 Encontrar 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,
}
```

10.6 ¿Con `typedef` se declaran nuevos tipos de datos, o bien permite cambiar el nombre de tipos de datos ya declarados?

10.7 Declarar un tipo de dato estructura para representar a un alumno; los campos que debe tener son: nombre, curso, edad, dirección y notas de las 10 asignaturas. Declarar otro tipo estructura para representar a un profesor; los campos que debe tener son: nombre, asignaturas que imparte y dirección. Por último, declarar una estructura que pueda representar a un profesor o a un alumno.

10.8 Definir tres variables correspondientes a los tres tipos de datos declarados en el ejercicio anterior.

10.9 Escribir una función que devuelva un profesor o un alumno cuyos datos se introducen por teclado.

10.10 Escribir la misma función que en el ejercicio anterior, pero pasando la estructura como argumento a la función.

10.11 Escribir una función que tenga como entrada una estructura, profesor o alumno, y escribir sus campos por pantalla.

## Problemas

10.1 Escribir un programa para calcular el número de días que hay entre dos fechas; declarar fecha como una estructura.

10.2 Escribir 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: moroso, atrasado, pagado. El programa debe generar a los diversos clientes.

10.3 Modificar el programa de facturación de clientes de tal modo que puedan obtenerse los siguientes listados.

- Clientes en estado moroso.
- Clientes en estado pagado con factura mayor de una determinada cantidad.

10.4 Escribir 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.

10.5 Un número racional se caracteriza por el numerador y el denominador. Escribir 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.

10.6 Se quiere informatizar los resultados obtenidos por los equipos de baloncesto y de fútbol de la localidad alcarreña Lupiana. La información de cada equipo es:

- Nombre del equipo.
- Número de victorias.
- Número de derrotas.

Para los equipos de baloncesto añadir la información:

- Números de pérdidas de balón.
- Número de rebotes cogidos.
- Nombre del mejor anotador de triples.
- Número de triples del mejor anotador de triples.

Para los equipos de fútbol añadir la información:

- Número de empates.
- Número de goles a favor.
- Nombre del goleador del equipo.
- Número de goles del goleador.

Escribir un programa para introducir la información para todos los integrantes en ambas ligas.

10.7 Modificar el problema 10.6 para obtener los siguientes informes o datos.

- Listado de los mejores anotadores de triples de cada equipo.
- Máximo goleador de la liga de fútbol.
- Suponiendo que el partido ganado significa tres puntos y el empate 1 punto: equipo ganador de la liga del fútbol.
- Equipo ganador de la liga de baloncesto.

10.8 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.



# Apuntadores (punteros)

## Contenido

- 11.1 Direcciones en memoria
- 11.2 Concepto de apuntador (puntero)
- 11.3 Apuntadores `NULL` y `void`
- 11.4 Apuntadores a apuntadores
- 11.5 Apuntadores y arreglos
- 11.6 Arreglos (arrays) de apuntadores
- 11.7 Apuntadores a cadenas
- 11.8 Aritmética de apuntadores
- 11.9 Apuntadores constantes frente a apuntadores a constantes
- 11.10 Apuntadores como argumentos de funciones

- 11.11 Apuntadores a funciones
- 11.12 Apuntadores a estructuras
- 11.13 Asignación dinámica de la memoria
- 11.14 Función de asignación de memoria `malloc( )`
- 11.15 La función `free( )`
- 11.16 Funciones de asignación de memoria: `calloc( )` y `realloc( )`
- 11.17 Reglas de funcionamiento de la asignación dinámica
  - › Resumen
  - › Ejercicios
  - › Problemas

## Introducción

Los apuntadores (punteros) en C tienen fama, en el mundo de la programación, de ser difíciles, tanto en el aprendizaje como en su uso. En este capítulo se tratará de mostrar que los apuntadores no son más difíciles de aprender que cualquier otra herramienta de programación ya examinada o por examinar a lo largo de este libro. El apuntador no es más que una herramienta muy potente que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los apuntadores son, sin género de dudas, una de las razones fundamentales para que el lenguaje C sea tan potente y tan utilizado.

Una *variable apuntador* o *variable puntero* (*apuntador* [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 apuntador contienen valores que son direcciones de memoria donde se almacenan datos. En resumen, un apuntador es una variable que contiene una dirección de memoria, y utilizando apuntadores 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 apuntadores:

- Apuntadores (punteros);
- Utilización de apuntadores;
- Asignación dinámica de memoria;



### Conceptos clave

- › Apuntador (puntero)
- › Arreglo dinámico
- › Arreglo estático
- › Arreglos (arrays) de apuntadores
- › Aritmética de apuntadores
- › Asignación dinámica de la memoria
- › Direcciones
- › Función `free`
- › Función `malloc`
- › `NULL`
- › Palabra reservada `const`
- › Tipos de apuntadores
- › `void`

- Aritmética de apuntadores;
- Arreglos (*arrays*) de apuntadores;
- Apuntadores a apuntadores, funciones y estructuras.

En este capítulo también analizamos que 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 solo 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 solo 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 cuánta 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.

## 11.1 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 11.1

```
int n;
```

0x4ffffd34



int

La caja del ejemplo 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:

0x4ffffd34



int

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 mediante el *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 muy alta con el mismo nivel que el operador lógico NOT (!) y el operador de preincremento ++ (véase capítulo 4).

Obtener el valor y la dirección de una variable.

### Ejemplo 11.2

```
#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.

## 11.2 Concepto de apuntador (puntero)<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 *apuntador*.

Una variable que se declara en C tiene una dirección asociada con ella. Un *apuntador* es una dirección de memoria. El concepto de apuntadores tiene correspondencia en la vida diaria. Cuando usted envía una carta por correo, su información se entrega con base en un apuntador que es la dirección de esa carta. Cuando usted telefonea a una persona, utiliza un apuntador (el número de teléfono que marca). Así pues, una dirección de correo y un número de teléfono tienen en común que ambos indican dónde encontrar algo. Son apuntadores a edificios y teléfonos, respectivamente. Un apuntador en C también indica dónde encontrar algo, ¿dónde encontrar los datos que están asociados con una variable? *Un apuntador C es la dirección de una variable*. Los apuntadores se rigen por estas reglas básicas:

- Un *apuntador* es una *variable* como cualquier otra;
- Una *variable* apuntador contiene una *dirección* que apunta a otra posición en memoria;
- En esa *posición* se almacenan los datos a los que apunta el apuntador;
- Un apuntador apunta a una variable de memoria.

#### A recordar

El tipo de variable que almacena una dirección se denomina *apuntador*.

<sup>1</sup> En Latinoamérica es usual emplear el término *apuntador* y en España, *puntero*.

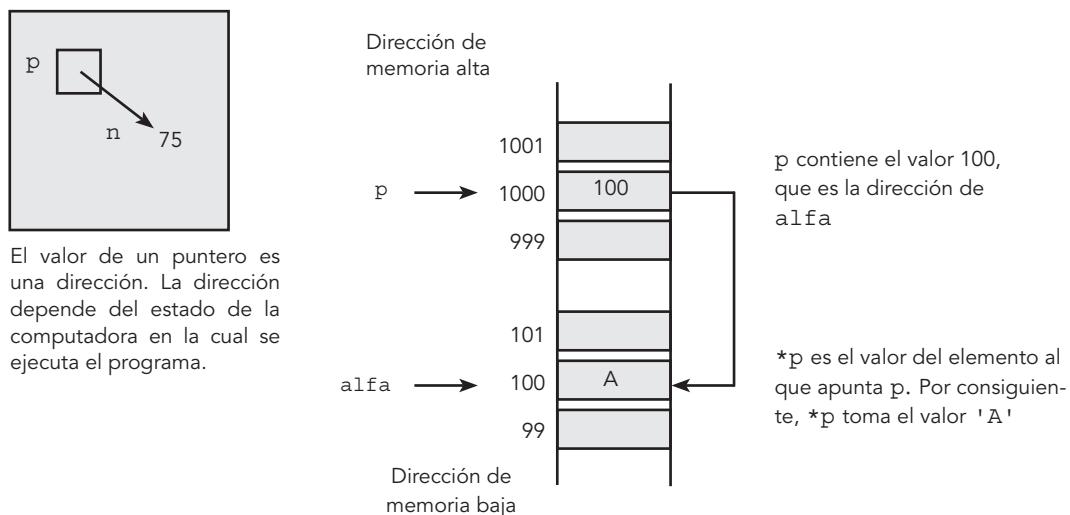


Figura 11.1 Relaciones entre  $*p$  y el valor de  $p$  (dirección de  $alfa$ ).



### Ejemplo 11.3

El programa siguiente es un reflejo de la figura 11.1.

```
#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 = 0x4ffffd34, p = 0x4ffffd34
&p = 0x4ffffd30
0x4ffffd30 0x4ffffd34
p 0x4ffffd34 n 75
int* int
```

La variable  $p$  se denomina **apuntador** debido a que su valor “apunta” a la posición de otro valor. Es un apuntador **int** cuando el valor al que apunta es de tipo **int** como en el ejemplo anterior.

### Declaración de apuntadores

Al igual que cualquier variable, las variables apuntadores han de ser declaradas antes de utilizarlas. La declaración de una variable apuntador debe indicar al compilador el tipo de dato al que apunta el apuntador; 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 apuntadores:

```
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 recolectará una dirección de un tipo de dato especificado. La variable `ptr1` almacenará la dirección de un entero, la variable `ptr2` acumulará la dirección de un dato tipo `long`, etcétera.

### A recordar

Siempre que aparezca un asterisco (\*) en una definición de una variable, esta es una variable apuntador.

## Inicialización<sup>2</sup> (iniciación) de apuntadores

Al igual que otras variables, C no inicializa los apuntadores cuando se declaran y es preciso inicializarlos antes de su uso. La inicialización de un apuntador proporciona a ese apuntador la dirección del dato correspondiente. Después de la inicialización, se puede utilizar el apuntador para referenciar los datos direccionados. Para asignar una dirección de memoria a un apuntador 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 apuntador 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 apuntador, el asterisco delante de la variable apuntador indica *el contenido de* la memoria apuntada por el apuntador 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 apuntador a esa variable se puede cambiar, pero permanecerá la cantidad de memoria reservada.

### A recordar

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; /* Se corresponde con la figura 11.1 */
 char alfa = 'A';
 p = &alfa;
3. char cd[] = "Compacto";
 char *c;
 c = cd; /* c tiene la dirección de la cadena cd */
```

<sup>2</sup> El diccionario de la Real Academia Española solo acepta el término **iniciación** (acción y efecto de iniciar o iniciarse) y no **inicialización**, aunque sí acepta **inicializar** (establecer los valores iniciales para la ejecución de un programa).

### Precaución

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 */
```

Existe un segundo método para inicializar un apuntador, es mediante *asignación dinámica de memoria*. Este método utiliza las funciones de asignación de memoria `malloc()`, `calloc()`, `realloc()` y `free()`.

### Indirección de apuntadores

Después de definir una variable apuntador, el siguiente paso es inicializar el apuntador y utilizarlo para direccionar algún dato específico en memoria. El uso de un apuntador para obtener el valor al que apunta, es decir, su dato apuntado se denomina *indireccionar el puntero* (“desreferenciar el apuntador”); 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 11.2. 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 desreferenciando el apuntador a `edad`:

```
printf("%d", *p_edad); /* indirecciona p_edad */
```

El listado del siguiente programa muestra el concepto de creación, inicialización e indirección de una variable apuntador.

```
#include <stdio.h>
char c; /* variable global de tipo carácter */
int main()
{
 char *pc; /* un puntero a una variable carácter */
 pc = &c;
 for (c = 'A'; c <= 'Z'; c++);
 printf("%c", *pc);
 return 0;
}
```

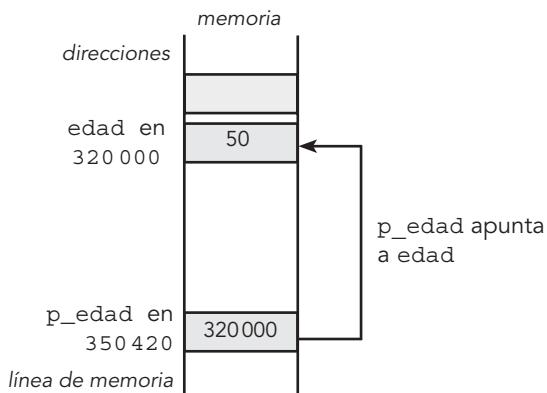


Figura 11.2 `p_edad` contiene la dirección de `edad`, `p_edad` apunta a la variable `edad`.

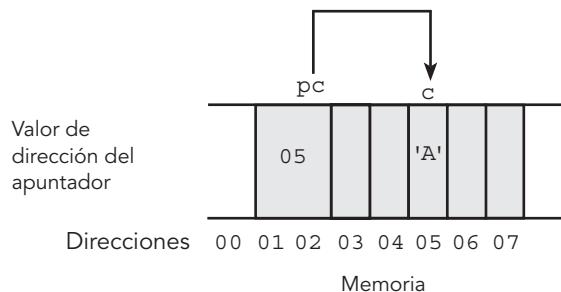


Figura 11.3 pc y c direccionan la misma posición de memoria.

La ejecución de este programa visualiza el alfabeto. La variable `pc` es un apuntador 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. La variable `c`, que se almacena en cualquier parte de la memoria, y `pc`, que apunta a esa misma posición, se refieren 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`.

La tabla 11.1 resume los operadores de apuntadores.

Tabla 11.1 / Operadores de apuntadores.

| Operador           | Propósito                                       |
|--------------------|-------------------------------------------------|
| <code>&amp;</code> | Obtiene la dirección de una variable.           |
| <code>*</code>     | Define una variable como apuntador.             |
| <code>*</code>     | Obtiene el contenido de una variable apuntador. |

### A recordar

Son variables apuntadores aquellas que apuntan a la posición en donde otra(s) variable(s) de programa se almacenan.

### Apuntadores y verificación de tipos

Los apuntadores 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 apuntador. Así, por ejemplo, si se define un apuntador 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 apuntador que apunta a datos de tipo real, `float`.

### Precaución

C requiere que las variables apuntador direccionen realmente variables del mismo tipo de dato que está ligado a los apuntadores en sus declaraciones.

### 11.3 Apuntadores NULL y void

Normalmente un apuntador inicializado de manera adecuada apunta a alguna posición específica de la memoria. Sin embargo, un apuntador no inicializado, como cualquier variable, tiene un valor aleatorio hasta que se inicializa el apuntador. En consecuencia, será preciso asegurarse de que las variables apuntador utilicen direcciones de memoria válida.

Existen dos tipos de apuntadores especiales muy utilizados en el tratamiento de sus programas: los apuntadores `void` y `NULL` (nulo).

Un *apuntador nulo* no apunta a ninguna parte, es decir, un apuntador nulo no direcciona ningún dato válido en memoria. Este apuntador se utiliza para proporcionar a un programa un medio de conocer cuando una variable apuntador no direcciona a un dato válido. Para declarar un apuntador 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 apuntador a nulo es:

```
char *p = NULL;
```

Algunas funciones C también devuelven el valor `NULL` si se encuentra un error. Se puede añadir un test para el valor `NULL` comparando el apuntador 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 apuntador nulo es asignar un valor de 0. Por ejemplo,

```
int *ptr = (int *) 0; /* ptr es un puntero nulo */
```

La conversión de tipos (*casting*) anterior (`int *`), no es necesario, hay una conversión estándar de 0 a una variable apuntador.

```
int *ptr = 0;
```

Nunca se utiliza un apuntador nulo para referenciar un valor. Como antes se ha comentado, los apuntadores nulos se utilizan en un test condicional para determinar si un apuntador 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 apuntador es válido y no es un apuntador nulo.

En C se puede declarar un apuntador 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 apuntador como un apuntador `void*`, denominado apuntador genérico.

```
void *ptr; /* declara un puntero void, puntero genérico */
```

El apuntador `ptr` puede direccionar cualquier posición en memoria, pero el apuntador no está unido a un tipo de dato específico. De modo similar, los apuntadores `void` pueden direccionar una variable `float`, una `char`, o una posición arbitraria o una cadena.

### Precaución

No confundir apuntadores `void` y `NULL`. Un apuntador nulo no direcciona ningún dato válido. Un apuntador `void` direcciona datos de un tipo no especificado. Un apuntador `void` se puede igualar a nulo si no se direcciona ningún dato válido. `NULL` es un valor; `void` es un tipo de dato.

## 11.4 Apuntadores a apuntadores

Un apuntador puede dirigirse a otra variable apuntador. Este concepto se utiliza con mucha frecuencia en programas complejos de C. Para declarar un apuntador a un apuntador se hace preceder a la variable con dos asteriscos (\*\*).

En el siguiente ejemplo, `ptr5` es un apuntador a un apuntador.

```
int valor_e = 100;
int *ptr1 = &valor_e;
int **ptr5 = &ptr1;
```

`ptr1` y `ptr5` son apuntadores. `ptr1` apunta a la variable `valor_e` de tipo `int`. `ptr5` contiene la dirección de `ptr1`. En la figura 11.4 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 */
```

Declarar y utilizar diversas variables apuntador.

### Ejemplo 11.4

```
char c = 'z';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
****pppc = 'm'; /* cambia el valor de c a 'm' */
```

La representación en memoria de estas variables es:

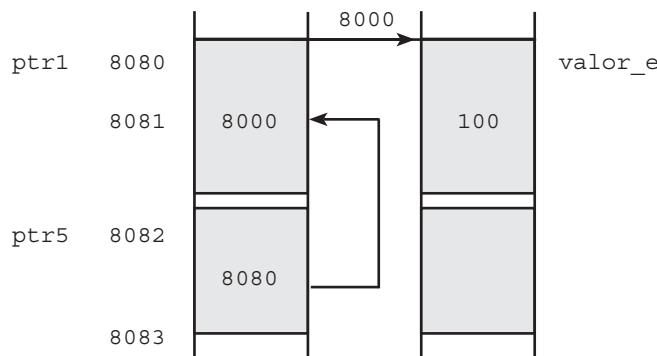
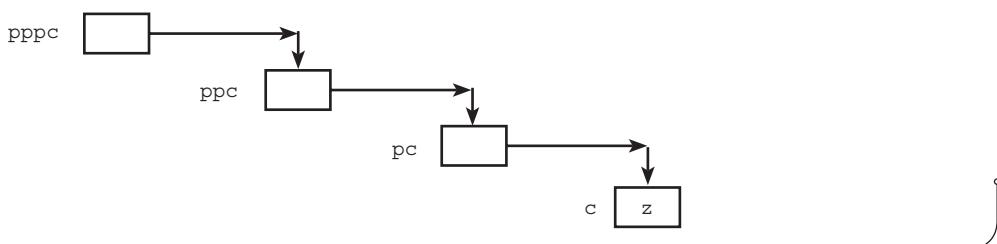


Figura 11.4 Un apuntador a un apuntador.

## 11.5 Apuntadores y arreglos

Los arreglos y apuntadores están fuertemente relacionados en el lenguaje C. Se pueden direccionar arreglos como si fueran apuntadores y apuntadores como si fueran arreglos. La posibilidad de almacenar y acceder a apuntadores y arreglos, hace posible que se puedan almacenar cadenas en elementos de un arreglo. Sin apuntadores 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.

### Nombres de arreglos como apuntadores

Un nombre de un arreglo es simplemente un apuntador. Supóngase la siguiente declaración de un arreglo:

```
int lista[5] = {10, 20, 30, 40, 50};
```

Si se manda visualizar `lista[0]` se verá 10. Pero, ¿qué sucederá si se manda visualizar `*lista`? Como un nombre de un arreglo es un apuntador, también se verá 10. Esto significa que:

|           |          |          |
|-----------|----------|----------|
| lista + 0 | apunta a | lista[0] |
| lista + 1 | apunta a | lista[1] |
| lista + 2 | apunta a | lista[2] |
| lista + 3 | apunta a | lista[3] |
| lista + 4 | apunta a | lista[4] |

Por consiguiente, para imprimir (visualizar), almacenar o calcular un elemento de un arreglo, se puede utilizar notación de subíndices o notación de apuntadores. Dado que un nombre de un arreglo contiene la dirección del primer elemento del arreglo, se debe indireccionar el apuntador para obtener el valor del elemento.

El nombre de un arreglo es un apuntador, contiene la dirección en memoria de comienzo de la secuencia de elementos que forma el arreglo. Es un apuntador constante ya que no se puede modificar, solo se puede acceder para indexar a los elementos del arreglo. En el siguiente ejemplo se ponen de manifiesto operaciones correctas y erróneas con nombres de arreglo.

```
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 */
```

### Ventajas de los apuntadores

Un nombre de un arreglo es una *constante apuntador*, no una variable apuntador. No se puede cambiar el valor de un nombre de arreglo, como no se pueden cambiar constantes. Esto explica por qué

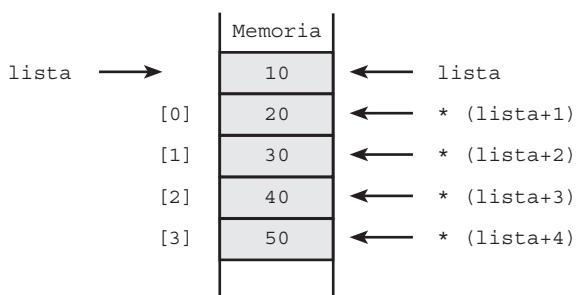


Figura 11.5 Un arreglo almacenado en memoria.

no se pueden asignar valores nuevos a un arreglo durante una ejecución de un programa. Por ejemplo, si `cnombre` es un arreglo de caracteres, la siguiente sentencia no es válida en C:

```
cnombre = "Hermanos Daltón";
```

Se pueden asignar valores al nombre de un arreglo solo en el momento de la declaración, o bien utilizando funciones, como (ya se ha hecho anteriormente) `strcpy()`.

Se pueden cambiar apuntadores para hacerlos apuntar a valores diferentes en memoria. El siguiente programa muestra cómo cambiar apuntadores. El programa define dos valores de coma flotante. Un apuntador de coma flotante apunta a la primera variable `v1` y se utiliza en `printf()`. A continuación el apuntador se cambia, 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 apuntadores, la mayoría de los programadores de C utilizan apuntadores en lugar de arreglos. Como los arreglos son fáciles de declarar, los programadores declaran arreglos y a continuación utilizan apuntadores para referenciar a los elementos de dichos arreglos.

## 11.6 Arreglos (arrays) de apuntadores

Si se necesita reservar muchos apuntadores a muchos valores diferentes, se puede declarar un arreglo de apuntadores. Un arreglo de apuntadores es un arreglo que contiene como elementos apuntadores, cada uno de los cuales apunta a un tipo de dato específico. La línea siguiente reserva un arreglo de diez variables apuntador a enteros:

```
int *ptr[10]; /* reserva un array de 10 apuntadores a enteros */
```

La figura 11.6 muestra cómo C organiza este arreglo. 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, como para variables apuntador no arreglos. 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 arreglos de apuntadores, en este caso de caracteres es:

```
char *puntos[25]; /* array de 25 apuntadores a carácter */
```

De igual forma, se podría declarar un apuntador a un arreglo de enteros.

```
int (*ptr10) [];
```

y las operaciones de dicha declaración paso a paso son:

```
(*ptr10) es un puntero, ptr10 es un nombre de variable.
(*ptr10) [] es un puntero a un array
int (*ptr10) [] es un puntero a un array de int
```

Una matriz de número enteros, o reales, puede verse como un arreglo de apuntadores; de tantos elementos como filas tenga la matriz, apuntando cada elemento del arreglo a un arreglo de enteros o reales, de tantos elementos como columnas.

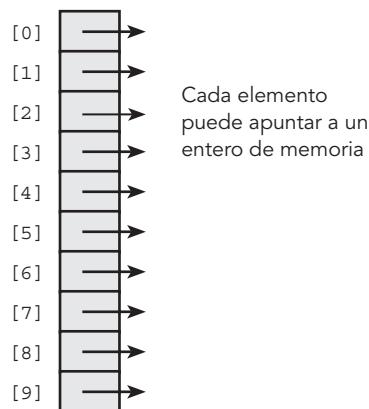


Figura 11.6 Un arreglo de 10 apuntadores a enteros.

## Inicialización de un arreglo de apuntadores a cadenas

La inicialización de un arreglo de apuntadores a cadenas se puede realizar con una declaración similar a esta:

```
char *nombres_meses[12] = { "Enero", "Febrero", "Marzo",
 "Abril", "Mayo", "Junio",
 "Julio", "Agosto", "Septiembre",
 "Octubre", "Noviembre",
 "Diciembre" };
```

## 11.7 Apuntadores a cadenas

Los apuntadores se pueden utilizar en lugar de índices de arreglos. Considérese la siguiente declaración de un arreglo de caracteres que contiene las veintiséis letras del alfabeto internacional (no se considera la ñ).

```
char alfabeto[27] = "ABCDEFGHIJKLMNPQRSTUVWXYZ" ;
```

Declaremos *p* un apuntador 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 decimosexto (la letra Q). Sin embargo, no se puede hacer.

```
p = &alfabeto;
```

ya que *alfabeto* es un arreglo cuyos elementos son de tipo *char*, y se produciría un error al compilar (tipo de asignación es incompatible).

Es posible, entonces, considerar dos tipos de definiciones de cadena:

```
char cadena1[] = "Hola viejo mundo"; /* array contiene una cadena */
char *cptr = "C a su alcance"; /* puntero a cadena, el sistema
 reserva memoria para la cadena */
```

### Apuntadores versus arreglos

Se escriben dos programas que definen una función para contar el número de caracteres de una cadena. En el primer programa, la cadena se describe utilizando un arreglo, y en el segundo, se describe utilizando un apuntador.

```
/* Implementación con un array */
#include <stdio.h>
int longitud(const char cad[]);
void main()
```

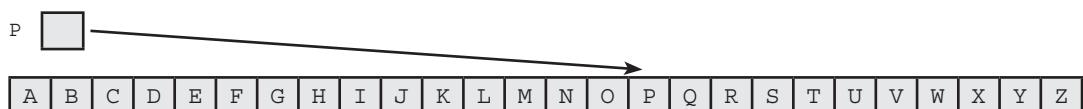


Figura 11.7 Un apuntador a alfabeto[15]

```

{
 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 apuntador para la función que cuenta los caracteres de la cadena. Además, utiliza la aritmética de apuntadores 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

### A recordar

#### Comparaciones entre apuntadores y arreglos de apuntadores

|                   |                                             |
|-------------------|---------------------------------------------|
| int *ptr1[];      | Arreglo de apuntadores a int                |
| int (*ptr2) [];   | Apuntador a un arreglo de elementos int     |
| int * (*ptr3) []; | Apuntador a un arreglo de apuntadores a int |

## 11.8 Aritmética de apuntadores

Al contrario que un nombre de arreglo, que es un apuntador constante y no se puede modificar, un apuntador es una variable que se puede modificar. Como consecuencia, se pueden realizar ciertas operaciones aritméticas sobre apuntadores.

A un apuntador se le puede sumar o restar un entero  $n$ ; esto hace que apunte  $n$  posiciones adelante, o atrás de la actual. Una variable apuntador puede modificarse para que contenga una dirección de memoria  $n$  posiciones adelante o atrás. Observe las siguientes sentencias:

```

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 apuntador 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 */

```

Recuerde que un apuntador es una dirección; por consiguiente, solo aquellas operaciones de “sentido común” son legales. No tiene sentido, por ejemplo, sumar o restar una constante de coma flotante a un apuntador.

### A recordar

#### Operaciones no válidas con apuntadores

- No se pueden sumar dos apuntadores.
- No se pueden multiplicar dos apuntadores.
- No se pueden dividir dos apuntadores



### Ejemplo 11.5

Acceso a elementos mediante un apuntador.

Suponga que `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 bucle `for` del ejemplo anterior se puede abreviar, haciendo uso de la característica de *terminador nulo* al final de la cadena utilizando la sentencia `while` para realizar el bucle. La condición de terminación del bucle es alcanzar el 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.

### Una aplicación de apuntadores: conversión de caracteres

El siguiente programa muestra un apuntador que recorre una cadena de caracteres y convierte cualquier carácter en minúsculas a caracteres mayúsculas.

```

#include <stdio.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);
}

```

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: restar 32 del código ASCII de una letra minúscula convierte a esa letra en mayúscula.

## 11.9 Apuntadores constantes frente a apuntadores a constantes

Ya está familiarizado con apuntadores constantes, como es el caso de un nombre de un arreglo. Un apuntador constante es un apuntador que no se puede cambiar, pero que los datos apuntados por el apuntador pueden ser cambiados. Por otra parte, un apuntador a una constante se puede modificar para apuntar a una constante diferente, pero los datos apuntados por el apuntador no se pueden cambiar.

### Apuntadores constantes

Para crear un apuntador constante diferente de un nombre de un arreglo, se debe utilizar el siguiente formato:

`<tipo de dato> *const <nombre puntero> = <dirección de variable>;`

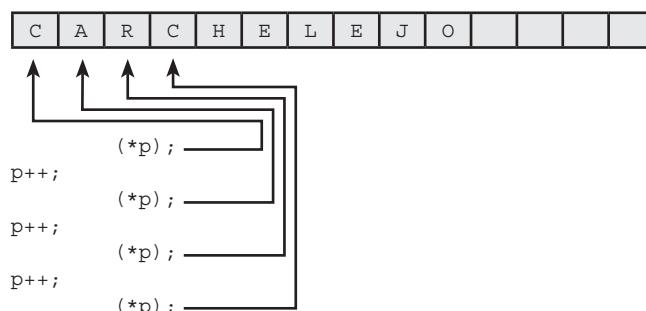


Figura 11.8 `*p++` se utiliza para acceder de modo incremental en la cadena.

Como ejemplo de una definición de apuntadores de constantes, considérense las siguientes sentencias:

```
int x;
int y;
int *const p1 = &x;
```

p1 es un apuntador 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 apuntador en sí.

```
*p1 = y;
```

Por otra parte, la siguiente asignación no es legal, ya que se intenta cambiar el valor del apuntador

```
p1 = &y;
```

El sistema para crear un apuntador 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 apuntador.

## Apuntadores a constantes

El formato para definir un apuntador 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 apuntador a la constante x. Los datos son constantes y no el apuntador; 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;
```

### A recordar

Una definición de un apuntador constante tiene la palabra reservada `const` delante del nombre del apuntador, mientras que el apuntador 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 “apuntador constante o de constante”, mientras que en el segundo caso la definición se lee “apuntador a tipo constante de datos”.

La creación de un *apuntador a una constante cadena* se puede hacer del modo siguiente:

```
const char *apellido = "Ramirez";
```

En el prototipo de la siguiente función se declara el argumento como apuntador a una constante:

```
float cargo(const float *v);
```

## Apuntadores constantes a constantes

El último caso a considerar es crear apuntadores 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 apuntador constante”. Un ejemplo puede ser:

```
const int x = 25;
const int *const p1 = &x;
```

que indica: “*p1 es un apuntador constante que apunta a la constante entera x*”. Cualquier intento de modificar *p1* o bien *\*p1* producirá un error de compilación.

### Consejo de programación

- Si sabe que un apuntador siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defínalo como un apuntador constante.
- Si sabe que el dato apuntado por el apuntador nunca necesitará cambiar, defina el apuntador como un apuntador a una constante.

Un apuntador a una constante es distinto de un apuntador constante. El siguiente ejemplo muestra las diferencias.

### Ejemplo 11.6

Este trozo de código define cuatro variables: un apuntador *p*; un apuntador constante *cp*; un apuntador *pc* a una constante y un apuntador constante *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 de programación

El espacio en blanco no es significativo en la declaración de apuntadores. Las declaraciones siguientes son equivalentes.

```
int* p;
int * p;
int *p;
```

## 11.10 Apuntadores como argumentos de funciones

Con frecuencia se desea que una función calcule y devuelva más de un valor, o bien se quiere 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 apuntador 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 apuntador a una variable local como argumento y cambiar la variable en la otra función.

Consideré 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 arreglo a una función es lo mismo que pasar un apuntador al arreglo. Se pueden cambiar cualquiera de los elementos del arreglo. 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.

### A recordar

En C, de manera predeterminada, 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 apuntadores 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 apuntador.

```
void registrotemp(struct temperatura *t)
{
 float actual;
 leerempactual(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;
registrotemp(&tmp);
```

## 11.11 Apuntadores a funciones

Hasta este momento se han analizado apuntadores a datos. Es posible declarar apuntadores a cualquier tipo de variables, estructura o arreglo. De igual modo, las funciones pueden declarar parámetros apuntadores para permitir que sentencias pasen las direcciones de los argumentos a esas funciones.

Es posible también crear apuntadores que apunten a funciones. En lugar de direccionar datos, los apuntadores 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 apuntadores. Tales funciones se pueden llamar de un modo indirecto, es decir, mediante un apuntador cuyo valor es igual a la dirección inicial de la función en cuestión.

La sintaxis general para la declaración de un apuntador a una función es:

```
Tipo_de_retorno (*PunteroFuncion) (<lista de parámetros>);
```

Este formato indica al compilador que `PunteroFuncion` es un apuntador a una función que devuelve el tipo `Tipo_de_retorno` y tiene una lista de parámetros.

### A recordar

Un apuntador a una función es simplemente un apuntador cuyo valor es la dirección del nombre de la función. Dado que el nombre es, en sí mismo, un apuntador; un apuntador a una función es un apuntador a un apuntador constante.

Por ejemplo:

```
int f(int);
int (*pf)(int);
/* declara la función f */
/* define puntero pf a función int con argumento
 int */
/* asigna la dirección de f a pf */
```

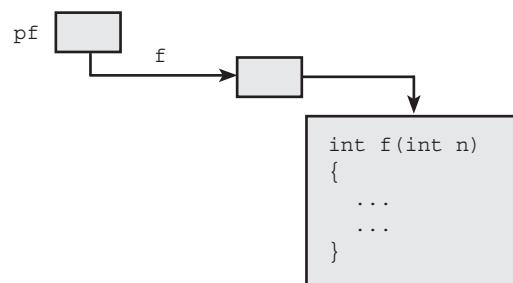


Figura 11.9 Un apuntador a función.



### Ejemplo 11.7

Se declaran apuntadores a funciones.

```
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 apuntador, `p`, apunta a una función que devuelve un tipo `float` y acepta dos parámetros de tipo `int`. El tercer apuntador, `sort`, es un apuntador a una función que devuelve un tipo `void` y toma dos parámetros: un apuntador a `int` y un tipo `unsigned`. Por último, `search` es un apuntador a una función que devuelve un tipo `unsigned` y tiene tres parámetros: un `int`, un apuntador a `int` y un `unsigned`.

## Inicialización de un apuntador a una función

La sintaxis general para inicializar un apuntador a una función es:

```
PunteroFuncion = unaFuncion
```

La función asignada debe tener el mismo tipo de retorno y lista de parámetros que el apuntador a función; en caso contrario, se producirá un error de compilación. Así, por ejemplo, un apuntador `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, como `qsort()`, requiere pasar un argumento que consta de un apuntador a una función. En el ejemplo 11.8 se llama a la función `qsort` con un apuntador de función.



### Ejemplo 11.8

Se desea ordenar un arreglo de números reales; la ordenación se va a realizar con la función `qsort()`.

Esta función tiene un parámetro que es un apuntador a función del tipo `int (*) (const void*, const void*)`. Se necesita una función de comparación, que devuelva negativo si el 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 = 8; /* número de elementos */
 printf ("\n Número 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);
}
}

```

Llamada a una función con apuntador a función.

Supongamos un apuntador *p* a una función 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 apuntador *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 apuntadores* 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 muchas veces. Mediante un apuntador, la elección solo se hace una vez: después de asignar (*la dirección de*) la función seleccionada a un apuntador y a continuación se puede llamar a través de ese apuntador.

Los *apuntadores 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 citadas:

```

void sufunc(int (*f)()); /* prototipo de sufunc */
int mifunc(int i); /* prototipo de mifunc */
void main()
{
 ...
 sufunc(mifunc);
}

```

### Ejemplo 11.9



```
int mifunc(int i)
{
 return 5*i;
}
```

En la función llamada se declara la función pasada como un apuntador función.

```
void sufunc(int (*f) ())
{
 ...
 j = f(5);
 ...
}
```

Como ejemplo se escribe una función genérica que calcule la suma de los resultados de la función *f*, es decir,

```
f(1) + f(2) + ... + f(n)
```

para cualquier función *f* que devuelva el tipo *double* y tenga un argumento *int*. Se diseña 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 inversos y de cuadrados:

```
inversos(k) = 1.0/k {para k = 1, 2, 3, 4, 5}
cuadrados(k) = k2 {para k = 1, 2, 3}
```

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)  $1 + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.0}{4} + \frac{1.0}{5}$

b) 1.0 + 4.0 + 9.0

y su salida será:

```
Suma de cinco inversos: 2.283
Suma de tres cuadrados: 14.000
```

## 11.12 Apunadores a estructuras

Un apuntador también puede apuntar a una estructura. Se declara una variable apuntador a estructura de igual forma que se declara un apuntador a cualquier otro tipo de dato. La declaración es similar a la de una variable estructura, pero anteponiendo `*` a la variable: `struct identificador *variable`.

```
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 utiliza el operador `->` para acceder a un miembro de una estructura referenciada mediante un apuntador a estructura.

En este ejemplo se declara el tipo estructura `t_persona`, que se asocia con el tipo `persona` para facilidad de escritura. Un arreglo de esta estructura se inicializa con campos al azar y se muestran por pantalla.

### Ejemplo 11.10

```
#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[] = { {"Mortimer, 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);
}

```

## 11.13 Asignación dinámica de la memoria

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 arreglo necesario, a menos que se reserve un arreglo de gran dimensión y se malgaste memoria cuando no se utilice. El método para resolver este inconveniente es recurrir a apuntadores y a técnicas de *asignación dinámica de la 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 la 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, no 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, 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.

### Precaución

#### Error típico de programación en C

La declaración de un arreglo exige especificar su longitud como una expresión constante, así `str` declara un arreglo de 100 elementos:

```
char str[100];
```

Si se utiliza una variable en la expresión que determina la longitud de un arreglo, se producirá un error.

```
int n;
... scanf("%d", &n);
char str[n]; /* error */
```

## Almacén libre (free store)

El mapa de memoria del modelo de un programa grande es muy similar al mostrado en la figura 11.10. 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.

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).

## 11.14 Función de asignación de memoria `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,

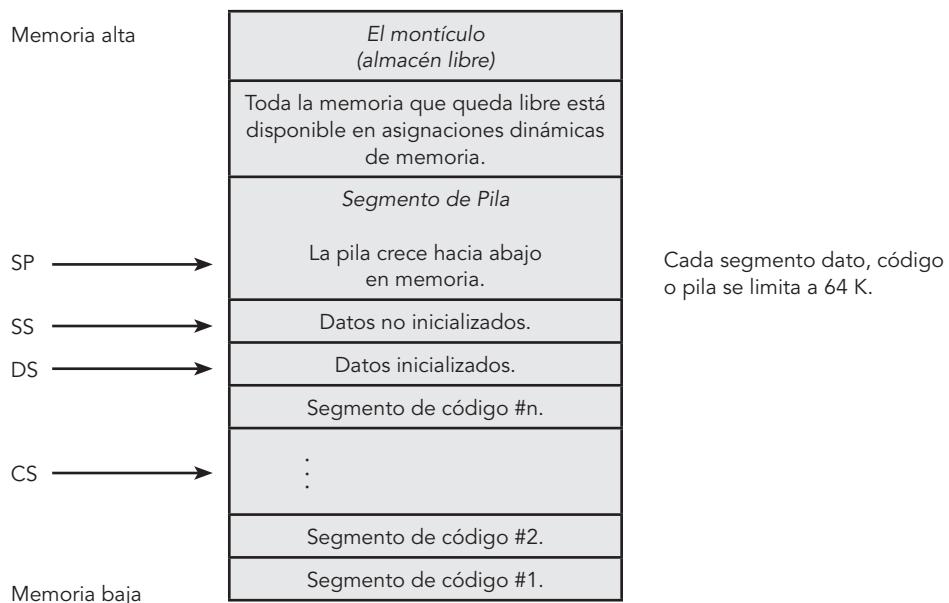


Figura 11.10 Mapa de memoria de un programa.

`malloc()` devuelve un apuntador, que es la dirección del bloque asignado de memoria. El apuntador se utiliza para referenciar el bloque de memoria; devuelve un apuntador 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 apuntador:

```
tipo *puntero;
puntero = (tipo *)malloc(tamaño en bytes);
```

Por ejemplo:

```
long* p;
p = (long*) malloc(32);
```

El operador `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`.

### A recordar

#### 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);
```

En la sintaxis de llamada, *apuntador* es el nombre de la variable apuntador 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 arreglo 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));
```

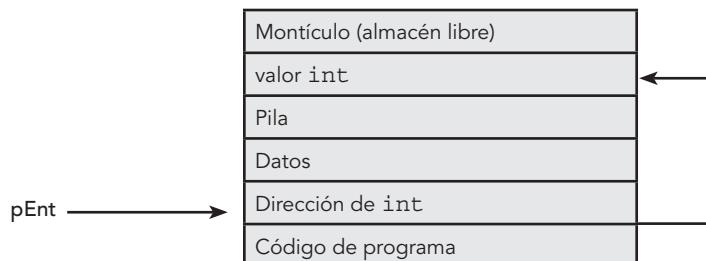
La llamada a *malloc( )* asigna espacio para un *int* (entero) y almacena la dirección de la asignación en *pEnt*. *pEnt* apunta ahora a la posición en el almacén libre (montículo) donde se establece la memoria. La figura 11.11 muestra cómo *pEnt* apunta a la asignación del almacén libre. Así, por ejemplo, para reservar memoria para un arreglo de 100 números reales:

```
float *BloqueMem;
BloqueMem = (float*) malloc(100*sizeof(float));
```

En el ejemplo se declara un apuntador denominado *BloqueMem* y se inicializa a la dirección devuelta por *malloc( )*. Si un bloque del tamaño solicitado está disponible, *malloc( )* devuelve un apuntador 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 escribir así:

```
int n;
char *s;
...
s = (char*) malloc(n*sizeof(char));
```

La función *malloc( )* está declarada en el archivo de cabecera *stdlib.h*.



**Figura 11.11** Despues de *malloc( )*, con el tamaño de un entero, *pEnt* apunta a la posición del montículo donde se ha asignado espacio para el entero.



### Ejemplo 11.11

Leer una línea de caracteres, reservar memoria para un *buffer* de tantos caracteres como los caracteres leídos y copiar en el *buffer* la cadena.

```
#include <stdio.h>
#include <string.h> /* por el uso de strcpy() */
#include <stdlib.h>

void main()
{
 char cad[121], *ptr;
 int lon;

 puts("\nIntroducir 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 apuntador que apunta a un bloque de memoria capaz de contener la cadena de longitud `strlen()` más un byte extra por el carácter '\0' al final de la cadena.

### Precaución

El almacenamiento libre no es una fuente inagotable de memoria. Si la función `malloc()` se ejecuta con falta de memoria, devuelve un apuntador `NULL`. Es responsabilidad del programador comprobar *siempre* el apuntador para asegurar que es válido, antes de que se asigne un valor al apuntador. Supongamos, por ejemplo, que se desea asignar un arreglo 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);

```

### A recordar

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 el archivo de cabecera `<stdlib.h>` para obtener la definición de `NULL`.

Este programa 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.

```

/*
programa para determinar memoria libre.
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
 void *p;
 int i;

```

### Ejemplo 11.12



```

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 (kilobyte) hasta que falla la asignación de memoria y el bucle termina.

## Asignación de memoria de un tamaño desconocido

Se puede invocar a la función `malloc()` para obtener memoria para un arreglo, incluso si no se conoce con antelación cuánta memoria requieren los elementos del arreglo. Todo lo que se ha de hacer es invocar a `malloc()` en tiempo de ejecución, pasando como argumento el número de elementos del arreglo multiplicado por el tamaño del tipo del arreglo. 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 arreglo 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("Cuantos 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);
}

```

## Uso de `malloc()` para arreglos multidimensionales

Un arreglo bidimensional es, en realidad, un arreglo cuyos elementos son arreglos. Al ser el nombre de un arreglo unidimensional un apuntador constante, un arreglo bidimensional será un apuntador a apuntador constante (tipo `**`). Para asignar memoria a un arreglo multidimensional, se indica cada dimensión del arreglo de igual forma que se declara un arreglo unidimensional. En el ejemplo 11.13 se reserva memoria en tiempo de ejecución para una matriz de `n` filas y `m` elementos en cada fila.

El siguiente programa crea una matriz de  $n \times m$  y lee los elementos del dispositivo de entrada (teclado).

### Ejemplo 11.13

```
/* 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 Número 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 0;
}
```

En el ejemplo, la sentencia `p = (int**) malloc(n*sizeof(int*));` reserva memoria para un arreglo de  $n$  elementos, cada elemento es un apuntador 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.

## 11.15 La 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

```
freeadc();
```

**Sintaxis de llamada a `free()`**

```
tipo *apuntador;
...
free(apuntador);
```

La variable `apuntador` puede apuntar a una dirección de memoria de cualquier tipo.

**Prototipo que incluye `free()`**

```
void free(void *);
```

**Ejemplo 11.14**

En este ejemplo se reserva memoria para un arreglo 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 */
}
```

## 11.16 Funciones de asignación de memoria `calloc()` y `realloc()`

Además de la función `malloc()` para la obtención de bloques de memoria, existen otras dos funciones que permiten obtener memoria libre en tiempo de ejecución: `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 apuntador al bloque asignado de memoria. El apuntador se utiliza para referenciar el bloque de memoria. El apuntador que devuelven es del tipo `void*`.

**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 apuntador:

```
tipo *puntero;
puntero = (tipo*)calloc(numero_elementos, tamaño_de_cada_elemento);
```

El tamaño de cada elemento se expresa en bytes; se utiliza para su obtención el operador `sizeof`. Por ejemplo, si se desea 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 *apuntador;
int numelementos;
. . .
apuntador = (tipo*)calloc(numelementos,tamaño de tipo);
```

La función devuelve la dirección de la variable asignada dinámicamente, el tipo de dato que devuelve es `void*`.

### Prototipo de `calloc()`

```
void* calloc(size_t n, size_t t);
```

En la sintaxis de llamada, *apuntador* es el nombre de la variable apuntador 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`, arreglos, estructuras e identificadores de `typedef`.

En el siguiente programa se considera una secuencia de números reales, con una variable apuntador a `float` se procesa un arreglo 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("\n Introduce %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;
}
```

### Función realloc( )

Esta función también sirve para asignar un bloque de memoria libre. Presenta una variante respecto a `malloc()` y `calloc()`, pues 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 apuntador:

```
tipo *puntero;
puntero = (tipo*)realloc(puntero a bloque, tamaño total de nuevo bloque);
```

El tamaño del bloque se expresa en bytes. El apuntador a bloque referencia a un bloque de memoria reservado previamente con `malloc()`, `calloc()` o la propia `realloc()`.



#### Ejemplo 11.15 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);
 /* Amplía el bloque de memoria */
 tam += (strlen(" en Lupiana\n") + 1) * sizeof(char);
 cadena = (char*)realloc(cadena, tam);
 strncat(cadena, " en Lupiana\n");
 puts(cadena);
 /* Se libera 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 apuntador 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 apuntador 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 11.16 Leer 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)
 {
 /* amplía 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);
```

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 apuntador transmitida como primer argumento. En cualquier caso, `realloc()` copia los datos referenciados por apuntador 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`.

### 11.17 Reglas de funcionamiento de la asignación dinámica

Como ya se ha comentado se puede asignar espacio para cualquier tipo 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 está 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 apuntador.

```
#include <stdlib.h>
void main()
{
 double* vec;
 int n;
 ...
 vec = (double*)calloc(n,sizeof(double));
}
```

3. Las funciones de asignación de memoria 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 se 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 para datos más complejos, como estructuras, arreglos, 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 arreglos multidimensionales de objetos con las funciones de asignación de memoria. Para un arreglo bidimensional  $n \times m$ , se asigna en primer lugar memoria para un arreglo de apun-  
dores (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]);
}
```

## Resumen

Los apun-  
dores son una de las herramientas más eficien-  
tes para realizar aplicaciones en C. Aunque su práctica  
puede resultar difícil y tediosa su aprendizaje es, sin lugar  
a dudas, una necesidad vital si desea obtener el máximo  
rendimiento de sus programas.

En este capítulo habrá aprendido los siguientes con-  
ceptos:

- Un apun-  
dor es una variable que contiene la direc-  
ción de una posición en memoria.
- Para declarar un apun-  
dor 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 uti-  
lizada por el apun-  
dor, se utiliza el operador de indi-  
rección `(*)`. El valor de `p` es una de memoria y el

valor de `*p` es el dato entero almacenado en esa direc-  
ción de memoria.

- Para obtener la dirección de una variable existente, se  
utiliza el operador de dirección `(&)`.
- Se debe declarar un apun-  
dor antes de su uso.
- Un apun-  
dor `void` es un apun-  
dor 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 apun-  
dor que no apunta a nada,  
se utiliza la constante `NULL`.
- Estableciendo un apun-  
dor a la dirección del primer  
elemento de un arreglo, se puede utilizar el apun-  
dor para acceder a cada elemento del arreglo de modo  
secuencial.

Asimismo, se han estudiado los conceptos de aritmética de apuntadores, apuntadores a funciones, apuntadores a estructuras y arreglos de apuntadores.

También estudiamos en este capítulo la asignación dinámica de memoria, la cual 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 deseé.

En C se utilizan las funciones `malloc()`, `calloc()`, `realloc()` y `free()` para asignar y liberar memoria. Las funciones `malloc()`, `calloc()` y `realloc()` permiten asignar memoria para cualquier tipo de dato especificado (un `int`, un `float`, una estructura, un arreglo 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.

## Ejercicios

11.1 Encontrar los errores en la siguiente declaración de apuntadores:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= 'C';
float x;
void* r = &x;
```

11.2 Dada la siguiente declaración, escribir una función que tenga como argumento un apuntador al tipo de dato y muestre por pantalla los campos.

```
struct boton
{
 char* rotulo;
 int codigo;
};
```

11.3 ¿Qué diferencias se pueden encontrar entre un apuntador a constante y una constante apuntador?

11.4 Un arreglo unidimensional se puede indexar con la aritmética de apuntadores. ¿Qué tipo de apuntador habría que definir para indexar un arreglo bidimensional?

11.5 En el siguiente código se accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de apuntadores.

```
#define N 4
#define M 5
int f,c;
```

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
{
 .
 .
 .
 free (pperro); /* libera espacio
apuntado por pperro */
}
```

```
double mt [N] [M];
.
.
for (f = 0; f < N; f++)
{
 for (c = 0; c < M; c++)
 printf("%lf ", mt[f] [c]);
 printf("\n");
}
```

11.6 Escribir una función con un argumento de tipo apuntador a `double` y otro argumento de tipo `int`. El primer argumento se debe de corresponder con un arreglo y el segundo con el número de elementos del arreglo. La función ha de ser de tipo apuntador a `double` para devolver la dirección del elemento menor.

11.7 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);`

- 11.8 ¿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];
```

- 11.9 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;
```

¿Qué problemas habría en la siguiente sentencia?

```
gets(t.mes);
```

- 11.10 El prototipo de una función es: `void escribe_mat (int**t, int nf, int nc);`  
 Escribir un fragmento de código que llame correctamente a la función.

- 11.11 Encontrar los errores en las siguientes declaraciones y sentencias.

```
int n, *p;
char** dob= "Cadena de dos punteros";
p = n*malloc(sizeof(int));
```

- 11.12 Dada la siguiente declaración, definir un apuntador `b` a la estructura, reservar memoria dinámicamente para una estructura asignando su dirección a `b`.

```
struct boton
{
 char* rotulo;
 int codigo;
};
```

- 11.13 Un arreglo unidimensional puede considerarse una constante apuntador. ¿Cómo puede considerarse un arreglo bidimensional?, ¿y un arreglo de tres dimensiones?

- 11.14 Declarar una estructura para representar un punto en el espacio tridimensional. Declarar un apuntador a la estructura para que tenga la dirección de un arreglo dinámico de `n` estructuras punto. Utilizar la función `calloc()` para asignar memoria al arreglo y comprobar que se ha podido asignar la memoria requerida.

- 11.15 Dada la declaración del arreglo de apuntadores:

```
#define N 4
char * [N];
```

Escribir las sentencias de código para leer `n` líneas de caracteres y asignar cada línea a un elemento del arreglo.

## Problemas

En todos los problemas utilizar siempre que sea posible apuntadores para acceder a los elementos de los arreglos, tanto numéricos como cadenas de caracteres.

- 11.1 Escribir un programa en el que se lean 20 líneas de texto, cada línea con un máximo de 80 caracteres. Mostrar por pantalla el número de vocales que tiene cada línea.

- 11.2 Escribir 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 apuntadores en la primera función, en la segunda, indexación.

- 11.3 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. Escribir 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.

- 11.4 Se quiere evaluar las funciones `f(x)`, `g(x)` y `z(x)` para todos los valores de `x` en el intervalo `0 ≤ x <`

3.5 con incremento de 0.2. Escribir un programa que evalúe dichas funciones, utilizar un arreglo de apuntadores a función. Las funciones son las siguientes:

$$\begin{aligned}f(x) &= 3 \cdot e^{x-1} - 2x \\g(x) &= -x \cdot \sin(x) + 1.5 \\z(x) &= x^2 - 2x + 3\end{aligned}$$

- 11.5 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 cadenas de caracteres para guardar cada entero y realizar la suma. Escribir un programa que lea dos enteros largos y realice la suma.
- 11.6 Escribir 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).
- 11.7 Escribir 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 descomponerse en subproblemas y utilizar apuntadores para acceder a los elementos de la matriz.
- 11.8 Desarrollar 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 arreglo y mostrar los pacientes cuyo día de visita sea uno determinado.
- 11.9 Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapezo y un círculo. Utilizar un arreglo de apuntadores de funciones; estas últimas son las que permiten calcular el área.
- 11.10 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 apuntador de función aplicar uno de estos métodos para encontrar una raíz de dicha ecuación.

- 11.11 Escribir 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 leída las cadenas se debe 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.
- 11.12 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 competición es por eliminación. Hay dos tipos de pruebas, 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 han intervenido en menos pruebas. El juego termina cuando no quedan ciclistas para alguna de las dos pruebas. Se han de mantener arreglos dinámicos con los ciclistas participantes y los eliminados. El ciclista ganador será el que más puntos tenga.
- 11.13 Se tiene una matriz de  $20 \times 20$  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 sostenidos solo los elementos no repetidos. Escribir un programa que tenga como entrada la matriz de  $20 \times 20$ ; genere la matriz dinámica perdida y que se muestre en pantalla.
- 11.14 Escribir 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.



# Entradas y salidas por archivos

## Contenido

- 12.1 Flujos
- 12.2 Apuntador (puntero) FILE
- 12.3 Apertura de un archivo
- 12.4 Funciones de entrada/salida para archivos
- 12.5 Archivos binarios en C

## 12.6 Funciones para acceso aleatorio

- 12.7 Datos externos al programa con argumentos de `main()`
- › Resumen
- › Ejercicios
- › Problemas

## 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** (*stream*) 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
- › Apertura de un archivo
- › Archivos binarios
- › Archivos secuenciales indexados
- › Cierre de un archivo
- › Flujos
- › Memoria intermedia
- › Memoria interna
- › Registro
- › Secuencias de caracteres

## 12.1 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 y permiten admitir *secuencias de caracteres*, en modo texto. Tienen el siguiente cometido:

|        |                                                                   |
|--------|-------------------------------------------------------------------|
| 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, y cuando se quiere 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 arreglo 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 una cadena del archivo, 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 archivos en C.

## 12.2 Apuntador (puntero) FILE

Los archivos se ubican en dispositivos externos como cintas, cartuchos, discos, discos compactos, etc., y tienen un nombre y unas características. En el programa el archivo tiene un nombre interno que es un apuntador a una estructura predefinida (*apuntador a archivo*). Esta estructura contiene información sobre el archivo, 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 está 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.

### A recordar

Al programador le interesa saber que existe el tipo `FILE` y que es necesario definir un apuntador 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.

Se declara un apuntador a FILE; se escribe el prototipo de una función de tipo apuntador a FILE y con un argumento del mismo tipo.

### Ejemplo 12.1

```
FILE* pf;
FILE* mostrar(FILE*); /* Prototipo de una función definida por el
programador*/
```

#### A recordar

La entrada estándar, al igual que la salida, están asociadas a variables apuntadores a FILE:

```
FILE *stdin, *stdout;
```

Al ejecutar todo programa C, automáticamente se abren los archivos de texto stdin (teclado) y stdout (pantalla).

## 12.3 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 apuntador a la estructura FILE, la función de apertura devuelve dicho apuntador. La función para abrir un archivo es fopen( ), 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 apuntador a FILE, a través de dicho apuntador 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 apuntador a FILE, para así después referirse a dicha variable.

Esta función puede detectar un error al abrir el archivo, por ejemplo que el archivo no exista y se quiera leer, entonces devuelve NULL.

Declarar una variable de tipo apuntador a FILE. A continuación escribir una sentencia de apertura de un archivo.

### Ejemplo 12.2

```
FILE* pf;
pf = fopen(nombre_archivo, modo);
```

Se desea abrir un archivo de nombre LICENCIA.EST para obtener ciertos datos.

### Ejemplo 12.3

```
#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 12.4**

Abrir 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: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` y es el siguiente:

```
FILE* fopen (const char* nombre_archivo, const char* modo);
```

**Modos de apertura de un archivo**

Al abrir el archivo, `fopen()` 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 la tabla que aparece más adelante.

Aparentemente en estos modos no se ha establecido el tipo del archivo, de texto o binario. Siempre en forma predeterminada, el archivo es de texto. Para no depender del entorno es mejor indicar si es de texto o binario. Se utiliza la letra `b` para modo binario como último carácter de la cadena `modo` (también se puede escribir como carácter intermedio). Algunos compiladores admiten la letra `t` para indicar archivo de texto. Por consiguiente, los modos de abrir un archivo de texto son:

```
"rt", "wt", "at", "r+t", "w+t", "a+t".
```

Y los modos de abrir un archivo binario son:

```
"rb", "wb", "ab", "r+b", "w+b", "a+b".
```

| Modo  | Significado                                                                                        |
|-------|----------------------------------------------------------------------------------------------------|
| "rt"  | Abre para lectura un archivo de texto, es igual que modo "r".                                      |
| "wt"  | Abre para crear nuevo archivo de texto (si ya existe se pierden sus datos), es igual que modo "w". |
| "a"   | Abre para añadir al final en un archivo de texto, igual que modo "at".                             |
| "r+t" | Abre archivo de texto ya existente para modificar (leer/escribir).                                 |
| "w+t" | Crea un archivo de texto para escribir/leer (si ya existe se pierden los datos).                   |
| "a+t" | Abre el archivo de texto para modificar (escribir/leer) al final. Si no existe es como w+.         |

**Ejemplo 12.5**

Se tiene el archivo de texto LICENCIA.EST, se quiere leer para realizar un cierto proceso y escribir datos obtenidos en el archivo binario 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);
}

```

### NULL y EOF

Las funciones de biblioteca que devuelven un apuntador (`strcpy( )`, `fopen( )`...) especifican que si no pueden realizar la operación (generalmente si hay un error) devuelven `NULL`. Esta clave es una macro definida en varios archivos de cabecera, entre los que se encuentran `stdio.h` y `stdlib.h`.

Las funciones de biblioteca 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`.

El siguiente segmento de código lee el flujo estándar de entrada hasta fin de archivo:

```

int c;
while ((c=getchar()) != EOF)
{
 ...
}

```

### Ejemplo 12.6



## Cierre de archivos

Los archivos en C trabajan con una *memoria intermedia, 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 este quedaría sin las últimas actualizaciones. Siempre que se termina de procesar un archivo y 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);
```

Abrir dos archivos de texto, después se cierra cada uno de ellos.

```

#include <stdio.h>
FILE *pf1, *pf2;

pf1 = fopen("C:\\DATOS.DAT", "at");
pf2 = fopen("C:\\TEMPS.RET", "wb");

fclose(pf1);
fclose(pf2);

```

### Ejemplo 12.7



## Volcado del buffer: `fflush( )`

El proceso de entrada/salida con archivos no se realiza accediendo directamente al dispositivo externo, sino a un *buffer (memoria interna)*. La operación de grabar escribe los datos en el *buffer*, en el momento en que este está lleno se vuelca al dispositivo externo. La operación de lectura vuelca datos del archivo al

*buffer* de tal forma que se lee de este, cuando se ha leído todo el *buffer* la misma operación de lectura vuelca nuevos datos desde el archivo al *buffer* para seguir leyendo de este.

La finalidad que tiene el *buffer* es, sencillamente, disminuir el tiempo de acceso a los archivos. La operación de *cierre de un archivo* (apartado anterior) vuelca el buffer en el archivo. Además, C dispone de la función `fflush()` para volcar y vaciar el *buffer* del archivo pasado como argumento.

```
FILE *fc
fc = fopen("mifichero", "w");
...
fflush(fc);
```

La entrada de datos por teclado está asociada al flujo (archivo) de entrada `stdin`. En ocasiones, principalmente cuando se mezcla entrada de cadenas y números, resulta útil vaciar el *buffer* de entrada llamando a `fflush(stdin)`. En el siguiente fragmento se realiza una entrada de un número entero, llamando a `scanf()`, y de una cadena de caracteres, llamando a `gets()`. La llamada `fflush(stdin)` hace que se vacíe íntegramente el *buffer* de entrada, en caso contrario quedaría el carácter fin de línea y `gets()` leería una cadena vacía.

```
int cuenta;
char b[81];
...
printf("Cantidad: ");
scanf("%d", &cuenta);
fflush(stdin);
printf("Dirección: ");
gets(b);
```

La función `fflush` devuelve 0 si no ha habido error, en caso de error devuelve la constante `EOF`. El prototipo se encuentra en la biblioteca `stdio.h` y es el siguiente:

```
int fflush(FILE* pf);
```

### A recordar

La llamada `fflush (NULL)` cierra todos los flujos (archivos) abiertos en el programa.

## 12.4 Funciones de entrada/salida para archivos

Una vez abierto un archivo para escribir datos hay que grabar los datos en el archivo. La biblioteca C proporciona diversas funciones a fin de escribir datos en el archivo a través del apuntador 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()` (las funciones específicas de archivos empiezan por `f`). Con estas funciones se escribe o lee cualquier dato del programa en un archivo de texto.

### Funciones `putc()` y `fputc()`

Ambas funciones son idénticas; `putc()` está definida como macro. Escriben un carácter en el archivo asociado con el apuntador 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);
```

donde `c` es el carácter a escribir.

11

## Ejercicio 12.1

Se desea crear un archivo SALIDA.PTA con los caracteres introducidos por teclado.

Una vez abierto el archivo, un bucle lee carácter a carácter mientras (while) no sea fin de archivo (macro EOF) y se escribe en el archivo asociado al apuntador 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 12.1 en vez de `putc(c,pf)` se puede utilizar `fputc(c,pf)`. El prototipo de ambas funciones se encuentra en `stdio.h` y es el siguiente:

```
int putc(int c, FILE* pf);
int fputc(int c, FILE* pf);
```

## Funciones `getc( )` y `fgetc( )`

Estas dos funciones son iguales, tienen el mismo formato y la misma funcionalidad; pueden considerarse que son recíprocas de `putc( )` y `fputc( )`; `getc( )` y `fgetc( )` leen un carácter (el siguiente carácter) del archivo asociado al apuntador a FILE y 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);
```

11

## Ejercicio 12.2

Se desea leer el archivo SALIDA.PTA, creado en el ejercicio 12.1 para mostrarlo por pantalla y contar las líneas que tiene.

Una vez abierto el archivo de texto en modo lectura, un bucle lee carácter a carácter mientras no sea *fin de archivo* (macro EOF) 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);

```

### 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, donde  $n$  es un argumento entero de la función. La función devuelve un apuntador a la cadena devuelta, o `NULL` si ha habido un error. El formato de llamada es:

```
fgets(cadena, n, puntero_archivo);
```



#### Ejemplo 12.8 Lectura de un máximo de 80 caracteres de un archivo:

```

#define T 81
char cad[T];
FILE *f;
fgets(cad, T, f);

```



#### Ejercicio 12.3

El archivo `CARTAS.DAT` contiene un texto al que se le quiere añadir nuevas líneas, de longitud mínima de 30 caracteres, desde el archivo `PRIMERO.DAT`.

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( )` y si cumplen la condición de longitud

se escriben en el archivo CARTAS. El proceso completo del archivo se lleva a cabo con un bucle while.

```
#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` y es el siguiente:

```
int fputs(char* cad, FILE* pf);
char* fgets(char* cad, int n, FILE* pf);
```

### Funciones `fprintf( )` y `fscanf( )`

Las funciones `printf( )` y `scanf( )` permiten escribir o leer variables de cualquier tipo de dato estándar, los códigos de formato (`%d`, `%f...`) indican a C la transformación que debe realizar con la secuencia de caracteres (conversión a entero...). La misma funcionalidad tienen `fprintf( )` y `fscanf( )` con los flujos (archivos asociados) a que se aplican. Estas dos funciones tienen como primer argumento el apuntador (puntero) a `FILE` asociado al archivo de texto.

#### Ejercicio 12.4

Crear el archivo de texto `PERSONAS.DAT` de tal forma que cada línea contenga un registro con los datos de una persona que tiene los campos nombre, fecha de nacimiento (día(nn), mes(nn), año(nnnn) y mes en ASCII).

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 <stdlib.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;
}

/* se leen los atributos de una persona por teclado */
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 se encuentra en `stdio.h` y es el siguiente:

```
int fprintf(FILE* pf,const char* formato, . . .);
int fscanf(FILE* pf,const char* formato, . . .);
```

### 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`).

El siguiente ejemplo transforma el bucle del ejercicio 12.2, 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");
 }
. . .
```

### Ejemplo 12.9

El prototipo de la función está en `stdio.h`, es el siguiente:

```
int feof(FILE* pf);
```

### 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 apuntador del archivo al inicio de este. El formato de llamada es el siguiente:

```
rewind(puntero_archivo);
```

El prototipo de la función se encuentra en `stdio.h`:

```
void rewind(FILE*pf);
```

Ler un archivo de texto, contar el número de líneas que tiene y a continuación situar el apuntador del archivo al inicio para una posterior lectura.

### Ejemplo 12.10

```
#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))
 {
 . . .
 }
 }
}

```

## 12.5 Archivos binarios en C

Para abrir un archivo en modo binario hay que especificar la opción `b` en la cadena *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 o 4 bytes (depende del sistema), y un número real de 4 u 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 ha de realizar en modo binario y solo 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 12.11** Declarar 3 apuntadores a FILE y a continuación abrir tres archivos en modo binario.

```

FILE *pf1, *pf2, *pf3;
pf1 = fopen("gorjal.arr", "rb"); /*Lectura de archivo binario */
pf2 = fopen ("tempes.feb", "w+b"); /*leer/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*: `fread( )` y `fwrite( )`.

### 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(direccion_buffer, tamaño, num_elementos, puntero_archivo);`



**Ejemplo 12.12** Abrir un archivo en modo binario para escribir y grabar números reales en doble precisión mediante un bucle `for`. El *buffer* es la variable `x` y el tamaño se devuelve con 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`: es un tipo `int`.

11

### Ejercicio 12.5

Se dispone de una muestra de una muestra de las coordenadas de puntos en el plano representada por pares de números enteros  $(x, y)$ , tales que  $1 \leq x \leq 100$  y  $1 \leq y \leq 100$ . Se desea guardar en un archivo binario todos los puntos disponibles.

El nombre del archivo que se crea es `PUNTOS.DAT`. Según se lee se comprueba la validez de un 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("Coordenadas 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;
}
```

### A recordar

Los archivos binarios están indicados especialmente para guardar estructuras en C. El método habitual consiste en escribir sucesivas estructuras en el archivo asociado al apuntador `FILE` de creación de archivos binarios.

### Función de lectura fread( )

Esta función lee de un archivo *n* bloques de bytes y lo 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 es:

```
 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 12.13

Abrir un archivo en modo binario para lectura. El archivo se lee hasta el fin del archivo y cada lectura es de un número real que se acumula en la variable *s*.

```
FILE *fd;
double x, s=0.0;
if((fd = fopen("reales.num", "rb"))== NULL)
 exit(-1);
while (!feof(fd))
{
 fread(&x, sizeof(double), 1, fd);
 s += x;
}
```

11

### Ejercicio 12.6

En el ejercicio 12.5 se ha creado un archivo binario de puntos en el plano. Se desea escribir un programa para determinar los siguientes valores:

- 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*.

$$\bar{i}_j = \frac{\sum_{i=1}^{100} n_{ij} * i}{\sum_{i=1}^{100} n_{ij}}$$

La primera instrucción es abrir el archivo binario para lectura. A continuación se solicita el punto del plano para el que 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 realizar el segundo apartado, se solicita el valor de *j*. Con un bucle desde *i=1* hasta 100 se cuentan las ocurrencias de cada punto (*i,j*) llamando a la función *cuenta\_pto()*; antes de cada llamada hay que situar el apuntador del archivo al inicio, llamando 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<= 100; 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;
}
/* Cuenta el número de veces del punto w */
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;
}

```

## 12.6 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. Es muy rápido acceder a la información que contienen. El 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. Estas funciones consideran el archivo como una secuencia de bytes; el número de byte es el índice del archivo. Según se van leyendo o escribiendo registros o datos en el archivo, el programa mantiene a través de un apuntador 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 apuntador al archivo a una dirección determinada.

La biblioteca de C también dispone de las funciones `fsetpos()` y `fgetpos()` para mover o conocer la posición actual del fichero.

### Función `fseek()`

Con la función `fseek()` se puede tratar un archivo en C como un arreglo, que es una estructura de datos de *acceso aleatorio*. `fseek()` sitúa el apuntador del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos. En el ejemplo 12.14 se supone que existe un archivo de productos, se solicita el número de producto y se sitúa el apuntador del archivo para leer el registro en una operación posterior.



#### Ejemplo 12.14

Declarar una estructura (registro) `PRODUCTO`, y abrir un archivo para lectura. Se desea leer un registro cuyo número (posición) se solicita 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 relativo del desplazamiento (0 indica que empieza a contar desde el principio del archivo). El formato para llamar a `fseek()`:

```
fseek(puntero_archivo, desplazamiento, origen);
```

`desplazamiento` : número de bytes a mover; tienen que ser de tipo *long*.

`origen` : 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);
```

### Función `ftell()`

La posición actual del archivo se puede obtener llamando a la función `ftell()` y pasando un apuntador al archivo como argumento. La función devuelve la posición dada en número de bytes (un entero largo: `long int`) desde el inicio del archivo (byte 0).

En este ejemplo se puede observar cómo se desplaza el apuntador del archivo según se escriben datos en él.

#### Ejemplo 12.15

```
#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 apuntador a `FILE`. El prototipo se encuentra en `stdio.h`:

```
long ftell(FILE *pf);
```

### Cambio de posición: `fgetpos()` y `fsetpos()`

Otra forma de conocer la *posición* actual del archivo, o bien mover dicha posición es mediante las funciones `fgetpos()` y `fsetpos()`.

La función `fgetpos()` tiene dos argumentos, el primero representa al archivo (flujo) mediante el apuntador `FILE` asociado. El segundo argumento de tipo apuntador a `fpos_t` (tipo entero declarado en `stdio.h`) es de *salida*; la función le asigna la posición actual del archivo. Por ejemplo:

```
FILE *pf;
fpos_t* posicion;
...
fgetpos(pf, posicion);
printf("Posición actual: %ld", (*posicion));
```

La función `fsetpos()` se utiliza para cambiar la posición actual del archivo. La nueva posición se pasa como segundo argumento (de tipo `const fpos_t*`) en la llamada a la función. El primer argumento es el apuntador `FILE` asociado al archivo. Por ejemplo, se lee a partir de una posición en un archivo, posteriormente se vuelve a la misma posición con el fin de completar un determinado proceso:

```
FILE *fc;
fpos_t* posicion;
registro reg;
fgetpos(fc, posicion);
while (condicion)
{
 fread(®, sizeof(registro), 1, fc)
 ...
}
fsetpos(fc, posicion); /* sitúa la posición actual del archivo
en la posición anterior al bucle */
```

Las dos funciones devuelven cero si no ha habido error en la ejecución, en caso contrario devuelven un valor distinto de cero (el número del error). Sus prototipos están en `stdio.h` y son los siguientes:

```
int fgetpos(FILE* pf, fpos_t* p);
int fsetpos(FILE* pf, const fpos_t* p);
```

11

### Ejercicio 12.7

En un archivo se quieren grabar las notas que tienen los alumnos de una asignatura junto al nombre del profesor y el resumen de aprobados y suspensos. La estructura será la siguiente:

- Primer registro con el nombre de la asignatura y curso.
- Segundo registro con el nombre del profesor, número de alumnos, de aprobados y suspensos.
- Cada uno de los alumnos, con su nombre y nota.

Se crea un archivo binario (modo `wb+`) con la estructura que se indica en el enunciado. Antes de escribir el segundo registro (profesor) se obtiene la posición actual, llamando a `fgetpos()`. Una vez que se han grabado todos los registros de alumnos, se sitúa como posición actual, llamando a `fgetpos()`, el registro del profesor con el fin de grabar el número de aprobados y suspensos. Naturalmente, según se pide las notas de los alumnos se contabiliza si está aprobado o suspenso. La entrada de datos se realiza desde el teclado.

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

typedef struct
{
 char asg[41];
 int curso;
} ASGTA;
typedef struct
{
 char nom[41];
 int nal, aprob, susp;
} PROFS;
typedef struct
{
 char nom[41];
 float nota;
} ALMNO;
void entrada(ALMNO* a);
```

```
void main(void)
{
 ASGTA a;
 PROFS h = {" ", 0, 0, 0}; /* valores iniciales: alumnos,
 aprobados, suspensos */
 ALMNO t;
 FILE* pf;
 int i;
 fpos_t* p = (fpos_t*)malloc(sizeof(fpos_t));

 pf = fopen("CURSO.DAT", "wb+");
 if (pf == NULL)
 {
 printf("Error al abrir el archivo, modo wb+");
 exit(-1);
 }
 printf("Asignatura: ");
 gets(a.asg);
 printf("Curso: ");
 scanf("%d%c", &a.curso);
 fwrite(&a, sizeof(ASGTA), 1, pf);
 printf("Nombre del profesor: ");
 gets(h.nom);
 printf("Número de alumnos: ");
 scanf("%d%c", &h.nal);
 fgetpos(pf, p); /* guarda en p la posición actual */
 fwrite(&h, sizeof(PROFS), 1, pf);

 for (i = 1; i <= h.nal; i++)
 {
 entrada(&t);
 if (t.nota <= 4.5)
 h.susp++;
 else
 h.aprob++;
 fwrite(&t, sizeof(ALMNO), 1, pf);
 }
 fflush(pf);
 fsetpos(pf, p); /*se sitúa en registro del profesor */
 fwrite(&h, sizeof(PROFS), 1, pf);
 fclose(pf);
}
void entrada(ALMNO* a)
{
 printf("Nombre: ");
 gets(a -> nom);
 printf("Nota: ");
 scanf("%f%c", &(a -> nota));
}
```

## 12.7 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 es:

C:>matrices

En la línea de órdenes se pueden añadir datos de entrada del programa. En el caso del programa matrices se pueden incluir, por ejemplo, las dimensiones de una matriz.

C:>matrices 4 5

Para que un programa C pueda capturar datos (información) en la línea de órdenes, la función `main()` tiene dos argumentosopcionales: el primero es un argumento entero que contiene el número de parámetros transmitidos al programa (incluye como argumento el nombre del programa). El segundo argumento contiene los parámetros transmitidos, en forma de cadenas de caracteres; por lo que el tipo de este argumento es un arreglo de apuntadores a `char`. Puede existir un tercer argumento que contiene las variables de entorno, definido también como arreglo de apuntadores 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 12.16

Escribir un programa el cual muestre 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 órdenes 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. Al objeto de que el blanco forme parte del argumento se debe encerrar entre dobles comillas. La salida de la ejecución de `ARGMTOS` (`ARGMTOS.EXE`):

```
Numero de argumentos 7
Argumentos de la linea 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 12.8

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.

El número mínimo de argumentos de la línea de órdenes ha de ser 3; entre otros, nombre del programa ejecutable, primer archivo, segundo archivo, etc. y el archivo nuevo. El programa debe comprobar este hecho. Para copiar un archivo se va a utilizar 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 y será invocada 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 órdenes, 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);
 }
}
```



## 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 son:

- `fopen( )`, `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( )`, `fseek( )`, `fsetpos( )` y `fgetpos( )` para desplazar el apuntador a una posición dada (en bytes).

Con estas funciones y otras que están disponibles se puede hacer cualquier tratamiento de un archivo, en modo texto o modo binario; en modo secuencial o modo directo.

Para asociar un archivo externo con un flujo, o también podríamos decir con el nombre interno en el programa (apuntador 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. El modo texto (`t`) es el modo de manera predeterminada de los archivos. 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*.



## Ejercicios

12.1 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.

12.2 Señalar 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;
}
```

12.3 Se tiene un archivo de caracteres de nombre “SALAS.DAT”. Escribir un programa para crear el archivo “SA-

LAS.BIN” con el contenido del primer archivo pero en modo binario.

12.4 La función `rewind( )` sitúa el apuntador del archivo en el inicio del archivo. Escribir una sentencia, con la función `fseek( )` que realice el mismo cometido.

12.5 Utilice 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.

12.6 Las funciones `fgetpos( )` y `fsetpos( )` devuelven la posición actual del apuntador del archivo, y establecen el apuntador en una posición dada. Escribir las funciones `pos_actual( )` y `mover_pos( )`, con los prototipos:

```
int pos_actual(FILE* pf, long* p);
int mover_pos(FILE* pf, const long* p);
```

La primera función devuelve en *p* la posición actual del archivo. La segunda función establece el apuntador del archivo en la posición *p*.

**12.7** Un archivo contiene enteros positivos y negativos. Utilizar la función `fscanf( )` para leer el archivo y determinar el número de enteros negativos.

**12.8** Un archivo de caracteres quiere visualizarse en la pantalla. Escribir un programa para ver el archivo, cuyo nombre viene dado en la línea de órdenes, en pantalla.

**12.9** Escribir una función que devuelva una cadena de caracteres de longitud *n* del archivo cuyo apuntador 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);
```

**12.10** Se quieren 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.

**12.11** Escribir una función que tenga como argumentos un apuntador de un archivo de texto, un número de línea inicial y otro número de línea final. La función debe mostrar las líneas del archivo comprendidas entre los límites indicados.

**12.12** Escribir un programa que escriba por pantalla las líneas de texto de un archivo, numerando cada línea del mismo.

## Problemas

**12.1** 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.

**12.2** Un atleta utiliza un pulsómetro para sus entrenamientos. El mecanismo almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de dos 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.

**12.3** 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.

**12.4** 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 escribir el par de valores en el archivo.

**12.5** 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.

**12.6** 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 contiene el nombre, la segunda está en blanco, la tercera la dirección y en la quinta empieza el texto `CARTA.TXT`.

- 12.7 Una librería guarda en el archivo `LIBER.DAT` la siguiente información sobre cada uno de sus libros: código, autor, título, precio de compra y precio de venta. El archivo está ordenado por los códigos de los libros (de tipo entero). Se precisa un programa con las opciones:
1. Añadir. Permitirá insertar nuevos registros, el archivo debe mantenerse ordenado.
  2. Consulta. Buscará un registro por el campo código y lo mostrará por pantalla.
  3. Modificar. Permitirá cambiar el precio de venta de un libro.
- 12.8 Escribir un programa para listar el contenido de un determinado subdirectorio, pasado como parámetro a la función `main()`.
- 12.9 Modificar el problema 12.2 para añadir un menú con opciones de añadir nuevos entrenamientos, obtener el tiempo que está por encima del umbral aeróbico (dato pedido por teclado) para un día determinado y la media de las pulsaciones.
- 12.10 Un archivo de texto consta en cada línea de dos cadenas de enteros separadas por el operador `+`, o `-`. Se quiere formar un archivo binario con los resultados de la operación que se encuentra en el archivo de texto.

PARTE

III

# Lenguaje unificado de modelado UML 2.5





# Programación orientada a objetos y UML 2.5

## Contenido

- 13.1 Programación orientada a objetos
- 13.2 Modelado e identificación de objetos
- 13.3 Propiedades fundamentales de la orientación a objetos
- 13.4 Modelado de aplicaciones: UML
- 13.5 Modelado y modelos

## 13.6 Diagramas de UML 2.5

- 13.7 Bloques de construcción (componentes) de UML 2.5
- 13.8 Especificaciones de UML
- 13.9 Historia de UML
  - › Resumen
  - › Ejercicios

## Introducción

La programación orientada a objetos (POO) es un enfoque conceptual específico para diseñar programas, utilizando un lenguaje de programación orientado a objetos, por ejemplo, C++ o Java. Las propiedades más importantes de la POO son:

- Abstracción.
- Encapsulamiento y ocultación de datos.
- Polimorfismo.
- Herencia.
- Reusabilidad o reutilización de código.

Este paradigma de programación supera las limitaciones que soporta la programación tradicional o “procedimental” y por esta razón se comenzará el capítulo con una breve revisión de los conceptos fundamentales de este paradigma de programación.

Los elementos fundamentales de la POO son las *clases* y *objetos*. En esencia, la POO se concentra en el objeto como lo percibe el usuario, pensando en los datos que se necesitan para describir el objeto y las *operaciones* que describirán la iteración del usuario con los datos. Después se desarrolla una descripción de la interfaz externa y se decide cómo implementar la interfaz y el almacenamiento de datos. Por último, se juntan en un programa que utilice su nuevo dueño.

En este texto nos limitaremos al campo de la programación, pero es también posible hablar de sistemas de administración de bases de datos orientadas a objetos, sistemas operativos orientados a objetos, interfaces de usuarios orientadas a objetos, etcétera.

En el capítulo se hace una introducción a **UML** como *Lenguaje Unificado de Modelado*. UML se ha convertido *de facto* en el estándar para modelado de aplicaciones de software y es un lenguaje con sintaxis y semántica propias; se compone de **pseudocódigo**, código real, programas, ... Se describe también en el capítulo una breve historia de UML desde la ya mítica versión 0.8 hasta la actual versión **2.5** aunque el gran público sigue denominando a la versión en uso, **UML 2.0**.



### Conceptos clave

- › Abstracción
- › Atributos
- › Clases y objetos
- › Clase base
- › Clase derivada
- › Comportamiento
- › Correspondencia entre objetos
- › Encapsulación o encapsulamiento
- › Estado
- › Función miembro
- › Herencia
- › Instancia
- › Mensaje
- › Método
- › Ocultación
- › Operaciones
- › Polimorfismo
- › Reutilización
- › Reusabilidad
- › Sobrecarga
- › Tipo abstracto de datos
- › Variables de instancia

## 13.1 Programación orientada a objetos

Al contrario del enfoque procedimental que se basaba en la interrogante: ¿qué hace este programa?, el enfoque orientado a objetos responde a otra interrogante: ¿qué objetos del mundo real puede modelar?

La POO se basa en el hecho de que se debe dividir el programa, no en tareas, sino en modelos de objetos físicos o simulados. Aunque esta idea parece abstracta a primera vista, se vuelve más clara cuando se consideran objetos físicos en términos de sus *clases*, *componentes*, *propiedades* y *comportamiento*, y sus objetos instanciados o creados de las clases.

Si se escribe un programa de computadora en un lenguaje orientado a objetos, se está creando, en su computadora, un modelo de alguna parte del mundo. Las partes que el modelo construye son los objetos que aparecen en el dominio del problema. Estos objetos deben ser representados en el modelo que se está creando en la computadora.

Los objetos se pueden agrupar en categorías, y una clase describe, de un modo abstracto, todos los objetos de un tipo o categoría determinada.

### Consejo de programación

Los objetos en C++, o en Java, modelan objetos del problema en el dominio de la aplicación.

### Consejo de programación

Los objetos se crean a partir de las clases. La clase describe el tipo del objeto; los objetos representan *instanciaciones* individuales de la clase.

La idea fundamental de la orientación a objetos y de los lenguajes que implementan este paradigma de programación es combinar (encapsular) en una sola unidad tanto los datos como las funciones que operan (manipulan) sobre los datos. Esta característica permite modelar los objetos del mundo real de un modo mucho más eficiente que con funciones y datos. Esta unidad de programación se denomina **objeto**.

Las funciones de un objeto, denominadas *funciones miembro* (en C++) o *métodos* (Java y otros lenguajes de programación), constituyen el único sistema para acceder a sus datos. Si se desea leer datos de un objeto se llama a una *función miembro* del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos están ocultos y se dice que junto con las funciones están encapsulados en una entidad única. La *encapsulación* o *encapsulamiento* de los datos y la *ocultación* de los datos son conceptos clave en programación orientada a objetos.

La modificación de los datos de un objeto se realiza a través de una de las funciones miembro de ese objeto que interactúa con él y ninguna otra función puede acceder a los datos. Esta propiedad facilita la escritura, depuración y mantenimiento de un programa.

En un sistema orientado a objetos, un programa se organiza en un conjunto finito de objetos que contienen datos y operaciones (*funciones miembro* o *métodos*) que se comunican entre sí mediante *mensajes* (llamadas a funciones miembro). La estructura de un programa orientado a objetos se muestra en la figura 13.1.

Las etapas necesarias para modelar un sistema (resolver en consecuencia un problema) empleando orientación a objetos son:

1. Identificación de los objetos del problema.
2. Agrupamiento en *clases* (tipos de objetos) de los objetos con características y comportamiento comunes.

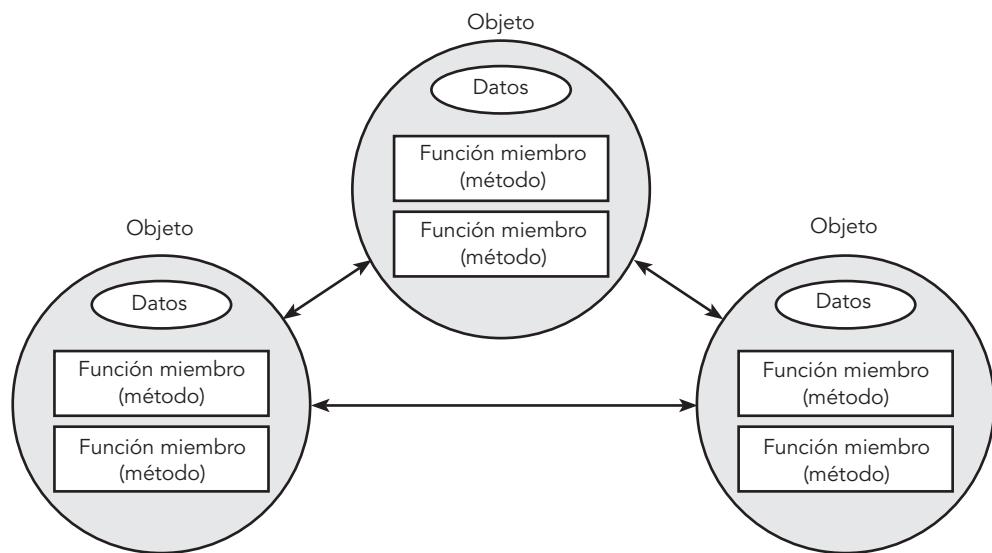


Figura 13.1 Organización típica de un programa orientado a objetos.

3. Identificación de los datos y operaciones de cada una de las clases.
4. Identificación de las *relaciones* existentes entre las diferentes clases del modelo.

Los objetos encapsulan datos y funciones miembro o métodos que manipulan los datos. Las funciones miembro también se conocen como métodos (dependiendo de los lenguajes de programación), por esta razón suelen ser términos sinónimos. Los elementos dato de un objeto se conocen también como *atributos* o *variables de instancia* (ya que *instancia* es un objeto específico). La llamada a una función miembro de un objeto se conoce como *envío de un mensaje* al objeto.

### Objeto

- Funciones miembro o métodos:
- Atributos o variables de instancia (datos):

### Envío de un mensaje a un objeto

- Llamada a la función miembro (mensaje):

## Objetos

Una forma de reducir la complejidad es, como se verá con más profundidad en los siguientes apartados, la *abstracción*. Las características y procesos de cualquier sistema se resumen en los aspectos esenciales y más relevantes; de este modo, las características complejas de los sistemas se vuelven más manejables.

En computación, la abstracción es el proceso crucial de representar la información en términos de su interfaz con el usuario. Es decir, se abstraen las características operacionales esenciales de un problema y se expresa su solución en dichos términos. La abstracción se manifiesta en C++ o en Java con el diseño de una clase que implementa la interfaz y que no es más que un tipo de dato específico.

Un ejemplo de abstracción expresada de diferentes formas según la aplicación a desarrollar, puede ser el término o clase *auto*:

- Un *auto* es la composición o combinación de diferentes partes (*motor*, *cuatro ruedas*, *3 o 5 puertas*, *asientos*, etcétera).
- Un *auto* también es un término común que define a tipos diferentes de automóviles; se pueden clasificar por el fabricante (*BMW*, *Seat*, *Opel*, *Toyota*, ...), por su categoría (o uso) (*deportivo*, *todo terreno*, *sedán*, *pick-up*, *limousine*, *coupé*, etcétera).

Si los miembros de autos o las diferencias entre autos individuales no son relevantes, entonces se utilizan expresiones y operaciones como: se fabrican autos; se usan autos para ir de la Ciudad de México a Guadalajara; se descompone el auto en sus partes, ...

El **objeto** es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y que juega un papel o un rol. Cuando se programa de modo orientado a objetos se trata de descubrir e implementar los objetos que juegan un rol en el dominio del problema del programa. La estructura interna y el comportamiento de un objeto, en consecuencia, no es prioritario durante el modelado del problema. Es importante considerar que un objeto, como un *auto* juega un rol o papel importante.

Dependiendo del problema, diferentes aspectos de un objeto son significativos. Así, los *atributos* indican propiedades de los objetos: propietario, marca, año de matriculación, potencia, etc. El objeto también tiene funciones que actuarán sobre los atributos o datos: matricular, comprar, vender, acelerar, frenar, etcétera.

Un objeto no tiene que ser necesariamente algo concreto o tangible. Puede ser por completo abstracto y puede también describir un proceso. Un equipo de baloncesto puede ser considerado como un objeto, los atributos pueden ser los jugadores, el color de sus camisetas, partidos jugados, tiempo de juego, etc. Las clases con atributos y funciones miembro permiten gestionar los objetos dentro de los programas.

## Tipos abstractos de datos: Clases

Un progreso importante en la historia de los lenguajes de programación se produjo cuando se comenzó a combinar juntos diferentes elementos de datos y, por consiguiente, encapsular o empaquetar distintas propiedades en un tipo de dato. Estos tipos fueron las *estructuras* o *registros* que permiten a una variable contener datos que pertenecen a las circunstancias representadas por ellas.

Las estructuras representan un modo de abstracción con los programas, concretamente la combinación (o *composición*) de partes diferentes o elementos (*miembros*). Así, por ejemplo, una estructura *auto* constará de miembros como marca, motor, número de matrícula, año de fabricación, etcétera.

Sin embargo, aunque las propiedades individuales de los objetos de los miembros se pueden almacenar en las estructuras o registros, es decir, cómo están organizados en la práctica, no pueden representar qué se puede hacer con ellos (moverse, acelerar, frenar, etc., en el caso de un *auto*). Se necesita que las operaciones que forman la interfaz de un objeto se incorporen también al objeto.

El *tipo abstracto de datos* (*TAD*) describe no solo los atributos de un objeto sino también su comportamiento (*operaciones* o *funciones*) y, en consecuencia, se puede incluir una descripción de los estados que puede tener el objeto.

Así, un objeto "equipo de baloncesto" no solo puede describir a los jugadores, la puntuación, el tiempo transcurrido, el periodo de juego, etc., sino que también se pueden representar *operaciones* como "sustituir un jugador", "solicitar tiempo muerto", ..., o *restricciones* como, "el momento en que comienza es en 0:00 y el momento en que termina cada cuarto de juego es a los 15:00", o incluso las detenciones del tiempo, por lanzamientos de tiros de castigo.

El término tipo abstracto de dato se conoce en programación orientada a objetos con el término *clase*. Una **clase** es la implementación de un tipo abstracto de dato y describe no solo los atributos (datos) de un objeto sino también sus operaciones (comportamiento). Así, la clase *auto* define que un coche consta de motor, ruedas, placa de matrícula, etc., y se puede conducir (manejar), acelerar, frenar, etcétera.

## Instancias

Una clase describe un objeto, en la práctica múltiples objetos. En conceptos de programación, una clase es, realmente, un tipo de dato, y se pueden crear, en consecuencia, variables de ese tipo. En programación orientada a objetos, a estas variables se les denomina *instancias* ("instances"), y también por sus sinónimos ejemplares, casos, etcétera.

Las instancias son la implementación de los objetos descritos en una clase. Estas instancias constan de los datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas en la propia clase.

En un lenguaje de programación OO objeto e instancia son términos sinónimos. Así, cuando se declara una variable de tipo *Auto*, se crea un objeto *Auto* (una instancia de la clase *Auto*).

## Métodos

En programación orientada a objetos, las operaciones definidas para los objetos, se denominan, como ya se ha comentado, *métodos*. Cuando se llama a una operación de un objeto se interpreta como el envío de un *mensaje* a dicho objeto.

Un programa orientado a objetos se forma enviando mensajes a los objetos, que a su vez producen (envían) más mensajes a otros objetos. Así cuando se llama a la operación "conducir" (*manejar*) para

un objeto *auto* en realidad lo que se hace es enviar el mensaje "conducir" al objeto *auto*, que procesa (ejecuta), a continuación, el método correspondiente.

En la práctica un programa orientado a objetos es una secuencia de operaciones de los objetos que actúan sobre sus propios datos.

### A recordar

Un **objeto** es una **instancia** o **ejemplar** de una **clase** (categoría o tipo de datos). Por ejemplo, un alumno y un profesor son instancias de la clase *Persona*. Un objeto tiene una *estructura*, como ya se ha comentado. Es decir tiene *atributos* (*propiedades*) y *comportamiento*. El comportamiento de un objeto consta de operaciones que se ejecutan. Los *atributos* y las *operaciones* se llaman *características* del objeto.

### Ejemplos

Un objeto de la clase *Persona* puede tener estos atributos: *altura*, *peso* y *edad*. Cada uno de estos atributos es único ya que son los valores específicos que tiene cada persona. Se pueden ejecutar estas operaciones: *dormir*, *leer*, *escribir*, *hablar*, *trabajar*, *correr*, etc. En C++, o en Java, estas operaciones dan lugar a las funciones miembro, o métodos: *dormir()*, *leer()*, *escribir()*, *hablar()*, *trabajar()*, *correr()*, etc. Si tratamos de modelar un sistema académico con profesores y alumnos, estas y otras operaciones y atributos pertenecerán a dichos objetos y clases.

En el mundo de la orientación a objetos, una clase sirve para otros propósitos que los indicados de clasificación o categoría. Una clase es una plantilla (tipo de dato) para hacer o construir objetos. Por ejemplo, una clase *Lavadora* puede tener los atributos *nombreMarca*, *númeroSerie*, *capacidad* y *potencia*; y las operaciones *encender()*, *(prender)*, *apagar()*, *lavar()*, *aclarar()*, ... .

### Consejo de programación

Un antípode de reglas de notación en UML 2.0 (adelante se describe más en detalle la notación UML 2.5).

- El nombre de la clase comienza con una letra mayúscula (*Lavadora*).
- El nombre de la clase puede tener varias palabras y todas comenzar con mayúsculas, por ejemplo *PáginaWeb*.
- El nombre de una característica (atributo u operación) comienza con una letra minúscula (*estatura*).
- El nombre de una característica puede constar a su vez de dos palabras, en este caso la primera comienza con minúscula y la segunda con mayúscula (*nombreMarca*).
- Un par de paréntesis sigue al nombre de una operación, *limpiar()*.

Esta notación facilita el añadido o supresión de características por lo cual representan gráficamente con mayor precisión un modelo del mundo real; por ejemplo, a la clase *Lavadora* se le podría añadir *velocidadMotor*, *volumen*, *aceptarDetergente()*, *ponerPrograma()*, etcétera.

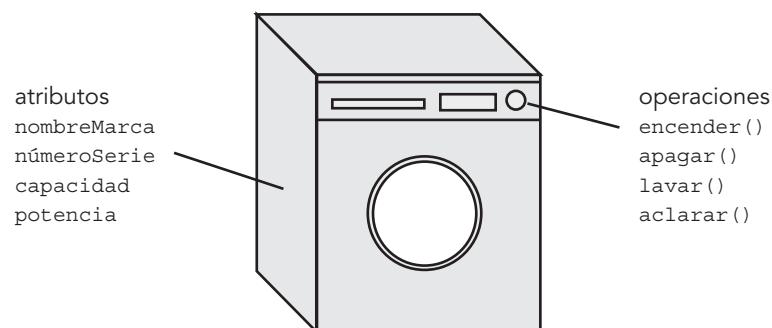


Figura 13.2 Notación gráfica de la clase *Lavadora* en UML.

## 13.2 Modelado e identificación de objetos

Un objeto en software es una entidad individual de un sistema que guarda una relación directa con los objetos del mundo real. La *correspondencia entre objetos* de programación y objetos del mundo real es el resultado práctico de combinar atributos y operaciones, o datos y funciones. Un objeto tiene un *estado*, un *comportamiento* y una *identidad*.

### Estado

Conjunto de valores de todos los atributos de un objeto en un instante de tiempo específico. El estado de un objeto viene determinado por los valores que toman sus datos o atributos. Estos valores han de cumplir siempre las restricciones (*invariantes de clase*, para objetos pertenecientes a la misma clase) que se hayan impuesto. El estado de un objeto tiene un carácter dinámico que evoluciona con el tiempo, con independencia de que ciertos elementos del objeto puedan permanecer constantes.

### Comportamiento

Conjunto de operaciones que se pueden realizar sobre un objeto. Las operaciones pueden ser de *observación* del estado interno del objeto, o bien de *modificación* de dicho estado. El estado de un objeto puede evolucionar en función de la aplicación de sus operaciones. Estas operaciones se realizan tras la recepción de un *mensaje* o estímulo externo enviado por otro objeto. Las interacciones entre los objetos se representan mediante *diagramas de objetos*. En UML se representarán por enlaces en ambas direcciones.

### Identidad

Permite diferenciar los objetos de modo no ambiguo independientemente de su estado. Es posible distinguir dos objetos en los cuales todos sus atributos sean iguales. Cada objeto posee su propia identidad de manera implícita. Cada objeto ocupa su propia posición en la memoria de la computadora.

En un sistema orientado a objetos los programas se organizan en conjuntos finitos que contienen atributos (datos) y operaciones (funciones) y que se comunican entre sí mediante mensajes. Los pasos típicos en el modelado de un sistema orientado a objetos son:

1. Identificar los *objetos* que forman parte del modelo.
2. Agrupar en *clases* todos aquellos objetos que tengan características y comportamientos comunes.
3. Identificar los *atributos* y las *operaciones* de cada clase.
4. Identificar las *relaciones* existentes entre las clases.

Cuando se diseña un problema en un lenguaje orientado a objetos, se debe pensar en dividir dicho problema en objetos, o dicho de otro modo, es preciso identificar y seleccionar los objetos del dominio del problema de modo que exista una correspondencia entre los objetos desde el punto de vista de programación y los objetos del mundo real.

¿Qué tipos de cosas se convierten en objetos en los programas orientados a objetos? La realidad es que la gama de casos es infinita y solo dependen de las características del problema a resolver y la imaginación del programador. Algunos casos típicos son:

| Personas    | Partes de una computadora | Estructuras de datos |
|-------------|---------------------------|----------------------|
| Empleados   | Pantalla                  | Pilas                |
| Estudiantes | Unidades de CD, DVD       | Colas                |
| Clientes    | Impresora                 | Listas enlazadas     |
| Profesores  | Teclado                   | Árboles              |
| Ingenieros  | WebCam                    | Grafos               |



Figura 13.3 Correspondencia entre objetos por mensajes.

| <i>Edición de revistas</i> | <i>Objetos físicos</i> | <i>Archivos de datos</i> |
|----------------------------|------------------------|--------------------------|
| Artículos                  | Autos                  | Diccionario              |
| Nombres de autores         | Mesas                  | Inventario               |
| Volumen                    | Carros                 | Listas                   |
| Número                     | Ventanas               | Ficheros                 |
| Fecha de publicación       | Farolas                | Almacén de datos         |

La correspondencia entre objetos de programación y objetos del mundo real se muestra en una combinación entre datos y funciones.

### 13.3 Propiedades fundamentales de la orientación a objetos

Los conceptos fundamentales de orientación a objetos que a su vez se constituyen en reglas de diseño en un lenguaje de programación son: abstracción, encapsulamiento y ocultación de datos, *herencia* (generalización), *reutilización* o *reusabilidad* y *polimorfismo*. Otras propiedades importantes son: envío de mensajes y diferentes tipos de relaciones como, asociaciones y agregaciones (capítulo 15).

#### Abstracción

La abstracción es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. En computación, la abstracción es la etapa crucial de representación de la información en términos de la interfaz con el usuario. La abstracción se representa con un tipo definido por el usuario, con el diseño de una clase que implementa la interfaz correspondiente. Una *clase* es un elemento en C++ o en Java, que traduce una abstracción a un tipo definido por el usuario. Combina representación de datos y métodos para manipular esos datos en un paquete.

La abstracción posee diversos grados denominados *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el análisis de un sistema hay que concentrarse en *¿qué hace?* y no en *¿cómo lo hace?*

El principio de la *abstracción* es más fácil de entender con una analogía del mundo real. Por ejemplo, una televisión es un aparato electrodoméstico que se encuentra en todos los hogares. Seguramente, usted está familiarizado con sus características y su manejo manual o con el mando (control) a distancia: encender (prender), apagar, cambiar de canal, ajustar el volumen, cambiar el brillo, ... y añadir componentes externos como altavoces, grabadoras de CD, reproductoras de DVD, conexión de un módem para internet, etc. Sin embargo, ¿sabe cómo funciona internamente?, ¿conoce cómo recibe la señal por la antena, por cable, por satélite, traduce la señal y la visualiza en pantalla? Normalmente *no* sabemos cómo funciona el aparato de televisión, pero *sí* sabemos cómo utilizarlo. Esta característica se debe a que la televisión separa claramente su *implementación interna* de su *interfaz externa* (el cuadro de mandos o control de su aparato o su mando a distancia). Actuamos con la televisión a través de su interfaz: los botones de alimentación, de cambio de canales, control de volumen, etc. No conocemos el tipo de tecnología que utiliza, el método de generar la imagen en la pantalla o cómo funciona internamente, es decir su *implementación*, ya que ello no afecta a su interfaz.

#### La abstracción en software

El principio de abstracción es similar en el software. Se puede utilizar código sin conocimiento de la implementación fundamental. En C++, por ejemplo, se puede hacer una llamada a la función `sqrt()`, declarada en el archivo de cabecera `<math.h>`, que puede utilizar sin necesidad de conocer la implementación del algoritmo real que calcula la raíz cuadrada. De hecho, la implementación fundamental del cálculo de la raíz cuadrada puede cambiar en las diferentes versiones de la biblioteca, mientras que la interfaz permanece igual.

También se puede aplicar el principio de abstracción a las clases. Así, se puede utilizar el objeto `cout` de la clase `ostream` para *fluir* (enviar) datos a la salida estándar, en C++, como

```
cout << " En un lugar de Cazorla \n";
```

o en pseudocódigo

```
escribir "En un lugar de Cazorla" <fin de línea>
```

En la línea anterior, se utiliza la interfaz documentada del operador de inserción `cout` con una cadena, pero no se necesita comprender cómo `cout` administra la visualización del texto en la interfaz del usuario. Solo necesita conocer la interfaz pública. Así, la implementación fundamental de `cout` es libre de cambiar, mientras que el comportamiento expuesto y la interfaz permanecen iguales.

La abstracción es el principio fundamental que se encuentra tras la reutilización. Solo se puede reutilizar un componente si en él se ha abstraído la esencia de un conjunto de elementos del confuso mundo real que aparecen una y otra vez, con ligeras variantes en sistemas diferentes.

## Encapsulamiento y ocultación de datos

La **encapsulación** o **encapsulamiento** es la reunión en una cierta estructura de todos los elementos que a un cierto nivel de abstracción se pueden considerar pertenecientes a una misma entidad y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.

En este caso, los objetos que poseen las mismas características y comportamiento se agrupan en clases que no son más que unidades de programación que encapsulan datos y operaciones. La encapsulación oculta lo que *hace* un objeto de lo que *hacen* otros objetos del mundo exterior, por lo que se denomina también *ocultación de datos*.

Un objeto tiene que presentar “una cara” al mundo exterior de modo que se puedan iniciar esas operaciones. El aparato de TV tiene un conjunto de botones, bien en la propia TV o incorporados en un mando o control a distancia. Una máquina lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura y el nivel del agua. Los botones de la TV y las marchas de la máquina lavadora constituyen la comunicación con el mundo exterior, las interfaces.

En esencia, la interfaz de una clase representa un “contrato” de prestación de servicios entre ella y los demás componentes del sistema. De este modo, los clientes de un componente solo necesitan conocer los servicios que este ofrece y no cómo están implementados internamente.

Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella. En general, esto implica mantener el contrato y se puede modificar la implementación de una clase sin afectar a las restantes clases relacionadas con ella; sólo es preciso mantener el contrato. Existe una separación de la interfaz y de la implementación. La interfaz pública establece *qué* se puede hacer con el objeto; de hecho, la clase actúa como una “caja negra”. Si bien la interfaz pública es estable, la implementación se puede modificar.

## Herencia

El concepto de clase conduce al concepto de herencia. En la vida diaria se utiliza el concepto de *clases* que se dividen a su vez en *subclases*. La clase *animal* se divide en *mamíferos*, *anfibios*, *insectos*, *pájaros*, etc. La clase *vehículo* se divide en *autos*, *camiones*, *autobuses*, *motos*, etc. La clase *electrodoméstico* se divide en *lavadora*, *frigorífico*, *tostadora*, *horno de microondas*, etcétera.

La idea principal de estos tipos de divisiones reside en el hecho de que cada subclase comparte características con la clase de la cual se deriva. Los autos, camiones, autobuses, motos, ..., tienen motor, ruedas y frenos. Además de estas características compartidas, cada subclase tiene sus propias características. Los autos, por ejemplo, pueden tener maletero (cofre), cinco asientos; los camiones, cabina y caja para transportar carga, etcétera.

La clase principal de la que derivan las restantes se denomina *clase base* (en C++), *clase padre* o *superclase* y las *subclases*, también se denominan *clases derivadas* (en C++).

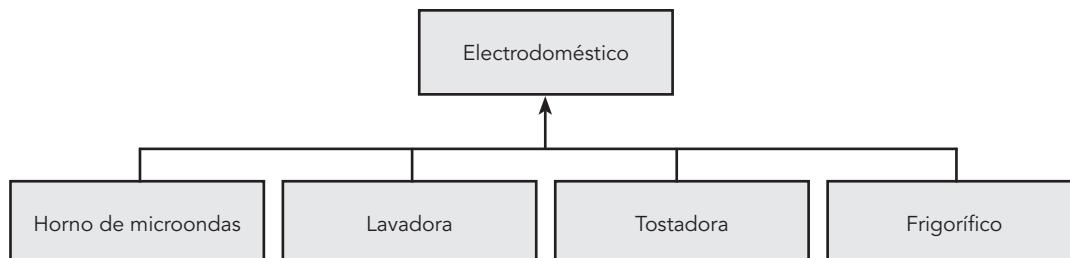


Figura 13.4 Herencia simple.

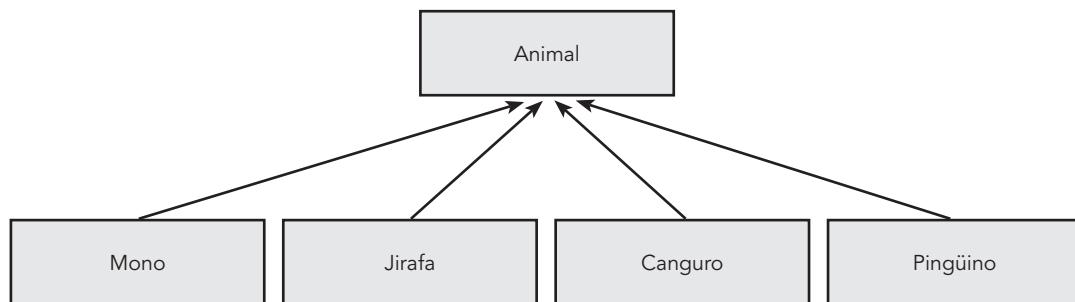


Figura 13.5 Herencia de la clase `Animal`.

Las clases modelan el hecho de que el mundo real contiene objetos con propiedades (*atributos*) y *comportamiento*. La herencia modela el hecho de que estos objetos tienden a organizarse en jerarquías. Esta jerarquía desde el punto de vista del modelado se denomina relación de *generalización* o *es-un (is-a)*. En programación orientada a objetos, la *relación de generalización* se denomina *herencia*. Cada *clase derivada* hereda las características de la clase base y además cada clase derivada añade sus propias características (atributos y operaciones). Las clases base pueden a su vez ser también subclases o clases derivadas de otras superclases o clases base.

Así, el programador puede definir una clase `Animal` que encapsula todas las propiedades o atributos (altura, peso, número de patas, etc.) y el comportamiento u operaciones (comer, dormir, andar) que pertenecen a cada animal. Los animales específicos como: `Mono`, `Jirafa`, `Canguro` o `Pingüino` tienen a su vez cada uno de ellos características propias.

Las técnicas de herencia se representan con la citada relación *es-un*. Así, se dice que un `Mono` es-un `Animal` que tiene características propias: puede subir a los árboles, saltar de un árbol a otro, etc.; además tiene en común con la `Jirafa`, el `Canguro` y el `Pingüino`, las características propias de cualquier animal (comer, beber, correr, dormir,...)

## Reutilización o reusabilidad

Otra propiedad fundamental de la programación orientada a objetos es la *reutilización* o *reusabilidad*. Este concepto significa que una vez que se ha creado, escrito y depurado una clase, se puede poner a disposición de otros programadores. El concepto es similar al uso de las bibliotecas de funciones en un lenguaje de programación procedural como C.

El concepto de herencia en C++ o Java proporciona una ampliación o extensión importante a la idea de *reusabilidad*. Una clase existente se puede ampliar añadiéndole nuevas características (atributos y operaciones). Esta acción se realiza derivando una nueva clase de la clase existente. La nueva clase derivada hereda las características de la clase base, pero podrá añadirle nuevas características.

## Reutilización de código

La facilidad de reutilizar o reusar el software existente es uno de los grandes beneficios de la POO. De este modo en una empresa de software se pueden reutilizar clases diseñadas en un proyecto en un nuevo proyecto con la consiguiente mejora de la productividad, al sacarle partido a la inversión realizada en el diseño de la clase primitiva.

¿En esencia cuáles son las ventajas de la herencia? Primero, reducción de código; las otras propiedades comunes de varias clases solo necesitan ser implementadas una vez y solo necesitan modificarse una vez si es necesario. Segundo, está soportado el concepto de abstracción de la funcionalidad común.

La facilidad con la que el software existente se puede reutilizar es una propiedad muy importante de la POO. La idea de usar código existente no es nueva en programación, ni, lógicamente en C++ o Java. Cada vez que se imprime algo con `cout` se está reutilizando código. No se escribe el código de salida a pantalla para visualizar los datos sino que se utiliza el flujo `ostream` existente para realizar el trabajo. Muchas veces los programadores no tienen en cuenta la ventaja de utilizar el código disponible y, como se suele decir en la jerga de la programación, “*inventan la rueda*” cada día, o al menos en cada proyecto.

Por ejemplo, supongamos que usted desea ejecutar un planificador de un sistema operativo (SO). El planificador (*scheduler*) es un componente del SO responsable de planificar procesos (cuáles se deben

ejecutar y durante qué tiempo). Para realizar esta planificación se suele recurrir a implementaciones basadas en prioridades, para lo cual se requiere una estructura de datos denominada *cola de prioridades* que almacena los procesos a la espera de que se vayan ejecutando en función de su prioridad. Existen en C++ dos métodos: 1) escribir el código correspondiente a una *cola de prioridades*; 2) la biblioteca estándar de plantillas STL incorpora un contenedor denominado *priority-queue* (cola de prioridad) que se puede utilizar para almacenar objetos de cualquier tipo. Lo más sensato es incorporar en su programa el contenedor *priority-queue* de la biblioteca STL en un diseño del planificador, en lugar de reescribir su propia cola de prioridad.

### Reescritura de código reusable

La escritura de código reusable es importante en el diseño de un código. Debe diseñar sus programas de modo que pueda reutilizar sus clases, sus algoritmos y sus estructuras de datos. Tanto usted como sus colegas en el proyecto en que trabaje deben poder utilizar los componentes anteriores, tanto en el proyecto actual como en futuros proyectos. En general, se debe evitar diseñar código en exceso o específico para casos puntuales; *siempre que sea posible reutilice código existente*.

Una técnica del lenguaje C++ para escribir código de propósito general es utilizar *plantillas (templates)*. En lugar de escribir, por ejemplo, una estructura de datos *Pila* para tratar números reales, etc., es preferible diseñar una plantilla o tipo genérico *Pila* que le sirva para cualquier tipo de dato a procesar.

## Polimorfismo

El *polimorfismo* es la propiedad que permite a una operación (función) tener el mismo nombre en clases diferentes y que actúe de modo distinto en cada una de ellas. Esta propiedad es intrínseca a la vida ordinaria ya que una misma operación puede realizar diferentes acciones dependiendo del *objeto* sobre el que se aplique. Así, por ejemplo, se puede *abrir* una puerta, *abrir* una ventana, *abrir* un libro, *abrir* un periódico, *abrir* una cuenta corriente en un banco, *abrir* una conversación, *abrir* un congreso, etc. En cada caso se realiza una operación diferente. En orientación a objetos, cada clase “*conoce*” cómo realizar esa operación.

En la práctica, el polimorfismo significa la capacidad de una operación de ser interpretada solo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente qué operación ha sido invocada.

En la vida diaria hay numerosos ejemplos de polimorfismo. En un taller de reparaciones de automóviles existen numerosos autos de marcas diferentes, de modelos distintos, de potencias diferentes, de carburantes distintos, etc. Constituyen una clase o colección heterogénea de autos. Supongamos que se ha de realizar una operación típica “cambiar los frenos del auto”. La operación a realizar es la misma, incluye los mismos principios de trabajo, sin embargo, dependiendo del auto, en particular, la operación será muy diferente, incluirá distintas acciones en cada caso. La propiedad de polimorfismo, en esencia, es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de distintas formas en cada clase.

El polimorfismo es importante tanto en el modelado de sistemas como en el desarrollo de software. En el modelado ya que el uso de palabras iguales tiene comportamientos distintos, según el problema a resolver; en software, ya que el polimorfismo toma ventaja de la propiedad de la herencia.

En el caso de operaciones en C++ o Java, el polimorfismo permite que un objeto determine en tiempo de ejecución la operación a realizar. Supongamos, por ejemplo, que se trata de realizar un diseño gráfico para representar figuras geométricas, como triángulo, rectángulo y circunferencia, y que se desea calcular la superficie o el perímetro (longitud) de cada figura. Cada figura tiene un método u operación distinta para calcular su perímetro y su superficie. El polimorfismo permite definir una única función *calcularSuperficie*, cuya implementación es diferente en la clase *Triángulo*, *Rectángulo* o *Circunferencia*, y cuando se selecciona un objeto específico en tiempo de ejecución, la función que se ejecuta es la correspondiente al objeto específico de la clase seleccionada.

En C++ existe otra propiedad importante derivada del polimorfismo y es la **sobrecarga de operadores y funciones**, que es un tipo especial de polimorfismo. La *sobrecarga* básica de operadores existe siempre. El operador *+* sirve para sumar números enteros o reales, pero si desea sumar números complejos deberá sobrecargar el operador *+* para que permita realizar esta suma.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se denomina polimorfismo (una cosa con distintas formas). Sin embargo, cuando a un operador existente, como *+* o *=*, se le permite la posibilidad de operar con diferentes tipos de datos, se dice que

dicho operador está sobrecargado. *La sobrecarga es un tipo especial de polimorfismo* y una característica sobresaliente de los lenguajes de programación orientada a objetos.

#### Lenguajes de programación orientada a objetos

Los lenguajes de programación orientados a objetos utilizados en la actualidad son numerosos, y aunque la mayoría siguen criterios de terminología universales, puede haber algunas diferencias relativas a su consideración de:

*puros* (Smalltalk, Eiffel, ...) *híbridos* (Object Pascal, VB .NET, C++, Java, C#, Objective-C...)

### 13.4 Modelado de aplicaciones: UML

El Lenguaje Unificado de Modelado (UML, *Unified Model Language*) es el lenguaje estándar de modelado para desarrollo de sistemas y de software. UML se ha convertido *de facto* en el estándar para modelado de aplicaciones software y ha crecido su popularidad en el modelado de otros dominios. Tiene una gran aplicación en la representación y modelado de la información que se utiliza en las fases de análisis y diseño. En diseño de sistemas, se modela por una importante razón: *gestionar la complejidad*.

Un modelo es una abstracción de cosas reales. Cuando se modela un sistema, se realiza una abstracción ignorando los detalles que sean irrelevantes. El modelo es una simplificación del sistema real. Con un lenguaje formal de modelado, el lenguaje es abstracto aunque tan preciso como un lenguaje de programación. Esta precisión permite que un lenguaje sea legible por la máquina, de modo que pueda ser interpretado, ejecutado y transformado entre sistemas.

Para modelar un sistema de modo eficiente, se necesita algo muy importante: un lenguaje que pueda describir el modelo. ¿Qué es UML? UML es un **lenguaje**. Esto significa que tiene tanto sintaxis como semántica y se compone de: pseudocódigo, código real, dibujos, programas, descripciones, ... . Los elementos que constituyen un lenguaje de modelado se denominan **notación**.

El bloque básico de construcción de UML es un *diagrama*. Existen tipos diferentes, algunos con propósitos muy específicos (*diagramas de tiempo*) y algunos con usos más genéricos (*diagramas de clases*).

Un lenguaje de modelado puede ser cualquier cosa que contiene una *notación* (un medio de expresar el modelo) y una *descripción* de lo que significa esa notación (un *metamodelo*). Aunque existen diferentes enfoques y visiones de modelado, UML tiene un gran número de ventajas que lo convierten en un lenguaje idóneo para un gran número de aplicaciones como:

- Diseño de software.
- Software de comunicaciones.
- Proceso de negocios.
- Captura de detalles acerca de un sistema, proceso u organización en análisis de requisitos.
- Documentación de un sistema, proceso o sistema existente.

UML se ha aplicado y se sigue aplicando en un sinfín de dominios, como:

- |                            |                                        |
|----------------------------|----------------------------------------|
| • Banca.                   | Industria del software.                |
| • Salud.                   | Desarrollo de aplicaciones web.        |
| • Defensa.                 | Desarrollo de aplicaciones educativas. |
| • Computación distribuida. | Sistemas empotrados.                   |
| • Sistemas en tiempo real. | Sector empresarial y de negocios.      |

El bloque básico fundamental de UML es un diagrama. Existen diferentes tipos, algunos con propósitos muy específicos (*diagramas de tiempos*) y algunos con propósitos más genéricos (*diagramas de clase*).

#### Lenguaje de modelado

Un **modelo** es una simplificación de la realidad que se consigue mediante una abstracción. El modelado de un sistema pretende capturar las partes fundamentales o esenciales de un sistema. El modelado se representa mediante una notación gráfica.

Un modelo se expresa en un **lenguaje de modelado** y consta de notación (símbolos utilizados en los modelos) y un conjunto de reglas que instruyen cómo utilizarlas. Las reglas son sintácticas, semánticas y pragmáticas.

- **Sintaxis.** Nos indica cómo se utilizan los símbolos para representar elementos y cómo se combinan los símbolos en el lenguaje de modelado. La sintaxis es comparable con las palabras del lenguaje natural; es importante conocer cómo se escriben correctamente y cómo poner los elementos para formar una sentencia.
- **Semántica.** Las reglas semánticas explican lo que significa cada símbolo y cómo se debe interpretar, bien por sí mismo o en el contexto de otros símbolos.
- **Pragmática.** Las reglas de pragmática definen las intenciones de los símbolos a través de los cuales se consigue el propósito de un modelo y se hace comprensible para los demás. Esta característica se corresponde en lenguaje natural con las reglas de construcción de frases que son claras y comprensibles. Para utilizar bien un lenguaje de modelado es necesario aprender estas reglas.

Un modelo UML tiene dos características muy importantes:

- **Estructura estática:** describe los tipos de objetos más importantes para modelar el sistema.
- **Comportamiento dinámico:** describe los ciclos de vida de los objetos y cómo interactúan entre sí para conseguir la funcionalidad requerida del sistema.

Estas dos características están estrechamente relacionadas entre sí.

UML como lenguaje de modelado visual es independiente del lenguaje de implementación, de modo que los diseños realizados usando UML se pueden implementar en cualquier lenguaje que soporte las características de UML, esencialmente lenguajes orientados a objetos como C++, C# o Java. Un lenguaje de modelado puede ser todo aquel que contenga una notación (un medio de expresar el modelo) y una descripción de lo que significa esa notación (un metamodelo).

¿Qué ventajas aporta UML a otros métodos, lenguajes o sistemas de modelado? Las principales ventajas de UML son:

- *Lenguaje formal*, cada elemento del lenguaje está rigurosamente definido.
- *Conciso*.
- *Comprensible y completo*, describe todos los aspectos importantes de un sistema.
- *Escalable*, sirve para gestionar grandes proyectos pero también pequeños proyectos.
- *Construido sobre la filosofía de "lecciones aprendidas"*, ya que recogió lo mejor de tres métodos populares (Booch, Rumbaugh, Jacobson) ya muy probados y desde su primer borrador, en 1994, ha incorporado las mejores prácticas de la comunidad de orientación a objetos.
- *Estándar*, ya que está avalado por la **OMG** (Object Management Group), organización con difusión y reconocimiento mundial.

## 13.5 Modelado y modelos

El enfoque principal de UML es el modelado. Sin embargo, ¿cuál es el significado de modelado?, ¿qué significa exactamente? Son preguntas abiertas que UML contesta adecuadamente. Modelado es un medio de capturar ideas, relaciones, decisiones y requerimientos en una notación bien definida que se puede aplicar a muchos dominios diferentes. Modelado no solo significa cosas diferentes para personas distintas, sino también utiliza diferentes piezas dependiendo de lo que se está tratando de transmitir.

En general, un modelo en UML se construye de uno o más *diagramas*. Un diagrama gráficamente representa cosas, y las relaciones entre estas cosas. Estas cosas pueden ser representaciones de objetos del mundo real, construcciones de software puras, o una descripción del comportamiento de algún otro objeto. Es frecuente que una cosa individual se muestre en diagramas múltiples; cada diagrama representa un interés particular, o vista, de la cosa que se está modelando. UML consta de un número de elementos gráficos que se combinan para formar diagramas. Ya que UML es un lenguaje, dispone de reglas para combinar estos elementos; su conocimiento será importante y también el conocimiento de los diagramas que le ayuden a diseñar el modelo.

El propósito de los diagramas es presentar múltiples vistas de un sistema; este conjunto de vistas múltiples se llama modelo. En la práctica un **modelo** es una descripción abstracta de un sistema o de un proceso, una representación simplificada que permite comprender y simular. Un modelo UML de un sistema es similar a un modelo en escala de un edificio junto con la representación artística del mismo. Es importante observar que un modelo UML de un sistema describe aquello que se supone hace. No indica cómo implementar el sistema. En el siguiente apartado se describen los diagramas fundamentales de UML.

El concepto de modelo es muy útil en numerosos campos, y en particular en los campos de la ciencia e ingeniería.

En el sentido más general, cuando se crea un modelo se está utilizando algo que se conoce en gran medida y trata de ayudar a comprender que ese algo se entienda mucho mejor. En algunos campos un modelo es un conjunto de ecuaciones; en otros, un modelo es una simulación computacional. Son posibles muchos tipos de modelos.

En nuestro caso, un modelo es un conjunto de diagramas que podemos examinar, evaluar y modificar con la finalidad de comprender y desarrollar un sistema. Se suele utilizar, a veces, el concepto de metamodelo como la descripción de modo formal de los elementos del modelado y la sintaxis y la semántica de la notación que permite su manipulación; en esencia, como ya se comentó anteriormente, es una descripción de la representación de las notaciones empleadas.

Un modelo es la unidad básica del desarrollo y se enlaza a una fase precisa del desarrollo y se construye a partir de elementos de modelado con sus diferentes vistas asociadas. El concepto de modelo es muy útil en numerosos campos, y, en particular, en ciencias e ingeniería.

En el campo de computación, un modelo es un conjunto de diagramas UML que se pueden examinar, evaluar y modificar con el objetivo de entender y desarrollar un sistema.

El término modelado se emplea a menudo como sinónimo de análisis, es decir, de descomposición en elementos simples, más fáciles de comprender. En computación (informática), el modelado consiste ante todo en describir un problema, luego en describir la solución de este problema; estas actividades se conocen, respectivamente, como el *análisis* y el *diseño*.

### Categorías de modelado

El término modelado expresa la descomposición en elementos simples más fáciles de comprender y diseñar. El modelado de un sistema se hace típicamente desde tres puntos de vista distintos:

- *Modelado de objetos* (descompone los sistemas en objetos colaboradores)
  - Aspectos estáticos y estructurales del sistema
- *Modelado dinámico* (describe los aspectos temporales “timeline” del sistema)
  - Aspectos temporales y de comportamiento del sistema
- *Modelado funcional* (descompone las tareas en funciones de realización más simples)
  - Aspectos de transformación funcional del sistema

Estos tres tipos generales de modelado se definen en UML por diferentes modelos que realizan la representación de los sistemas:

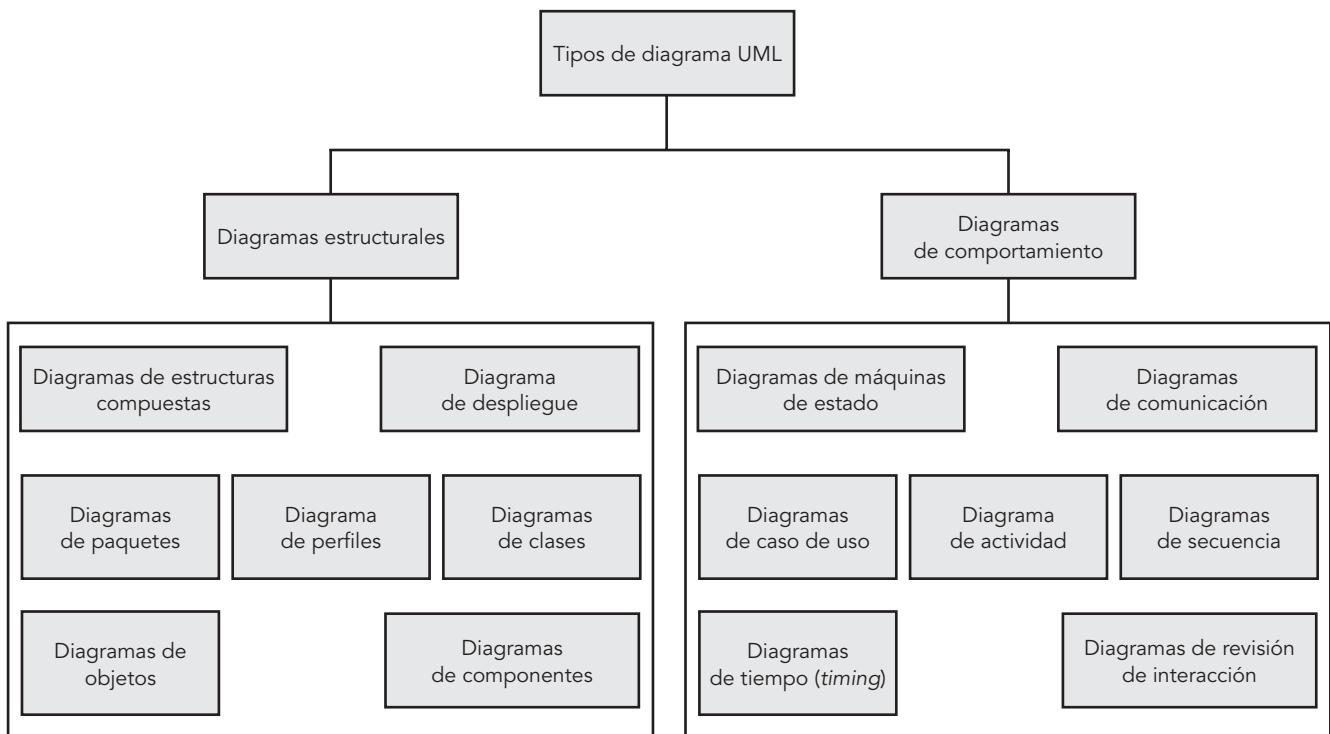
- Modelo de clases que captura la estructura estática;
- Modelo de estados que expresa el comportamiento dinámico de los objetos;
- Modelo de casos de uso que describe las necesidades del usuario;
- Modelo de interacción que representa los escenarios y los flujos de mensajes;
- Modelo de realización que muestra las unidades de trabajo;
- Modelo de despliegue que precisa el reparto de procesos.
- ...

Los modelos se ven y manipulan por los usuarios mediante vistas gráficas, verdaderas proyecciones a través de los elementos de modelado contenidos por uno o varios modelos. Se pueden construir numerosas vistas a partir de los modelos básicos que pueden mostrar la totalidad o parte de los mismos. A cada vista corresponden uno o varios diagramas. UML define 14 tipos de diagramas diferentes que veremos más adelante. UML es una representación gráfica normalizada de un modelo semántico común.

## 13.6 Diagramas de UML 2.5

UML es un lenguaje unificado de modelado y de propósito general y estandarizado (ISO/IEC 19501: 2005) se utiliza normalmente en ingeniería de software orientado a objetos, pero al ser un lenguaje muy rico, se puede utilizar para modelar estructuras de aplicaciones, comportamiento e incluso procesos de negocio.

Existen 14 tipos de diagramas en UML 2.5 que ayudan a modelar este comportamiento. La taxonomía de estos diagramas se divide en dos grandes categorías principales: diagramas de estructura (estructurales) y diagramas de comportamiento (figura 13.6).



Fuente: OMG y Creately.com (adaptada).

**Figura 13.6** Taxonomía de diagramas (notaciones) UML 2.5.

Los *diagramas estructurados* o *estructurales* se utilizan para capturar la organización física de las cosas del sistema, por ejemplo cómo se relacionan unos objetos con otros. Los diagramas estructurados son:

- Diagramas de clases.
- Diagramas de componentes.
- Diagramas de despliegue.
- Diagramas de objetos.
- Diagramas de paquetes.
- Diagramas de perfiles.
- Diagramas de estructuras compuestas.

El modelo estructural representa el marco de trabajo del sistema y este marco de trabajo es el lugar donde existen los demás componentes. De modo que los diagramas de clases, diagrama de componentes y diagramas de despliegue son las partes del modelado estructural; representan los elementos y mecanismos para ensamblarlos. Sin embargo el modelo estructural nunca describe el comportamiento dinámico del sistema. El diagrama de clase es el diagrama estructural más ampliamente utilizado.

Los *diagramas de comportamiento* se centran en el comportamiento de los elementos de un sistema y describen la interacción en el sistema. Representan la interacción entre los diagramas estructurales. En resumen, el modelado de comportamiento muestra la naturaleza dinámica del sistema. Por ejemplo, se pueden utilizar diagramas de comportamiento para capturar requerimientos, operaciones y cambios de los estados internos de los elementos. Los diagramas de comportamiento son:

- Diagramas de actividad.
- Diagramas de comunicación.
- Diagramas de revisión de interacción.
- Diagramas de secuencia.
- Diagramas de máquinas de estado.
- Diagramas de tiempo (*timing*).
- Diagramas de caso de uso.

### Diagramas de clase

Los diagramas de clase representan la estructura estática en términos de clases y relaciones; utilizan clases de interfaces para capturar detalles sobre las entidades que constituyen su sistema y las relaciones estáticas entre ellas. Los diagramas de clases son uno de los diagramas UML más utilizados en modelado y en la generación de código fuente en un lenguaje de programación.

### Diagramas de componentes

Representan los componentes físicos de una aplicación; muestran la organización y dependencias implicadas en la implementación de un sistema. Pueden agrupar elementos más pequeños, como clases, en piezas más grandes y desplegables.

### Diagramas de estructura compuesta

Los diagramas de estructura compuesta enlazan diagramas de clases y diagramas de componentes.

### Diagramas de despliegue

Estos diagramas muestran cómo un sistema se ejecuta realmente o se asignan a varias piezas de hardware. Normalmente, los diagramas de despliegue muestran cómo los componentes se configuran en tiempo de ejecución y representan el despliegue de los componentes sobre los dispositivos materiales.

### Diagramas de paquetes

Los diagramas de paquetes son realmente tipos especiales de diagramas de clases. Proporcionan un medio para visualizar dependencias entre partes de su sistema y se utilizan, a menudo, para examinar problemas o determinar la orden de compilación.

### Diagramas de objetos

Estos diagramas representan los objetos y sus relaciones y corresponden a diagramas de colaboración simplificados, sin representación de los envíos de mensajes. Utilizan la misma sintaxis que los diagramas de clase y muestran cómo las instancias reales de clases se relacionan en una instancia específica en un momento dado. Los diagramas de clases muestran instantáneas de las relaciones de su sistema en tiempo de ejecución.

### Diagramas de actividad

Los diagramas de actividades representan el comportamiento de una operación en términos de acciones. Capturan el flujo de un comportamiento o actividad, al siguiente. Son similares, en concepto, a los clásicos diagramas de flujos pero de manera más expresiva.

### Diagramas de comunicación

Estos diagramas son un tipo de diagrama de interacción que se centra en los elementos implicados en un comportamiento particular y cuáles mensajes se pasan en uno y otro sentido. Los diagramas de comunicación enfatizan más en los objetos implicados que en el orden y naturaleza de los mensajes intercambiados.

### Diagramas de descripción de la interacción

Estos diagramas son versiones simplificadas de diagramas de actividad. En lugar de enfatizar en la actividad de cada etapa, los diagramas de descripción de la interacción enfatizan en cuál o cuáles elementos están implicados en la realización de esa actividad.

### Diagramas de secuencia

Los diagramas de secuencia son una representación temporal de los objetos y sus interacciones; se centran en el tipo y orden de los mensajes que se pasan entre elementos durante la ejecución y son el tipo más común de diagrama de interacción y son muy intuitivos.

### Diagramas de máquinas de estados

Estos diagramas capturan las transiciones internas de los estados de un elemento. El elemento puede ser tan pequeño como una única clase o tan grande como el sistema completo. Los diagramas de estado se utilizan frecuentemente para modelar los sistemas embebidos y especificaciones o implementaciones de protocolos.

## Diagramas de tiempo

Son un tipo de diagrama de interacción que enfatiza especificaciones detalladas de tiempo en los mensajes. Se utilizan, normalmente, para modelar sistemas en tiempo real como sistemas de comunicación por satélites.

## Diagramas de casos de uso

Estos diagramas representan las funciones del sistema desde el punto de vista del usuario, o lo que es igual, se utilizan para capturar los requerimientos funcionales de un sistema. Proporcionan una vista de implementación independiente de lo que hace un sistema y permitir al modelador centrarse en las necesidades del usuario en lugar de en detalles de realización.

UML tiene un amplio vocabulario para capturar el comportamiento y los flujos de procesos. Los diagramas de actividad y los de estado se pueden utilizar para capturar procesos de negocio que implican a personas, grupos internos o incluso organizaciones completas. UML 2.5 tiene una notación que ayuda considerablemente a modelar fronteras geográficas, responsabilidades del trabajador, transacciones complejas, etcétera.

### A recordar

UML es un lenguaje con una sintaxis y una semántica (vocabulario y reglas) que permiten una comunicación.

Es importante considerar que UML *no es* un proceso de software. Esto significa que se utilizará UML dentro de un proceso de software pero está concebido, claramente, para ser parte de un enfoque de desarrollo iterativo.

## Desarrollo de software orientado a objetos con UML

UML tiene sus orígenes en la orientación a objetos. Por esta razón se puede definir UML como: “un lenguaje de modelado orientado a objetos para desarrollo de sistemas de software modernos”. Al ser un lenguaje de modelado orientado a objetos, todos los elementos y diagramas en UML se basan en el paradigma orientado a objetos. El desarrollo orientado a objetos se centra en el mundo real y resuelve los problemas a través de la interpretación de “objetos” que representan los elementos tangibles de un sistema.

La idea fundamental de UML es modelar software y sistemas como colecciones de objetos que interactúan entre sí. Esta idea se adapta muy bien al desarrollo de software y a los lenguajes orientados a objetos, así como a numerosas aplicaciones, como los procesos de negocios.

## 13.7 Bloques de construcción (componentes) de UML 2.5

Los bloques de construcción, elementos o cosas de UML 2.5 se dividen en cuatro categorías:

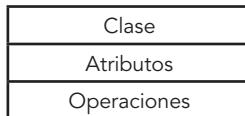
- Elementos estructurales.
- Elementos de comportamiento.
- Elementos de agrupación.
- Elementos de notación.

### Elementos estructurales

Las cosas estructurales definen la parte estática del modelo y representan elementos conceptuales y físicos. La descripción de las cosas estructurales de UML más importantes son las siguientes:

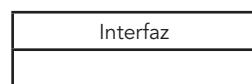
#### Clase

Una clase representa un conjunto de objetos que tienen responsabilidades similares.



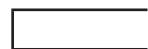
### Interfaz

Una interfaz define un conjunto de operaciones que especifican la responsabilidad de una clase.



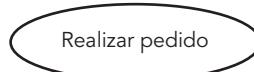
### Colaboración

La colaboración define la interacción entre elementos.



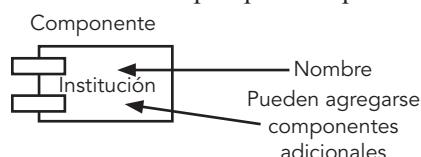
### Caso de uso

Un caso de uso representa un conjunto de acciones realizadas por un sistema para un objeto específico.



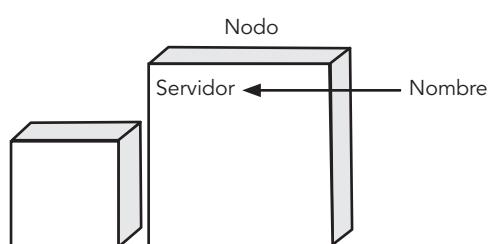
### Componente

El componente describe cualquier parte física de un sistema y se representan con un nombre en el interior. Se pueden añadir elementos adicionales siempre que se requieran



### Nodo

Un nodo se puede definir como un elemento físico que existe en tiempo de ejecución como un servidor, red, etcétera.



### Elementos de comportamiento

Una cosa de comportamiento consta de las partes dinámicas de los modelos UML y las más importantes son:

#### Interacción

La interacción se define como un comportamiento que consta de un grupo de mensajes intercambiados entre elementos para realizar una tarea específica.



#### Máquina de estado

La máquina de estado es útil cuando el estado de un objeto en su ciclo de vida es importante. Define la secuencia de estados de un objeto que se produce en respuesta a eventos. Los eventos son factores externos responsables del cambio de estado.

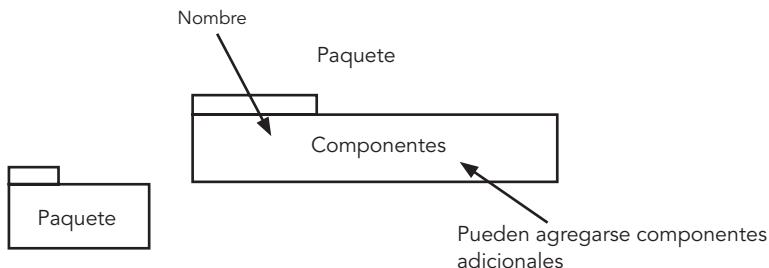


## Elementos de agrupación

Las cosas de agrupamiento se pueden definir como un mecanismo que agrupa los elementos juntos de un modelo. Existe una sola cosa de agrupamiento disponible:

### Paquete

El paquete es cosa que permite agrupar cosas estructurales y de comportamiento.



### Elementos de notación (notas)

Las cosas notacionales se definen como un mecanismo para capturar comentarios, descripciones y comentarios de los elementos del modelo UML. La nota es la única cosa notacional disponible.

### Nota

Una nota se utiliza para realizar comentarios, restricción, etc., de un elemento UML. Las notas añaden información a los diagramas del usuario para capturar cualquier cosa en su diagrama. Las notas se utilizan para expresar información adicional. Algunas herramientas permiten embeber enlaces URL en notas, proporcionando un medio para navegar de un diagrama al siguiente, documentos HTML, etc.



## 13.8 Especificaciones de UML

El lenguaje unificado de modelado (UML, The Unified Modeling Language™), es la especificación más utilizada de OMG y el modo en que se usan los modelos del mundo, no solo la estructura de la aplicación, comportamiento y arquitectura, sino también los procesos de negocio y estructura de datos. La documentación completa de UML se muestra en el sitio [www.omg.org](http://www.omg.org) y en ella puede encontrar manuales, tutoriales, normas de certificaciones... para iniciarse en modelado así como los beneficios de modelado en el desarrollo de sus aplicaciones.

La especificación actual oficial y las especificaciones de UML (1), así como los perfiles UML y especificaciones relacionadas (2) pueden descargarse en:

1. [www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm](http://www.omg.org/technology/documents/modeling_spec_catalog.htm)
2. [www.omg.org/technology/documents/profile\\_catalog.htm](http://www.omg.org/technology/documents/profile_catalog.htm)

Ambas direcciones anteriores OMG, las he refundido en [www.omg.org/spec](http://www.omg.org/spec)

Si desea profundizar en UML le recomendamos que visite el sitio [www.omg.org](http://www.omg.org) y consulte la bibliografía que se incluye en este libro. Por otra parte, los modelos UML tienen al menos dos dimensiones: una dimensión gráfica para visualizar el modelo usando diagramas e íconos (notaciones), y otra dimensión con texto, que describe las especificaciones de distintos elementos de modelado.

La especificación o lenguaje OCL define un lenguaje para escritura de restricciones y expresiones de los elementos del modelo.

## 13.9 Historia de UML

En la primera mitad de la década de 1990, los lenguajes de modelado que imperaban eran: OMT-2 (Object Modeling Technique), creado por James Rumbaugh; OOSE (Object Oriented Software Enginee-

ring), creado por Ivan Jacobson y Booch<sup>1</sup>, cuyo creador era Grady Booch. Existían otros métodos que fueron muy utilizados: Coad/Yourdon y Fusion, entre otros. Fusion de Coleman constituyó un primer intento de unificación de métodos, pero al no contar con el apoyo de los restantes métodos, pese a ser apoyado por Hewlett-Packard en algún momento, no llegó a triunfar.

Grady Booch, propietario de Rational llamó a James Rumbaugh y formaron equipo en una nueva Rational Corporation con el objetivo de fusionar sus dos métodos. En octubre de 1994, Grady Booch y James Rumbaugh comenzaron a trabajar en la unificación de ambos métodos produciendo una versión en borrador llamada 0.8 y de nombre “*Unified Method*”.<sup>2</sup> Posteriormente, a partir de 1995, Jacobson se unió al tandem y formaron un equipo que se llegó a conocer como “los tres amigos”.

El primer fruto de su trabajo colectivo se lanzó en enero de 1997 y fue presentado como versión 1.0 de UML. La gran ventaja de UML es que fue recogiendo aportaciones de los grandes gurús de objetos: David Hasel con sus diagramas de estado; partes de la notación de Fusion, el criterio de responsabilidad-collaboración y los diagramas de Rebeca Wirfs-Brock, y el trabajo de patrones y documentación de Gamma-Helm-Johnson-Ulissides.

En 1997, OMG aceptó UML como estándar (versión 1.1) y nació el primer lenguaje de modelado visual orientado a objetos como estándar abierto de la industria. Desde entonces han desaparecido, prácticamente, todas las demás metodologías y UML se ha convertido en el estándar de la industria del software y de muchas otras industrias que requieren el uso de modelos. En 1998, OMG lanzó dos revisiones más, 1.2 y 1.3. En el año 2000 se presentó UML 1.4 con la importante aportación de la semántica de acción, que describe el comportamiento de un conjunto de acciones permitidas que pueden implementarse mediante lenguajes de acción. La versión 1.5 siguió a las ya citadas.

En 2005 se lanzó la versión 2.0 con su especificación completa y se convirtió en un lenguaje de modelado muy maduro. **UML 2** ha incorporado numerosas mejoras a **UML 1**. Los cambios más importantes se han realizado en el *metamodelo* (generador de modelos), conservando los principios fundamentales y evolutivos de las últimas versiones. Los diseñadores de UML 2.0 (ya no solo los tres creadores originales, sino una inmensa pléyade que ha contribuido con infinitas aportaciones) han tenido mucho cuidado en asegurar que fuera totalmente compatible con las versiones anteriores para que los usuarios de estas no tuvieran problemas de adaptación.

UML ha seguido evolucionando: versión 2.1 en abril de 2006, 2.2 en febrero de 2009, en mayo de 2010, la versión 2.3 y la versión 2.4 en agosto de 2011. La última versión presentada ha sido **UML 2.5** en noviembre de 2012 y con carácter de versión beta, aunque se espera la aprobación de la versión a lo largo del año 2013.

En la página oficial de OMG [www.omg.org](http://www.omg.org) encontrará la información más actualizada, así como especificaciones y manuales de todas las versiones disponibles.

#### Línea de tiempo (cronología, *timeline*)

- 1997 – UML 1.1
- 1997 – OMG UML 1.1 (a partir de esta fecha todas las versiones son OMG UML)
- 1998 – UML 1.2
- 1999 – UML 1.3
- 2001 – UML 1.4.2 (norma ISO/IEC 19501)
- 2003 – UML 1.5
- 2005 – UML 2.0
- 2006 – UML 2.1
- 2007 – UML 2.1.1/2.1.2
- 2009 – UML 2.2
- 2010 – UML 2.3
- 2011 – UML 2.4.1
- 2012 – UML 2.5 (adoptada, borrador)
- 2013 – UML 2.5 (prevista aprobación)

<sup>1</sup> Los libros originales de OMT y Booch'93 los tradujo un equipo de profesores universitarios dirigidos por el profesor Joyanes.

<sup>2</sup> Todavía conservo el borrador de una versión, en papel, que fue presentada por James Rumbaugh, entre otros lugares, en un hotel en Madrid, invitado por su editorial Prentice-Hall (nota de Luis Joyanes).



## Resumen

El *tipo abstracto de datos* se implementa a través de *clases*. Una *clase* es un conjunto de objetos que constituyen instancias de ella, cada una de las cuales tiene la misma estructura y comportamiento. Una clase tiene un nombre, una colección de operaciones para manipular sus instancias y una representación. Las operaciones que manipulan las instancias de una clase se llaman *métodos*. El estado o representación de una instancia se almacena en variables de instancia. Estos métodos se invocan mediante el envío de *mensajes* a instancias. El envío de mensajes a objetos (instancias) es similar a la llamada a procedimientos en lenguajes de programación tradicionales.

El *polimorfismo* permite desarrollar sistemas en los que objetos diferentes pueden responder de modo distinto al mismo mensaje.

La programación orientada a objetos incorpora estos seis componentes importantes:

- Objetos.
- Clases.
- Métodos.
- Mensajes.
- Herencia.
- Polimorfismo.

Un objeto se compone de datos y funciones que operan sobre esos objetos.

La técnica de situar datos dentro de objetos, de modo que no se puede acceder directamente a los datos, se llama *ocultación de la información*.

Una *clase* es una descripción de un conjunto de objetos. Una *instancia* es una variable de tipo objeto y *un objeto es una instancia de una clase*.

La *herencia* es la propiedad que permite a un objeto pasar sus propiedades a otro objeto, o dicho de otro modo, un objeto puede heredar de otro objeto.

Los objetos se comunican entre sí pasando *mensajes*.

La clase padre o ascendiente se denomina *clase base* y las clases descendientes, *clases derivadas*.

La *reutilización de software* es una de las propiedades más importantes que presenta la programación orientada a objetos.

El *polimorfismo* es la propiedad por la cual un mismo mensaje puede actuar de diferente modo cuando actúa sobre objetos distintos ligados por la propiedad de la herencia.

**UML**, Lenguaje Unificado de Modelado, es utilizado en el desarrollo de sistemas.

UML tiene dos categorías importantes de diagramas: *diagramas estructurales* (de estructura) y de *comportamiento*. Las construcciones contenidas en cada uno de los diagramas UML se describen al lado de cada diagrama:

- Diagrama de actividad – actividades.
- Diagrama de clases – clasificadores estructurados.
- Diagrama de comunicación – interacciones.
- Diagrama de componentes – clasificadores estructurados.
- Diagrama compuesto de estructura – clasificadores estructurados.
- Diagrama de despliegue – despliegues.
- Diagrama de interacción global – interacciones.
- Diagrama de objetos – clasificación.
- Diagrama de paquetes – paquetes.
- Diagrama de perfiles – paquetes.
- Diagrama de máquina de estados – máquina de estados.
- Diagrama de secuencias – interacciones.
- Diagramas de tiempo – interacciones.
- Diagramas de casos de uso – casos de uso.

## Ejercicios

13.1 Describir y justificar los objetos que obtiene de cada uno de estos casos:

- a) Los habitantes de Europa y sus direcciones de correo.
- b) Los clientes de un banco que tienen una caja fuerte alquilada.
- c) Las direcciones de correo electrónico de una universidad.

d) Los empleados de una empresa y sus claves de acceso a sistemas de seguridad.

13.2 ¿Cuáles serían los objetos que han de considerarse en los siguientes sistemas?

- a) Un programa para maquetar una revista.
- b) Un contestador telefónico.
- c) Un sistema de control de ascensores.
- d) Un sistema de suscripción a una revista.

**13.3** Definir los siguientes términos:

- |                                      |                                   |
|--------------------------------------|-----------------------------------|
| <i>a)</i> clase.                     | <i>g)</i> miembro dato.           |
| <i>b)</i> objeto.                    | <i>h)</i> constructor.            |
| <i>c)</i> sección de declaración.    | <i>i)</i> instancia de una clase. |
| <i>d)</i> sección de implementación. | <i>j)</i> métodos o servicios.    |
| <i>e)</i> variable de instancia.     | <i>k)</i> sobrecarga.             |
| <i>f)</i> función miembro.           | <i>l)</i> interfaz.               |

**13.4** Deducir los objetos necesarios para diseñar un programa de computadora que permita jugar a diferentes juegos de cartas.

**13.5** Determinar los atributos y operaciones que pueden ser de interés para los siguientes objetos, partiendo de la base de que van a ser elementos de un almacén de regalos: un libro, un disco, una grabadora de video, una cinta de video, un televisor, un radio, un tostador de pan, un equipo de sonido, una calculadora y un teléfono celular (móvil).

**13.6** Crear una clase que describa un rectángulo que se pueda visualizar en la pantalla de la computadora, cambiar de tamaño, modificar su color de fondo y los colores de los lados.

**13.7** Representar una clase ascensor (elevador) que tenga las funciones usuales de subir, bajar, parar entre niveles

(pisos), alarma, sobrecarga y en cada nivel botones de llamada para subir o bajar.

**13.8** Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo *Figura*.

**13.9** Construir una clase *Persona* con las funciones miembro y atributos que crea oportunos.

**13.10** Construir una clase llamada *Luz* que simule una luz de semáforo. El atributo *color* de la clase debe cambiar de *Verde* *Amarillo* y a *Rojo* y de nuevo regresar a *Verde* mediante la función *Cambio*. Cuando se crea un objeto *Luz*, su color inicial será *Rojo*.

**13.11** Construir una definición de clase que se pueda utilizar para representar a un empleado de una compañía. Cada empleado se define por un número entero *ID*, un salario y el número máximo de horas de trabajo por semana. Los servicios que debe proporcionar la clase, al menos deben permitir introducir datos de un nuevo empleado, visualizar los datos existentes de un nuevo empleado y capacidad para procesar las operaciones necesarias para dar de alta y de baja en la seguridad social y en los seguros que tenga contratados la compañía.



# Diseño de clases y objetos: representaciones gráficas en UML

## Contenido

- 14.1 Diseño y representación gráfica de objetos en UML
- 14.2 Diseño y representación gráfica de clases en UML
- 14.3 Declaración de objetos de clases
  - › Resumen
  - › Ejercicios

## Introducción

Hasta ahora se ha estudiado el concepto de estructuras, con lo que se ha visto un medio para agrupar datos. También se han examinado funciones que sirven para realizar acciones determinadas a las que se les asigna un nombre. En este capítulo se tratarán las clases, un nuevo tipo de dato cuyas variables serán objetos. Una **clase** es un tipo de dato que contiene código (funciones) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En este capítulo aprenderá a crear (definir y especificar) y utilizar clases individuales y en el capítulo 15 verá cómo definir y utilizar jerarquías y otras relaciones entre clases.

Los diagramas de clases son uno de los tipos de diagramas más fundamentales en UML. Se utilizan para capturar las relaciones estáticas de su software; entre otras palabras, cómo poner o relacionar cosas juntas. Cuando se escribe software se está constantemente tomando decisiones de diseño: qué clases hacen referencia a otras clases, qué clases “poseen” alguna otra clase, etc. Los diagramas de clase proporcionan un medio de capturar la estructura física de un sistema.

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard, quien al intentar escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento. Descubrió que los elementos del entorno que trataba de modelar (barcos, mareas y línea de la costa de los fiordos) y las acciones que cada elemento podía ejecutar, constituyan unas relaciones que eran más fáciles de manejar.



## Conceptos clave

- › Abstracta
- › Clase compuesta
- › Comunicación entre objetos
- › Constructor
- › Destructor
- › Encapsulamiento
- › Especificador de acceso
- › Función miembro
- › Instancia
- › Mensaje
- › Método
- › Miembro dato
- › Objeto
- › Ocultación de la información
- › Privada
- › Protegida
- › Pública

Las tecnologías orientadas a objetos han evolucionado mucho pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas son los elementos clave sobre los que se articula la orientación a objetos.

## 14.1 Diseño y representación gráfica de objetos en UML

Un *objeto* es la *instancia* de una clase. El señor Mackoy es un objeto de la clase *Persona*. Un objeto es simplemente una colección de información relacionada y funcionalidad. Un objeto puede ser algo que tenga una manifestación o correspondencia en el mundo real (como un objeto empleado), algo que tenga algún significado virtual (como una ventana en la pantalla) o alguna abstracción adecuada dentro de un programa (una lista de trabajos a realizar, por ejemplo). Un objeto es una entidad atómica formada por la unión del estado y del comportamiento. Proporciona una relación de *encapsulamiento* que asegura una fuerte cohesión interna y un débil acoplamiento con el exterior. Un objeto revela su rol verdadero y la responsabilidad cuando al enviar mensajes se convierte en parte de un escenario de comunicaciones. Un objeto contiene su propio estado interno y un comportamiento accesible a otros objetos.

El mundo en que vivimos se compone de objetos tangibles de todo tipo. El tamaño de estos objetos es variable, pequeños como un grano de arena a grandes como una montaña o un buque de recreo. Nuestra idea intuitiva de objeto viene directamente relacionada con el concepto de masa, es decir la propiedad que caracteriza la cantidad de materia dentro de un determinado cuerpo. Sin embargo, es posible definir otros objetos que no tengan ninguna masa, como una cuenta corriente, una póliza de seguros, una ecuación matemática o los datos personales de un alumno de una universidad. Estos objetos corresponden a conceptos, en lugar de a entidades físicas.

Se puede ir más lejos y extender la idea de objeto haciendo que pertenezcan a “mundos virtuales” (asociados con la red internet, por ejemplo) con objeto de crear comunidades de personas que no estén localizadas en la misma área geográfica. Objetos de software definen una representación *abstracta* de las entidades del mundo real con el fin de controlarlo o simularlo. Los objetos de software pueden ir desde listas enlazadas, árboles hasta archivos completos o interfaces gráficas de usuario.

En síntesis, un objeto se compone de datos que describen el objeto y las operaciones que se pueden ejecutar sobre ese objeto. La información almacenada en un objeto empleado, por ejemplo, puede ser información de identificación (nombre, dirección, edad, titulación), información laboral (título del trabajo, salario, antigüedad), etc. Las operaciones realizadas pueden incluir la creación del sueldo o la promoción de un empleado.

Al igual que los objetos del mundo real que nacen, viven y mueren, los objetos del mundo del software tienen una representación similar conocida como su ciclo de vida.

### Nota

Un objeto es algo que encapsula información y comportamiento. Es un término que representa una cosa concreta o del mundo real.

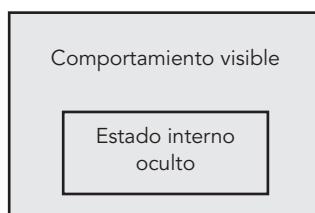


Figura 14.1 Objeto: estado y comportamiento.

### Ejemplos de objetos

- Vuelo 6520 de Iberia (Santo Domingo-Madrid con escala en San Juan de Puerto Rico).
- Casa núm. 31 de la Avenida Reforma en León (Guanajuato).
- Flor roja en el balcón de la terraza del profesor Mackoy.

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch<sup>1</sup> define un *objeto* como “algo que tiene un estado, un comportamiento y una identidad”. Supongamos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/apagada* (“on/off”), y trabajar a su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar en el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin y Odell definen un objeto como “cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos”. Para realizar esa actividad se añaden a cada objeto de la clase los propios datos y asociados con sus propias funciones miembro que pertenecen a la clase.

Cualquier programa orientado a objetos puede manejar muchos objetos. Por ejemplo, un programa que maneja el inventario de un almacén de ventas al por menor, utiliza un objeto de cada producto manipulado en el almacén. El programa manipula los mismos datos de cada objeto, incluyendo el número de producto, descripción, precio, número de artículos en existencia y el momento de nuevos pedidos.

Cada objeto conoce también cómo ejecutar acciones con sus propios datos. El objeto producto del programa de inventario, por ejemplo, conoce cómo crearse a sí mismo y establecer los valores iniciales de todos sus datos, cómo modificarlos y cómo evaluar si hay artículos suficientes en existencia para cumplir una petición de compra. En esencia, la cosa más importante de un objeto es reconocer que consta de datos y las acciones que puede ejecutar.

Un objeto de un programa de computadora no es algo que se pueda tocar. Cuando un programa se ejecuta, la mayoría existe en memoria principal. Los objetos se crean por un programa para su uso mientras el programa se está ejecutando. A menos que se guarden los datos de un objeto en un disco, el objeto se pierde cuando el programa termina (este objeto se llama *transitorio* para diferenciarlo del objeto *permanente* que se mantiene después de la terminación del programa).

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón “enviar”. El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el periodo de garantía puede anularla.

### Representación gráfica en UML

Un objeto es una instancia de una clase. Por ejemplo, se pueden tener varias instancias de una clase llamada *Carro*: Un carro rojo de dos puertas, un carro azul de cuatro puertas y un carro todo terreno verde de cinco puertas. Cada instancia de *Carro* es un objeto al que se puede dar un nombre o dejarlo anónimo

<sup>1</sup> Booch, Grady. *Ánalisis y diseño orientado a objetos con aplicaciones*. Madrid: Díaz de Santos/Addison-Wesley, 1995.

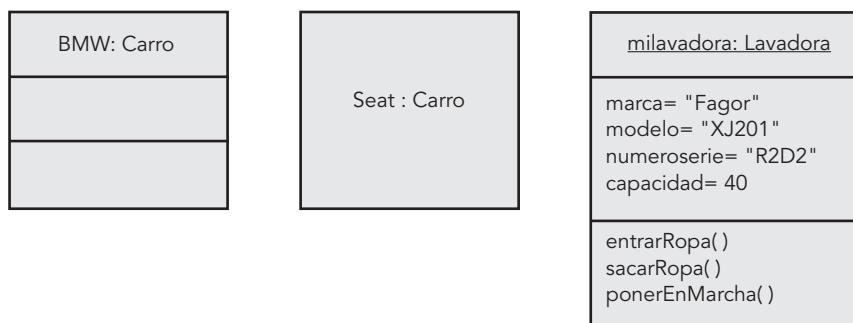


Figura 14.2 Diferentes objetos: Seat y BMW de la clase Carro, milavadora de la clase Lavadora.

y se representa en los diagramas de objetos. Normalmente se muestra el nombre del objeto seguido por el símbolo dos puntos y el nombre de la clase o su tipo. Tanto el nombre del objeto como el nombre de la clase se subrayan.

En UML, un objeto se representa por un rectángulo en cuyo interior se escribe el nombre del objeto subrayado. El diagrama de representación tiene tres modelos (figura 14.2).

El diagrama de la figura 14.3 representa diferentes clientes de un banco y las cuentas asociadas con cada uno de estos clientes. Las líneas que conectan estos objetos representan los enlaces que existen entre un cliente determinado y sus cuentas. El diagrama muestra también un rectángulo con un doblete en la esquina superior derecha; este diagrama representa un comentario (una nota, un texto de información libre concebida con propósito de clarificación de la figura y de facilitar la comprensión del diagrama); las líneas punteadas implementan la conexión de cualquier elemento del modelo a una nota descriptiva o de comentario.

A veces es difícil encontrar un nombre para cada objeto, por esta razón se suele utilizar con mucha mayor frecuencia un nombre genérico en lugar de un nombre individual. Esta característica permite nombrar los objetos con términos genéricos y evitar abreviaturas de nombres o letras, como se hacía antiguamente, a, b o c.

El diagrama de la figura 14.4 muestra representaciones de objetos estudiantes y profesores. La ausencia de cualquier texto precedente delante de los dos puntos significa que estamos hablando de tipos de objetos genéricos o anónimos de tipos Estudiante y Profesor.

## Características de los objetos

Todos los objetos tienen tres características o propiedades fundamentales que sirven para definir a un objeto de modo inequívoco: *un estado, un comportamiento y una identidad*.

### Objeto = Estado + Comportamiento + Identidad

Un objeto debe tener todas o alguna de las propiedades anteriores. Un objeto sin el estado o sin el comportamiento puede existir, pero un objeto siempre tiene una identidad.

### A recordar

Un objeto es una entidad que tiene estado, comportamiento e identidad. La estructura y comportamiento de objetos similares se definen en sus clases comunes. Los términos *instancia* y *objeto* son intercambiables [Booch *et al.*, 2007].

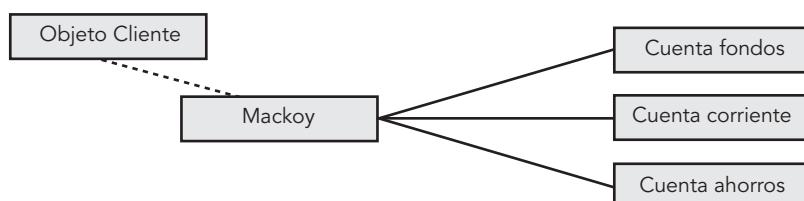


Figura 14.3 Enlaces entre objetos de las clases Cliente y Cuenta.

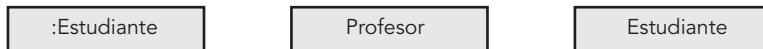


Figura 14.4 Diferentes representaciones de objetos (Ejemplo: Profesor y Estudiante).

## Estado

El estado agrupa los valores de todos los **atributos** de un objeto en un momento dado, en donde un atributo es una pieza de información que califica el objeto contenedor. Cada atributo puede tomar un valor determinado en un dominio de definición dado. *El estado de un objeto, en un momento dado, se corresponde con una selección determinada de valores a partir de valores posibles de los diversos atributos.* En esencia, un atributo es una propiedad o característica de una clase y describe un rango de valores que la propiedad podrá contener en los objetos de la clase. Una clase podrá contener varios atributos o ninguno.

### A recordar

El estado de un objeto abarca todas las propiedades (normalmente estáticas) del objeto más los valores actuales (normalmente dinámicos) de cada una de estas propiedades [Booch *et al.*, 2007].

Los atributos de una clase son las partes de información que representan el estado de un objeto y constituyen las propiedades de una clase. Así los detalles de la clase *Carro* son atributos: color, número de puertas, potencia, etc. Los atributos pueden ser tipos primitivos simples (enteros, reales, ...), compuestos (cadena, complejo, ...) o relaciones a otros objetos complejos.

Una clase puede tener cero o más atributos. Un nombre de un atributo puede ser cualquier conjunto de caracteres, pero dos atributos de la misma clase no pueden tener el mismo nombre. Un atributo se puede mostrar utilizando dos notaciones diferentes: *en línea* o *en relaciones entre clases*. Además la notación está disponible para representar otras propiedades, como multiplicidad, unicidad u ordenación.

Por convenio, el nombre de un atributo puede ser de una palabra o varias palabras unidas. Si el nombre es de una palabra se escribe en minúsculas y si es más de una palabra, las palabras se unen y cada palabra, excepto la primera, comienza con una letra mayúscula. La lista de atributos se sitúa en el compartimento o banda debajo del compartimento que contiene al nombre de la clase.

### Nombre de objetos

|            |          |                                       |                             |
|------------|----------|---------------------------------------|-----------------------------|
| miLavadora | :        | Lavadora                              | <i>instancia con nombre</i> |
| :          | Lavadora | <i>instancia sin nombre (anónima)</i> |                             |

UML proporciona la opción de indicar información adicional para los atributos. En la notación de la clase, se pueden especificar un tipo para cada valor del atributo. Se pueden incluir tipos como cadena (string), número de coma flotante, entero o boolean (y otros tipos enumerados). Para indicar un tipo, se utilizan dos puntos (:) para separar el nombre del atributo del tipo. Se puede indicar también un valor por defecto (en forma predeterminada) para un atributo.

Un atributo se representa con una sola palabra en minúsculas; por otro lado, si el nombre contiene más de una palabra, cada palabra será unida a la anterior e iniciará con una letra mayúscula, a excepción de la primera palabra que comenzará en minúscula. La lista de atributos se inicia en la segunda banda del icono de la clase. Todo objeto de la clase tiene un valor específico en cada atributo. El nombre de un objeto se inicia con una letra minúscula y está precedido de dos puntos que a su vez están precedidos del nombre de la clase, y todo el nombre está subrayado.

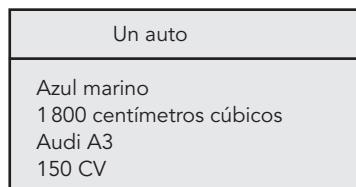
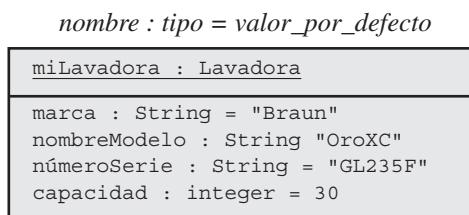


Figura 14.5 Un objeto con sus atributos.



**Figura 14.6** Atributos con valores por defecto (en forma predeterminada).

El nombre miComputadora : Computadora es una instancia con nombre (un objeto), pero también es posible tener un objeto o instancia anónima y se representa como :Computadora .

### Ejemplos de objetos

- El profesor Mariano.
- La ciudad de Pachuca.
- La casa en Calle Real 25, Pachuca (Hidalgo).
- El carro amarillo que está estacionado en la calle al lado de mi ventana.

Cada objeto encapsula una información y un comportamiento. El objeto vuelo IB 6520, por ejemplo. La fecha de salida es el 16 de agosto de 2009, la hora de salida es 2.30 horas, el número de vuelo es el 6520, la compañía de aviación es Iberia, la ciudad de partida es Santo Domingo y la ciudad destino es Madrid con breve escala en San Juan de Puerto Rico. El objeto vuelo también tiene un comportamiento. Se conocen los procedimientos de cómo añadir un pasajero al vuelo, quitar un pasajero del vuelo o determinar cuando el vuelo está lleno; es decir, añadir, quitar, estáLleno. Aunque los valores de los atributos cambiarán con el tiempo (el vuelo IB 525 tendrá una fecha de salida, el día siguiente, 17 de agosto), los atributos por sí mismo nunca cambiarán. El vuelo 6520 siempre tendrá una fecha de salida, una hora de salida y una ciudad de salida; es decir, sus atributos son fijos.

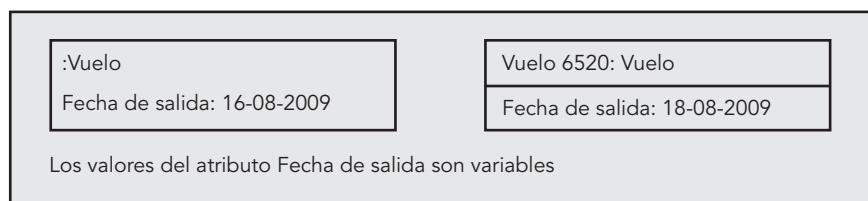
En UML se pueden representar los tipos y valores de los atributos. Para indicar un tipo, utilice dos puntos (:) que separan el nombre del atributo de su tipo.

### Múltiples instancias de un objeto

En un diagrama de clases se pueden representar múltiples instancias de un objeto mediante iconos múltiples. Por ejemplo, si se necesita representar una lista de vuelos de Iberia para su representación en un diagrama de clases u objetos, en lugar de mostrar cada vuelo como un objeto independiente, se puede utilizar un ícono con múltiples instancias para mostrar la lista de vuelos. La notación UML para representar instancias múltiples se representa en la figura 14.8.

#### Nota

Los atributos son los trozos de información contenidos en un objeto. Los valores de los atributos pueden cambiar durante la vida del objeto.



**Figura 14.7** Un objeto con atributos, sus tipos así como sus valores predeterminados.



Figura 14.8 Instancias múltiples del objeto *Vuelo*.

## Evolución de un objeto

El estado de un objeto evoluciona con el tiempo. Por ejemplo, el objeto *Auto* tiene los atributos: Marca, Color, Modelo, Capacidad del depósito o tanque de la gasolina, Potencia (en caballos de potencia). Si el auto comienza un viaje, normalmente se llenará el depósito (el atributo Capacidad puede tomar, por ejemplo, el valor 50 “litros” o 12 galones), el color del auto, en principio no cambiará (azul cielo), la potencia tampoco cambiará (150 Caballos, HP). Es decir, hay atributos cuyo valor va variando, como la capacidad (ya que a medida que avance, disminuirá la cantidad que contiene el depósito), pero habrá otros que normalmente no cambiarán como el color y la potencia del auto, o la marca y el modelo, inclusive el país en donde se ha construido.

El diagrama de la figura 14.9 representa la evolución de la clase *Auto* con un comentario explicativo de la disminución de la gasolina del depósito debido a los kilómetros recorridos.

### A recordar

Los objetos de software (objetos) encapsulan una parte del conocimiento del mundo en el que ellos evolucionan.

## Comportamiento

El comportamiento es el conjunto de capacidades y aptitudes de un objeto y describe las acciones y reacciones de ese objeto. Cada componente del comportamiento individual de un objeto se denomina **operación**. Una **operación** es algo que la clase puede realizar o que se puede hacer a una clase. Las operaciones de un objeto se disparan (activan) como resultado de un estímulo externo representado en la forma de un mensaje enviado a otro objeto.

Las operaciones son las características de las clases que especifican el modo de invocar un comportamiento específico. Una operación de una clase describe *qué* hace una clase pero no necesariamente

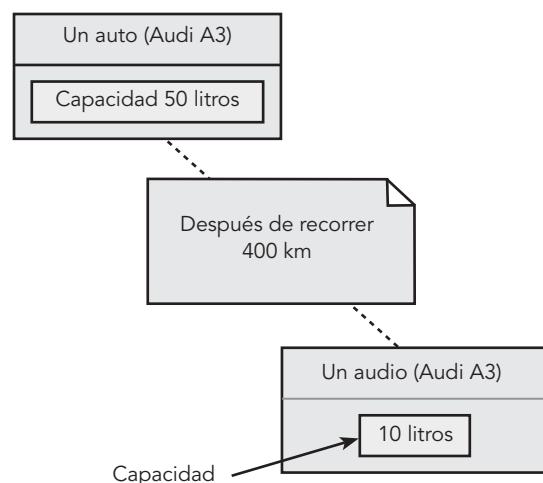


Figura 14.9 Evolución de una clase.

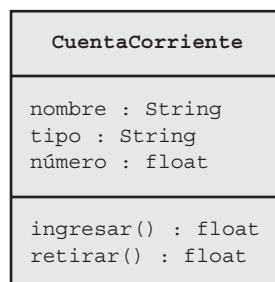


Figura 14.10 Clase CuentaCorriente.

cómo lo hace. Por ejemplo, una clase puede ofrecer una operación para dibujar un rectángulo en la pantalla, o bien contar el número de elementos seleccionados de una lista. UML hace una diferencia clara entre la especificación de cómo invocar un comportamiento (una operación) y la implementación real de ese comportamiento (*método o función*).

Las operaciones en UML se especifican en un diagrama de clase con una estructura compuesta por nombre, un par de paréntesis (vacíos o con la lista de parámetros que necesita la operación) y un tipo de retorno.

### Sintaxis operaciones

1. *nombre (parámetros) : tipo\_retorno*
2. *nombre( )*

Al igual que sucede con los nombres de los atributos, el nombre de una operación se pone en minúsculas si es una palabra; en el caso de que el nombre conste de más de una palabra se unen ambas y comienzan todas las palabras reservadas después de la primera con una letra mayúscula. Así en la clase Lavadora, por ejemplo puede tener las siguientes operaciones:

```

aceptarRopa(r: String)
aceptarDetergente(d: String)
darBotónPrender(): boolean
darBotónApagar(): boolean

```

### A recordar

Comportamiento es el modo en que un objeto actúa y reacciona, en términos de sus cambios de estado y paso de mensajes [Booch *et al.*, 2007].

En la figura 14.11, se disparan una serie de interacciones dependiendo del contenido del mensaje.

Las interacciones entre objetos se representan utilizando diagramas en los que los objetos que interactúan se unen a los restantes vía líneas continuas denominadas **enlaces**. La existencia de un enlace

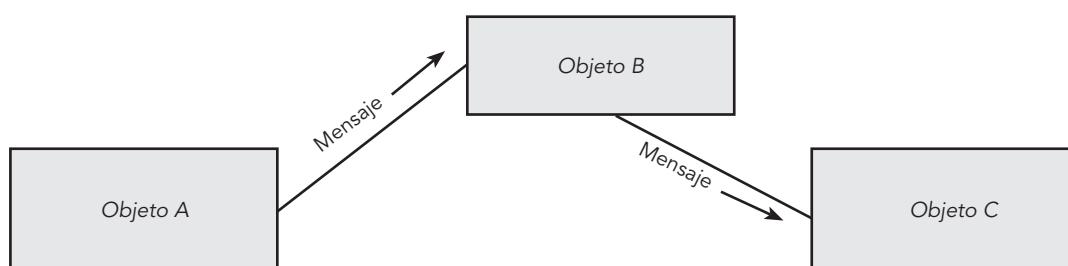


Figura 14.11 Mensajes entre objetos.



Figura 14.12 Envío de mensajes.

indica que un objeto conoce o ve otro objeto. Los mensajes navegan junto a los enlaces, normalmente en ambas direcciones.

### Ejemplo

El objeto A envía un mensaje `Almorzar` al objeto B y el objeto B envía un mensaje `EcharLaSiesta` al objeto C. Las operaciones que se realizan mediante la comunicación de mensajes presuponen que el objeto B tiene la capacidad de almorzar y que el objeto C es capaz de irse a echar la siesta (figura 14.12).

El estado y el comportamiento están enlazados; realmente, el comportamiento en un momento dado depende del estado actual y el estado puede ser modificado por el comportamiento. Solo es posible aterrizar un avión si está volando, de modo que el comportamiento `aterrizar` solo es válido si la información `enVuelo` es verdadera. Después de aterrizar la información `enVuelo` se vuelve `falsa` y la operación `aterrizar` ya no tiene sentido; en este caso tendría sentido, `despegar` ya que la información del atributo `enVuelo` es falsa, cuando el avión está en tierra pendiente de despegar. El diagrama de colaboración de clases ilustra la conexión entre el estado y el comportamiento de los objetos de la clase. En el caso del objeto `Vuelo 6520`, las operaciones del objeto `vuelo` pueden ser añadir o quitar un pasajero y verificar si el vuelo está lleno.

### Definición

El comportamiento de un objeto es el conjunto de sus operaciones.

### Regla

De igual modo que el nombre de un atributo, el nombre de una operación se escribe en minúscula si consta de una sola palabra. En caso de constar de más de una palabra, se unen y se inician todas con mayúsculas, excepto la primera. La lista de operaciones se inicia en la tercera banda del ícono de la clase y justo debajo de la línea que separa las operaciones de los atributos.

### Identidad

La **identidad** es la propiedad que diferencia un objeto de otro objeto similar. En esencia, la identidad de un objeto caracteriza su propia existencia. La identidad hace posible distinguir cualquier objeto sin ambigüedad, e independientemente de su estado. Esto permite, entre otras cosas, la diferenciación de dos objetos que tengan los atributos idénticos.

La identidad no se representa de manera específica en la fase de modelado de un problema. Cada objeto tiene en forma implícita una identidad. Durante la fase de implementación, la identidad se crea por lo general utilizando un identificador que viene naturalmente del dominio del problema. Nuestros autos tienen un número de placa, nuestros teléfonos celulares tienen un número a donde nos pueden llamar y nosotros mismos podemos ser identificados por el número del pasaporte o el número de la seguridad social. El tipo de identificador, denominado también “clave natural”, se puede añadir a los estados del objeto a fin de diferenciarlos. Sin embargo, solo es un artefacto de implementación, de modo que el concepto de identidad permanece independiente del concepto de estado.

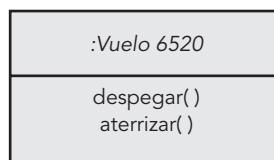


Figura 14.13 Objeto Vuelo 6520.

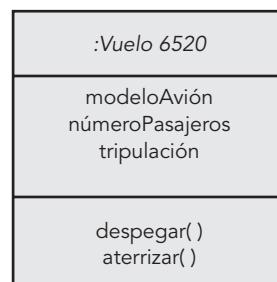


Figura 14.14 Representación gráfica de una clase con estado y comportamiento.

## Los mensajes

El mensaje es el fundamento de una relación de comunicación que enlaza dinámicamente los objetos que fueron separados en el proceso de descomposición de un módulo. En la práctica, un **mensaje** es una *comunicación entre objetos* en los que un objeto (el cliente) solicita al otro objeto (el proveedor o servidor) hacer o ejecutar alguna acción.



Figura 14.15 Comunicación entre objetos.

También se puede mostrar en UML mediante un diagrama de secuencia (figura 14.16).

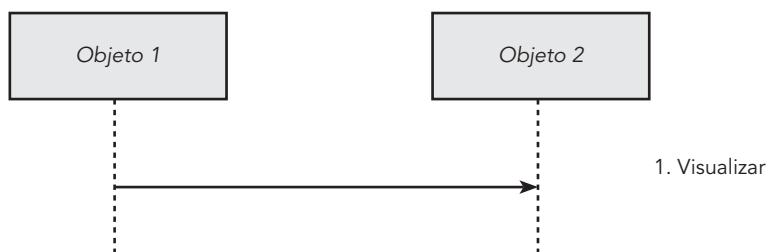


Figura 14.16 Diagrama de secuencia.

El mensaje puede ser reflexivo: un objeto se envía un mensaje a sí mismo.

La noción de un mensaje es un concepto abstracto que se puede implementar de varias formas como una llamada a una función, un evento o suceso directo, una interrupción, una búsqueda dinámica, etc.

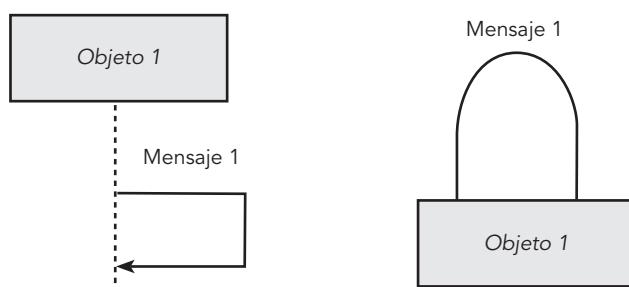


Figura 14.17 Mensaje reflexivo.

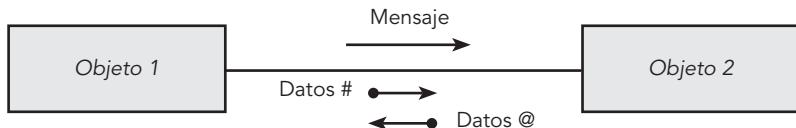


Figura 14.18 Representación gráfica de un mensaje con flujos de control y de datos.

En realidad un mensaje combina flujos de control y flujos de datos en una única entidad. Las flechas simples indican el flujo de control y las flechas con un pequeño círculo en el origen son flujos de datos.

### Tipos de mensajes

Existen diferentes categorías de mensajes:

- *Constructores* (crean objetos).
- *Destructores* (destruyen objetos).
- *Selectores* (devuelven todo o parte del estado de un objeto).
- *Modificadores* (cambian todo o parte del estado de un objeto).
- *Iteradores* (visitan el estado de un objeto o el contenido de una estructura de datos que incluye varios objetos).

### Ejemplo

Esquema de una clase con métodos o funciones correspondientes a los tipos de mensajes.

```
clase VueloAvión
 Público:
 // constructores
 ...
 // destructores
 ...
 // selectores
 ...
 // modificadores
 ...
 // iteradores
 ...
 privado:
 // atributos del vuelo
fin_clase
```

### Responsabilidad y restricciones

El ícono de la clase permite especificar otro tipo de información sobre la misma: la responsabilidad. La *responsabilidad* es un contrato o una obligación de una clase; es una descripción de lo que ha de hacer la clase. Al crear una clase se está expresando que todos los objetos de esa clase tienen el mismo tipo de estado y el mismo tipo de comportamiento. A un nivel más abstracto, estos atributos y operaciones son simplemente las características por medio de las cuales se llevan a cabo las responsabilidades de la clase [Booch06]. Así una clase *Lavadora* tiene la responsabilidad de: “recibir ropa sucia como entrada y producir ropa limpia como salida”. Una clase *Pared* de una casa es responsable de conocer la altura, anchura, grosor y color de la pared; una clase *SensorDeTemperatura* es responsable de medir la temperatura del carro y disparar la alarma (el piloto rojo de peligro) si esta temperatura alcanza un valor determinado de riesgo.

Las responsabilidades de la clase se pueden escribir en la zona inferior, debajo del área que contiene la lista de operaciones. En el ícono de la clase se debe incluir la suficiente información para describir una clase de un modo no ambiguo. La descripción de las responsabilidades de la clase es un modo informal de eliminar su ambigüedad.

Una *restricción* es un modo más formal de eliminar la ambigüedad. La regla que describe la restricción es encerrar entre llaves el texto con la restricción especificada y situarla cerca del ícono de la clase. Por ejemplo, se puede escribir `{temperatura = 35 o 39 o 42}` o bien `{capacidad = 20 o 30 o 40 kg}`.

UML permite definir restricciones para hacer las definiciones más explícitas. El método que emplea es un lenguaje completo denominado OCL (Object Constraint Language), Lenguaje de restricción de objetos, que tiene sus propias reglas, términos y operadores (en el sitio oficial de OMG puede ver la documentación completa de la última versión de OCL).

## 14.2 Diseño y representación gráfica de clases en UML

En términos prácticos, una *clase* es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como “un conjunto de objetos que comparten una estructura y comportamiento comunes”.

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios*, *métodos* o *funciones miembro*. El término *función miembro* se utiliza, específicamente, en C++. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos* o *variables*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable* se suele utilizar en programas orientados a objetos.

El mundo real se compone de un gran número de objetos que interactúan entre sí. Estos objetos, en numerosas ocasiones, resultan muy complejos para poder ser entendidos en su totalidad. Por esta circunstancia se suelen agrupar juntos elementos similares y con características comunes en función de las propiedades más sobresalientes e ignorando aquellas otras propiedades no tan relevantes. Este es el proceso de abstracción ya citado anteriormente.

Este proceso de abstracción suele comenzar con la identificación de características comunes de un conjunto de elementos y prosigue con la descripción concisa de estas características en lo que convencionalmente se ha venido en llamar **clase**.

Una clase describe el dominio de definición de un conjunto de objetos. Cada objeto pertenece a una clase. Las características generales están contenidas dentro de la clase y las características especializadas están contenidas en los objetos. Los objetos software se construyen a partir de las clases vía un proceso conocido como **instanciación**. De este modo un objeto es una **instancia** (ejemplar o caso) de una clase.

Así pues, una clase define la estructura y el comportamiento (datos y código) que serán compartidos por un conjunto de objetos. Cada objeto de una clase dada contiene la estructura (el estado) y el comportamiento definido por la clase y los objetos, como se ha definido anteriormente, suelen conocerse por instancias de una clase. Por consiguiente, una clase es una construcción lógica; un objeto tiene realidad física.

### A recordar

Una clase es una entidad que encapsula información y comportamiento.

Cuando se crea una clase, se especificará el código y los datos que constituyen esa clase. De modo general, estos elementos se llaman *miembros* de la clase. De modo específico, los datos definidos en la clase se denominan *variables miembro* o *variables de instancia*. El código que opera sobre los datos se conocen como *métodos miembro* o simplemente *métodos*. En la mayoría de las clases, las variables de instancia son manipuladas o accedidas por los métodos definidos por esa clase. Por consiguiente, los métodos son los que determinan cómo se pueden utilizar los datos de la clase.

Las variables definidas en el interior de una clase se llaman variables de instancia debido a que cada instancia de la clase (es decir cada objeto de la clase) contiene su propia copia de estas variables. Por consiguiente los datos de un objeto son independientes y únicos de los datos de otro objeto.

### Regla

- Los métodos y variables definidos en una clase se denominan *miembros* de la clase.
- En Java las operaciones se denominan *métodos*.
- En C/C++ las operaciones se denominan *funciones*.
- En C# las operaciones se denominan *métodos* aunque también se admite el término *función*.

Dado que el propósito de una clase es encapsular complejidad, existen mecanismos para ocultar la complejidad de la implementación dentro de la clase. Cada método o variable de una clase se puede señalar como público o privado. La interfaz pública de una clase representa todo lo que los usuarios externos de la clase necesitan conocer o pueden conocer. Los métodos privados y los datos privados solo pueden ser accedidos por el código que es miembro de la clase. Por consiguiente, cualquier otro código que no es miembro de la clase no puede acceder a un método privado o variable *privada*. Dado que los miembros privados de una clase solo pueden ser accedidos por otras partes de su programa a través de los métodos públicos de la clase, se puede asegurar que no sucederá ninguna acción no deseada. Naturalmente, esto significa que la interfaz pública debe ser diseñada de manera cuidadosa para no exponer en forma innecesaria a la clase.

Una **clase** representa un conjunto de cosas o elementos que tienen un estado y un comportamiento común. Así, por ejemplo, Volkswagen, Toyota, Honda y Mercedes son todos coches que representan una clase denominada *Coché*. Cada tipo específico de coche es una *instancia* de una clase, o dicho de otro modo, un **objeto**. Los objetos son miembros de clases y una clase es, por consiguiente, una descripción de un número de objetos similares. Así Juanes, Carlos Vives, Shakira y Paulina Rubio son miembros de la clase *CantantePop*, o de la clase *Músico*.

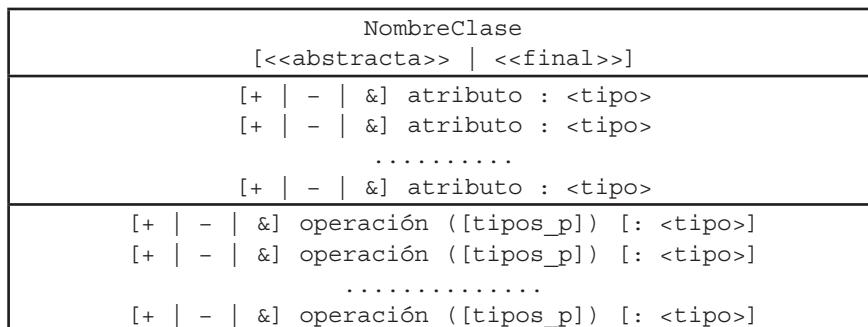
### A recordar

Un *objeto* es una *instancia* o ejemplar de una *clase*.

## Representación gráfica de una clase

Una clase puede representar un concepto tangible y concreto, como un avión; puede ser abstracto, como un documento, o puede ser un concepto intangible como *inversiones de alto riesgo*.

En UML 2.0 una clase se representa con una caja rectangular dividida en compartimentos, o secciones, o bandas. Un compartimento es el área del rectángulo donde se escribe información. El primer compartimento contiene el nombre de la clase, el segundo compartimento contiene los atributos y el tercero se utiliza para las operaciones. Se puede ocultar o quitar cualquier compartimento de la clase para aumentar la legibilidad del diagrama. Cuando no existe un compartimento no significa que esté vacío. Se pueden añadir compartimentos a una clase para mostrar información adicional, como excepciones o eventos, aunque no suele ser normal recurrir a incluir estas propiedades.



*<<tipo>>*: un dato estándar (entero, real, carácter, booleano o cadena) o tipo abstracto de datos.

UML propone que el nombre de una clase:

- Comience con una letra mayúscula.
- El nombre esté centrado en el compartimento (banda) superior.
- Se escriba en un tipo de letra (fuente) negrita.
- Se escriba en cursiva cuando la clase sea abstracta.

Los atributos y operaciones son opcionales, aunque como se ha dicho anteriormente no significa que si no se muestran implique que estén vacíos. La figura 14.19 muestra unos diagramas más fáciles de comprender con las informaciones ocultas.

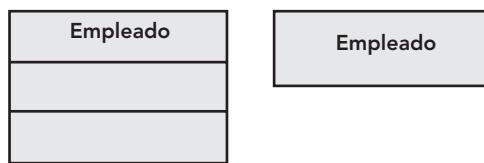


Figura 14.19 Clases en UML.

Cada clase se representa como un rectángulo subdividido en tres compartimentos o bandas. El primer compartimento contiene el nombre de la clase, el segundo comprende los atributos y el último las operaciones. En forma predeterminada, los atributos están ocultos y las operaciones son visibles. Estos compartimentos se pueden omitir para simplificar los diagramas.

### Ejemplos

*La clase Auto contiene los atributos Color, Motor y VelocidadMáxima. La clase puede agrupar las operaciones arrancar, acelerar y frenar*

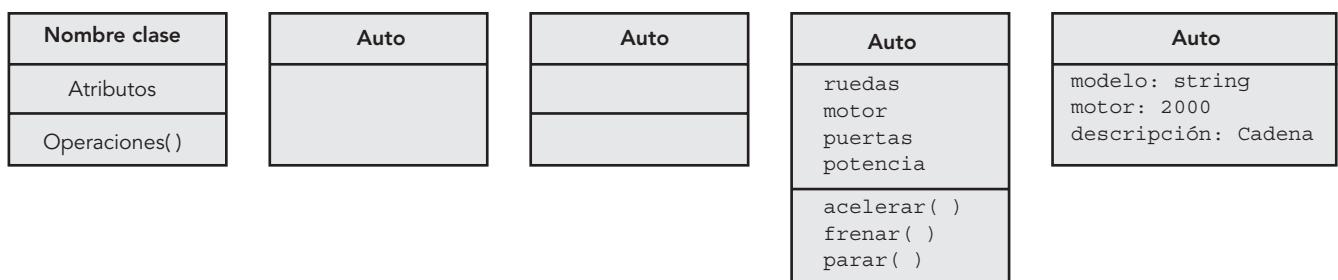


Figura 14.20 Cuatro formas diferentes de representar una clase.

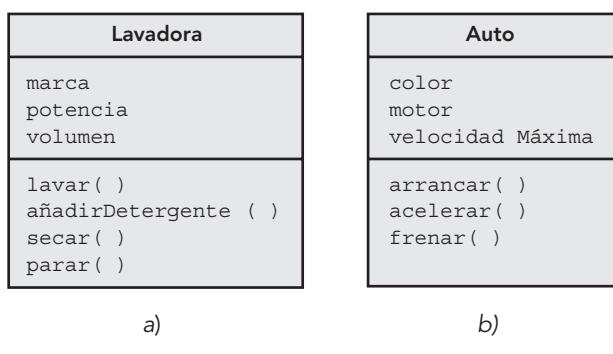


Figura 14.21 Diagrama de clases: a) Lavadora; b) Auto.

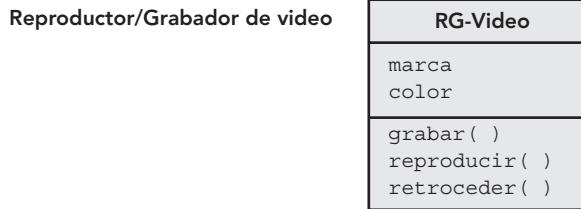


Figura 14.22 Clase RG de video.

### Números complejos

Son aquellos que contienen una parte real y una parte imaginaria. Los elementos esenciales de un número complejo son sus coordenadas y se pueden realizar con ellos numerosas operaciones, como sumar, restar, dividir, multiplicar, etcétera.

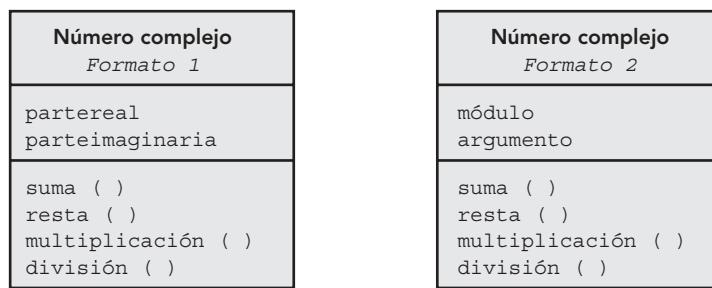


Figura 14.23 Clases Número Complejo.

### Aparato de televisión

Es un dispositivo electrónico de complejidad considerable pero que pueden utilizar adultos y niños. El aparato de televisión ofrece un alto grado de abstracción merced a sus operaciones elementales.

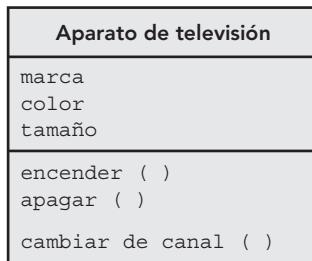


Figura 14.24 Clase aparato de televisión.

### Estructuras de datos

Representar los tipos abstractos de datos que manipulan las estructuras dinámicas de datos fundamentales: listas, pilas y colas.

### Declaración de una clase

La declaración de una clase se divide en dos partes:

- La *especificación* de una clase describe el dominio de la definición y las propiedades de las instancias de esa clase; a la noción de un tipo le corresponde la definición en los lenguajes de programación convencional.
- La *implementación* de una clase describe cómo se implementa la especificación y contiene los cuerpos de las operaciones y los datos necesarios para que las funciones actúen adecuadamente.

Los lenguajes modulares permiten la compilación independiente de la especificación y de la implementación de modo que es posible validar primero la consistencia de las especificaciones (también llamados interfaces) y a continuación validar la implementación en una etapa posterior. En lenguajes de programación, el concepto de tipo, descripción y módulo se integran en el concepto de clase con mayor o menor extensión.

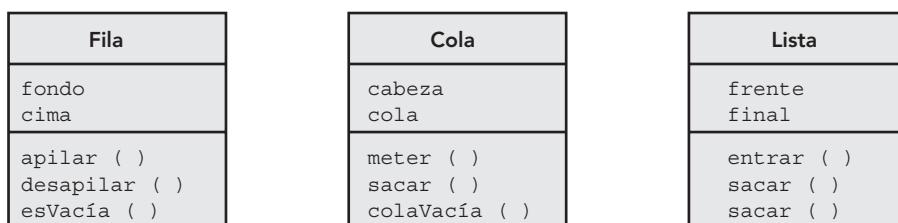


Figura 14.25 Clases de estructuras de datos.

- En C++, la clase se implementa directamente por una construcción sintáctica que incorpora el concepto de tipo, descripción y módulo. La clase se puede utilizar para obtener un módulo único añadiendo la palabra reservada `static` delante de todas las operaciones.
- En Java, la clase también es la integración de los conceptos de tipo, descripción y módulo. También existe un concepto más general de módulos (el paquete) que puede contener varias clases.

La división entre especificación e implementación juega un papel importante en el nivel de abstracción y en consecuencia en el encapsulamiento. Las características más notables se describen en la especificación mientras que los detalles se circunscriben a la implementación.

### Especificación de una clase

Antes de que un programa pueda crear objetos de cualquier clase, se debe *definir* la clase. La definición de una clase significa que a este se le debe dar un nombre, darle nombre a los elementos que almacenan sus datos y describir las funciones que realizarán las acciones consideradas en los objetos.

Las *definiciones* o *especificaciones* no son código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer las funciones que utilizará el programa. Normalmente se sitúan en archivos diferentes de los archivos de código ejecutable, utilizando un archivo para cada clase. Se conocen como *archivos de cabecera* que se almacenan con un nombre de archivo con extensión `.h` en el caso del lenguaje de programación C++.

### Formato

```
clase NombreClase
lista_de_miembros
fin_clase
```

*NombreClase*

Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador válido).

*lista\_de\_miembros*

Funciones y datos miembros de la clase presentan algunas diferencias.

### Reglas de visibilidad

Las reglas de visibilidad complementan o refinan el concepto de encapsulamiento. Los diferentes niveles de visibilidad dependen del lenguaje de programación con el que se trabaje, pero en general siguen el modelo de C++, aunque los lenguajes de programación Java y C# siguen también estas reglas. Estos niveles de visibilidad son:

- El nivel más fuerte se denomina nivel "privado"; la sección privada de una clase es totalmente opaca y solo los amigos (término como se conoce en C++) pueden acceder a atributos localizados en la sección privada.
- Es posible disminuir el nivel de ocultamiento situando algunos atributos en la sección "protegida" de la clase. Estos atributos son visibles tanto para amigos como las clases derivadas de la clase servidor. Para las restantes clases permanecen invisibles.
- El nivel más débil se obtiene situando los atributos en la sección *pública* de la clase, con lo cual se hacen visibles a todas las clases.

Una clase se puede visualizar como en la figura 14.26.



Figura 14.26 Representación de atributos y operaciones con una clase.

El nivel de visibilidad se puede especificar en la representación gráfica de las clases en UML con los símbolos o caracteres #, + y -, que corresponden con los niveles público, protegido y privado, respectivamente.

| Reglas de visibilidad                                                        |
|------------------------------------------------------------------------------|
| + Atributo público<br># Atributo protegido<br>- Atributo privado             |
| + Operación pública ()<br># Operación protegida ()<br>- Operación privada () |

Los atributos privados están contenidos en el interior de la clase ocultos a cualquier otra clase. Ya que los atributos están encapsulados dentro de una clase, se necesitarán definir cuáles son las clases que tienen acceso a visualizar y cambiar los atributos. Esta característica se conoce como *visibilidad de los atributos*. Existen tres opciones de visibilidad (aunque algunos lenguajes como Java y C# admiten una cuarta opción de visibilidad denominada “paquete” o “implementación”). El significado de cada visibilidad es el siguiente:

- **Público.** El atributo es visible a todas las restantes clases. Cualquier otra clase puede visualizar o modificar el valor del atributo. La notación UML de un atributo público es un signo más (+).
- **Privado.** El atributo no es visible a ninguna otra clase. La notación UML de un atributo privado es un signo menos (-).
- **Protegido.** La clase y cualquiera de sus descendientes tienen acceso a los atributos. La notación UML de un atributo protegido es el carácter “libra” o “almodadilla” o numeral (#).

## Ejemplo

Representar una clase *Empleado*

| Empleado                 |
|--------------------------|
| - EmpleadoID: entero = 0 |
| # NSS: cadena            |
| # salario: real          |
| + dirección: cadena      |
| + ciudad: cadena         |
| + provincia: cadena      |
| + código postal: cadena  |
| + contratar()            |
| + despedir()             |
| + promover()             |
| + degradar()             |
| # trasladar()            |

## Notas de ejecución

En general, se recomienda visibilidad privada o protegida para los atributos.

### Ejemplo

Representación de la clase número complejo:

| Número complejo                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- parteReal</li> <li>- parte Imaginaria</li> </ul>                                           |
| <ul style="list-style-type: none"> <li>+ suma ()</li> <li>+ resta ()</li> <li>+ multiplicación ()</li> <li>+ división ()</li> </ul> |

### Nota

1. En Java la declaración de la clase y la implementación de los métodos se almacenan en el mismo sitio y no se definen por separado.
2. En C++, normalmente, la declaración de la clase y la implementación de los métodos se definen en forma separada.

## 14.3 Declaración de objetos de clases

Una vez que se define una clase, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*. Cuando se crea una clase se está creando un nuevo tipo de dato. Se puede utilizar este tipo para declarar objetos de ese tipo.

### Formato

```
nombre_clase: identificador
```

### Ejemplo

```
Punto p; // Clase Punto, objeto p
```

En algunos lenguajes de programación se requiere un proceso en dos etapas para la declaración y asignación de un objeto.

1. Se declara una variable del tipo clase. Esta variable no define un objeto, es simplemente una variable que puede referir a un objeto.
2. Se debe adquirir una copia física, real del objeto y se asigna a esa variable utilizando el operador nuevo (en inglés, *new*). El operador nuevo asigna dinámicamente, es decir, en tiempo de ejecución, memoria para un objeto y devuelve una referencia al mismo. Esta referencia viene a ser una dirección de memoria del objeto asignado por nuevo. Esta referencia se almacena entonces en la variable.<sup>2</sup>

### Sintaxis

```
varClase = nuevo nombreClase()
```

Las dos etapas citadas anteriormente son:

```
Libro milibro // declara una referencia al objeto
miLibro = nuevo Libro() // asigna un objeto Libro
```

<sup>2</sup> En Java, todos los objetos de una clase se deben asignar dinámicamente.

Así, la definición de un objeto Punto es:

```
Punto P;
```

El *operador de acceso* a un miembro (.) selecciona un miembro individual de un objeto de la clase. Las siguientes sentencias, por ejemplo, crean un punto P, que fija su coordenada x y visualiza su coordenada x.

```
Punto p;
P.FijarX (100);
Escribir "coordenada x es", P.devolverX()
```

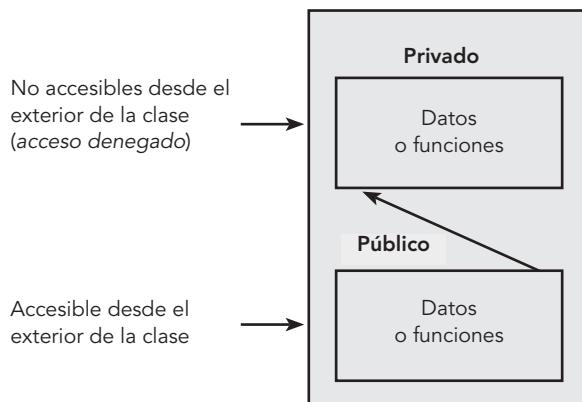
El operador punto se utiliza con los nombres de las funciones miembro para especificar que son miembros de un objeto.

*Ejemplo:* Clase DiaSemana, contiene una función Visualizar

```
DiaSemana: Hoy // Hoy es un objeto
Hoy.Visualizar() // ejecuta la función Visualizar
```

## Acceso a miembros de la clase: encapsulamiento

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*, la cual significa que a determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o funciones privados solo se puede acceder desde dentro de la clase. Por el contrario, los datos o funciones públicos son accesibles desde el exterior de la clase.



**Figura 14.27** Secciones pública y privada de una clase.

Se utilizan tres diferentes *especificadores de acceso* para controlar el acceso a los miembros de la clase: público, privado y protegido.

El especificador público define miembros públicos, que son aquellos a los que se puede acceder por cualquier función. A los miembros que siguen al especificador privado solo se puede acceder por funciones miembro de la misma clase o por funciones y clases amigas.<sup>3</sup> A los miembros que siguen al especificador protegido se puede acceder por funciones miembro de la misma clase o de clases derivadas de la misma, así como por amigas. Los especificadores público, protegido y privado pueden aparecer en cualquier orden.

En la tabla 14.1 cada "x" indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

<sup>3</sup> Las funciones y clases amigas son propias de C++.

En C si se omite el especificador de acceso, el acceso a los atributos se considera privado y a los métodos, público. En la siguiente clase `Estudiante`, por ejemplo, todos los datos son privados, mientras que las funciones miembro son públicas.

**Tabla 14.1** Visibilidad.

| Tipo de miembro | Miembro de la misma clase | Amiga | Miembro de una clase derivada | Función no miembro |
|-----------------|---------------------------|-------|-------------------------------|--------------------|
| privado         | x                         | x     |                               |                    |
| protegido       | x                         | x     | x                             |                    |
| publico         | x                         | x     | x                             | x                  |

En C++ se establecen secciones públicas y privadas en las clases y el mismo especificador de acceso puede aparecer más de una vez en una definición de una clase, pero, en este caso, no es fácil de leer.

```
Clase Estudiante{
private:
 long numId;
public:
 long leerNumId();
private:
 char nombre[40];
 int edad;
public:
 char leerNombre();
 int leerEdad();
};
```

### A recordar

El especificador de acceso se aplica a *todos* los miembros que vienen después de él en la definición de la clase (hasta que se encuentra otro especificador de acceso).

Aunque las secciones públicas y privadas pueden aparecer en cualquier orden, en C++, los programadores suelen seguir algunas reglas en el diseño que citamos a continuación, y que usted puede elegir la que considere que es más eficiente.

1. Poner la sección privada primero, debido a que contiene los atributos (datos).
2. Poner la sección pública primero debido a que las funciones miembro y los constructores son la interfaz del usuario de la clase.

La regla 2 presenta realmente la ventaja de que los datos son algo secundario en el uso de la clase y con una clase definida en forma adecuada, realmente no se suele necesitar nunca ver cómo están declarados los atributos.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, las funciones miembro y los miembros datos de la sección pública forman la interfaz externa del objeto, mientras que los elementos de la sección privada son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

### A recordar

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

**Nota**

Los lenguajes C++, Java y C# proporcionan un especificador de acceso `protected`.

## Declaración de métodos

Las clases normalmente constan de dos tipos de elementos: variables de instancia y métodos. Existen dos formatos para la declaración de los métodos, dependiendo de que se siga el modelo C++ (*función miembro*) o el modelo Java / C# (*método*).

```
C++: tipo_retorno NombreClase:: nombreFuncion(listaParámetros)
{
 // cuerpo de la función
}
Java tipo_retorno NombreClase(listaParámetros)
{
 // cuerpo del método
}
```

Los métodos que tienen un tipo de retorno distinto de `void` devuelven un valor a la rutina llamadora utilizando el formato siguiente de la sentencia `return`:

```
return valor;
```

`valor` es el valor devuelto.

### Notación y asignación de valores a los miembros

La sintaxis estándar para la referencia y asignación de valores a miembros de una clase utiliza los siguientes formatos:

**Formato:**

```
nombre-objeto.nombre-atributo
nombre-objeto.nombre-función(parámetros)
```

### Código Java

```
import java.io.*;
class Fecha
{
 protected int dia, mes, anyo;
 public void fijarFecha(int d, int m, int a)
 {
 dia = d;
 mes = m;
 anyo = a;
 }
 public void mostrarFecha ()
 {
 System.out.println(dia+"/"+mes+"/"+anyo)
 }
 public class PruebaFecha
 {
 public static void main (String[] args)
 {
 Fecha f1, f2;
 f1 = new Fecha();
 }
 }
}
```

```

f2 = new Fecha();
f1.dia = 15;
f1.mes = 7;
f1.anyo = 2002;
f1.mostrarFecha()
f2.fijarFecha(20, 8, 2002);
f2.mostrarFecha();
}
}

```

El resultado de la ejecución es:

```

15/7/2002
20/8/2002

```

A continuación se muestra un código en C++ donde se establecen los atributos como privados y las funciones miembro como públicas. A diferencia de Java, el modo de acceso predeterminado en C++ es `private`. Otra diferencia con Java es que en C++ las funciones miembro se declaran dentro de la clase pero, aunque puede hacerse dentro, suelen definirse fuera de la clase. En la definición de una función miembro fuera de la clase el nombre de la función ha de escribirse precedido por el nombre de la clase y por el operador binario de resolución de alcance (::)

```

#include <iostream>
//Declaración de datos miembro y funciones miembro
class Fecha
{
 int dia, mes, anyo;
public:
 void fijarFecha(int, int, int);
 void mostrarFecha();
};

void Fecha::fijarFecha(int d, int m, int a)
{
 dia = d;
 mes = m;
 anyo = a;
}
void Fecha::mostrarFecha ()
{
 cout << dia << "/" << mes << "/" << anyo;
}

//Prueba Fecha
int main()
{
 Fecha f;
 f.fijarFecha(20, 8, 2002);
 cout << "La fecha es "
 f.mostrarFecha();
 cout << endl;
 return 0;
}

```

El código en C# sería:

```

using System;
class Fecha
{
 int dia, mes, anyo;
 public void fijarFecha(int d, int m, int a)

```

```

{
 dia = d;
 mes = m;
 anyo = a;
}
public void mostrarFecha(){
 Console.WriteLine(dia+"/"+mes+"/"+anyo);
}
}

class PruebaFecha
{
 public static void main()
 {
 Fecha f;
 f = new Fecha();
 f.fijarFecha(20, 8, 2002);
 f.mostrarFecha();
 }
}

```

Los campos de datos en este último ejemplo también son privados, por lo que, con el código especificado, la sentencia

```
Console.WriteLine (f.dia);
```

daría error.

## Tipos de métodos

Los métodos que pueden aparecer en la definición de una clase se clasifican en función del tipo de operación que representan. Estos métodos tienen una correspondencia con los tipos de mensajes que se pueden enviar entre los objetos de una aplicación, como por otra parte era lógico pensar.

- *Constructores y destructores* son funciones miembro a las que se llama automáticamente cuando un operador se crea o se destruye.
- *Selectores*, que devuelven los valores de los miembros dato.
- *Modificadores o mutadores*, que permiten a un programa cliente cambiar los contenidos de los miembros dato.
- *Operadores*, que permiten definir operadores estándar para los objetos de las clases.
- *Iteradores*, que procesan colecciones de objetos, como arreglos y listas.



## Resumen

Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real.

Un objeto de una clase tiene dos componentes; un conjunto de atributos y un conjunto de comportamientos (operaciones). Los atributos se llaman miembros dato y los comportamientos se llaman funciones miembro.

```

clase Circulo
var
 publico real: x_centro,

```

```

y_centro, radio
pùblico real función superficie()
 inicio
 ...
 fin_función
fin_clase

```

Cuando se crea un nuevo tipo de clase, se deben realizar dos etapas fundamentales: determinar los atributos y el comportamiento de los objetos.

Un objeto es una instancia de una clase.

Circulo un\_circulo

Una declaración de una clase se divide en tres secciones: pública, privada y protegida. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Se recomienda que los constructores sean declarados en la sección pública. La sección privada contiene las funciones miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estas funciones miembro y atributos dato son accesibles solo por la función miembro del objeto.

El acceso a los miembros de una clase se puede declarar como *privado* (private), *público* (public) o *protegido* (protected).

```
clase Circulo
var
 privado real: centro_x,
 centro_y, radio
público real función Superficie ()
 inicio
 ...
 devolver (...)
 fin función
público procedimiento fijar-Centro
(E real: x, y)
 inicio
 ...
 fin_procedimiento
público procedimiento Fijar-Radio
(E real: x, y)
 inicio
 ...
```

```
 fin_procedimiento
público procedimiento Fijar-Radio
(E real: r)
 inicio
 ...
 fin_procedimiento
público real función DevolverRadio ()
 inicio
 ...
 devolver (...)
 fin_función
fin_clase
```

Los miembros dato `centro_x`, `centro_y` y `radio` son ejemplos de ocultación de datos.

El procedimiento fundamental de especificar un objeto es:

`circulo :c` // un objeto

y para especificar un miembro de una clase

`radio = 10.0` // Miembro de la clase

El operador de acceso a miembro (el operador punto).

`c. radio = 10.0;`

Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobre cargado.

```
clase Complejo
...
constructor Complejo (real: x,y)
 inicio
 ...
 fin_constructor
```

## Ejercicios

14.1 Considera una pila como un tipo abstracto de datos. Se trata de definir una clase que implementa una pila de 100 caracteres mediante un arreglo. Las funciones miembro de la clase deben ser:

`meter`, `sacar`, `pilavacia` y `pilallena`.

14.2 Escribir la clase pila que utilice una lista enlazada en lugar de un arreglo (sugerencia: utilice otra clase para representar los modos de la lista).

14.3 Crear una clase llamada hora que tenga miembros datos separados de tipo `int` para horas, minutos y se-

gundos. Un método o función inicializará este dato a 0 y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal `main()` crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.

14.4 Crear una clase llamada empleado que contenga como miembro dato el nombre y el número de empleado, y como funciones miembro `leerdatos()` y `verda-`

tos() que lean los datos del teclado y los visualice en pantalla, respectivamente.

Escribir un programa que utilice la clase, creando un arreglo de tipo empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez llenado el arreglo, visualizar los datos de todos los empleados.

14.5 Realizar una clase `vector3d` que permita manipular vectores de tres componentes (coordenadas x, y, z) de acuerdo con las siguientes normas:

- Solo posee una función constructor y es en línea.
- Tiene una función miembro `igual` que permite saber si dos vectores tienen sus componentes o coordenadas iguales (la declaración de `igual` se realizará utilizando: a) transmisión por valor; b) transmisión por dirección; c) transmisión por referencia).

14.6 Incluir en la clase `vector3d` del ejercicio anterior una función miembro denominada `normamax` que permite obtener la norma mayor de dos vectores (Nota: La norma de un vector  $v = x, y, z$  es  $x^2 + y^2 + z^2$  o bien  $x*x + y*y + z*z$ ).

14.7 Incluir en la clase `vector3d` del ejercicio anterior las funciones miembro `suma` (suma de dos vectores), `productoescalar` (producto escalar de dos vectores:  $v1 = x1, y1, z1; v2 = x2, y2, z2; v1 * v2 = x1 * x2 + y1 * y2 + z1 * z2$ ).

14.8 Realice una clase `Complejo` que permita la gestión de números complejos (un número complejo = dos números reales `double`: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- a) Una función `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- b) Una función `imprimir()` realiza la visualización formateada de un `Complejo`.
- c) Dos funciones `agregar()` (sobrecargadas) permiten añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

14.9 Escribir una clase `Conjunto` que gestione un conjunto de enteros (`int`) con ayuda de una tabla de tamaño fijo (un `conjunto` contiene una lista no ordenada de elementos y se caracteriza por el hecho de que cada elemento es único: no se debe encontrar dos veces el mismo valor en la tabla). Las operaciones a implementar son las siguientes:

- a) La función `vacía()` vacía el conjunto.
- b) La función `agregar()` añade un entero al conjunto.

c) La función `eliminar()` retira un entero del conjunto.

d) La función `copiar()` recopila un conjunto en otro.

e) La función `es_miembro()` reenvía un valor booleano (lógicos que indica si el conjunto contiene un elemento, un entero dado).

f) La función `es_igual()` reenvía un valor booleano que indica si un conjunto es igual a otro.

g) La función `imprimir()` realiza la visualización formateada del conjunto.

14.10 Crear una clase `lista` que realice las siguientes tareas:

a) Una lista simple que contenga cero o más elementos de algún tipo específico.

b) Crear una lista vacía.

c) Añadir elementos a la lista.

d) Determinar si la lista está vacía.

e) Determinar si la lista está llena.

f) Acceder a cada elemento de la lista y realizar alguna acción sobre ella.

14.11 Añadir a la clase `Hora` del ejercicio 14.3 las funciones de acceso, una función `adelantar(int h, int m, int s)` para adelantar la hora actual de un objeto existente, una función `reiniciar(int h, int m, int s)` que reinicia la hora actual de un objeto existente y una función `imprimir()`.

14.12 Añadir a la clase `Complejo` del ejercicio 14.8 las operaciones.

a) Suma:  $a + c = (A + C, (B + D)i)$ .

b) Resta:  $a - c = (A - C, B - Di)$ .

c) Multiplicación:  $a * c = (A * C - B * D, (A * D + B * C)i)$ .

d) Multiplicación:  $x * c = (x * C, x * Di)$ , donde  $x$  es real.

e) Conjugado:  $\bar{a} = (A, -Bi)$ .

14.13 Implementar una clase `Random` (aleatoria) para generar números pseudoaleatorios.

14.14 Implementar una clase `Fecha` con miembros `dato` para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, un constructor de copia, funciones de acceso, una función `reiniciar(int d, int m, int a)` para reiniciar la fecha de un objeto existente, una función `adelantar(int d, int m, int a)` para avanzar a una fecha existente (dia, d, mes, m, y año a) y una función `imprimir()`. Utilizar una función de utilidad `normalizar()` que asegure que los miembros `dato` están en el rango correcto  $1 \leq \text{año} \leq 99$ ,  $1 \leq \text{mes} \leq 12$ ,  $1 \leq \text{día} \leq 31$ .

$\leq \text{días}(\text{Mes})$ , donde  $\text{días}(\text{Mes})$  es otra función que devuelve el número de días de cada mes.

- 14.15 Ampliar el programa anterior de modo que pueda aceptar años bisiestos. **Nota:** un año es bisiesto si es

divisible entre 400 o si es divisible entre 4 pero no entre 100. Por ejemplo el año 1992 y 2000 son años bisiestos y 1997 y 1900 no lo son.



# Relaciones entre clases: delegaciones, asociaciones, agregaciones, herencia

## Contenido

- 15.1 Relaciones entre clases
- 15.2 Dependencia
- 15.3 Asociación
- 15.4 Agregación
- 15.5 Jerarquía de clases: generalización y especialización

## 15.6 Herencia: clases derivadas

- 15.7 Accesibilidad y visibilidad en la herencia
- 15.8 Un caso de estudio especial: herencia múltiple
- 15.9 Clases abstractas
  - › Resumen
  - › Ejercicios y problemas

## Introducción

En este capítulo se introducen los conceptos fundamentales de relaciones entre clases. Las relaciones más importantes soportadas por la mayoría de las metodologías de orientación a objetos y en particular por UML son: *asociación, agregación y generalización/especialización*. En el capítulo se describen estas relaciones así como las notaciones gráficas correspondientes en UML.

De modo especial se introduce el concepto de *herencia* como exponente directo de la relación de generalización/especialización y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base a clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, bajo costo, adaptable y reutilizable.

## Conceptos clave

- › Agregación
- › Asociación
- › Clase abstracta
- › Clase base
- › Clase derivada
- › Composición
- › Especificador de acceso
- › Generalización
- › Herencia
- › Herencia múltiple
- › Herencia privada
- › Herencia protegida
- › Herencia pública
- › Herencia simple
- › Multiplicidad
- › Relación es-un
- › Relación todo-parte

## 15.1 Relaciones entre clases

Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las clases no actúan aisladas entre sí, al contrario, están relacionadas unas con otras. Una clase puede

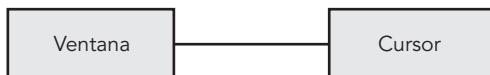


Figura 15.1 Relaciones entre clases.

ser un tipo de otra clase, *generalización*, o bien puede contener objetos de otra clase de varias formas posibles, dependiendo de la fortaleza de la relación entre las dos clases.

La fortaleza de una relación de clases [Miles, Hamilton, 2006] se basa en el modo de dependencia de las clases implicadas en las relaciones entre ellas. Se dice que dos clases son fuertemente dependientes una de otra si están *acopladas fuertemente* y en caso contrario, que están *acopladas débilmente*.

Las relaciones entre clases se corresponden con las relaciones entre objetos físicos del mundo real, o bien objetos imaginarios en un mundo virtual. En UML las formas en las que se conectan entre sí las clases, lógica o físicamente, se modelan como relaciones. En el modelado orientado a objetos existen tres clases de relaciones muy importantes: *dependencias*, *generalizaciones-especializaciones* y *asociaciones* [Booch 06:66]:

- Las *dependencias* son relaciones de uso.
- Las *asociaciones* son relaciones estructurales entre objetos. Una relación de *asociación* “todo/parte”, en la cual una clase representa un cosa grande (“el todo”) que consta de elementos más pequeños (“las partes”) se denomina *agregación*.
- Las *generalizaciones* conectan clases generales con otras más especializadas en lo que se conoce como relaciones subclase/superclase o hijo/padre.

Una *relación* es una conexión entre elementos. En el modelado orientado a objetos una relación se representa gráficamente con una línea (continua, punteada) que une las clases. En UML la relación es uno de los bloques o componentes más importantes de UML. Una relación en UML muestra cómo los elementos se asocian unos con otros y esta asociación describe la funcionalidad de una aplicación. En UML 2.5 existen cuatro tipos de relaciones disponibles (tabla 15.1):

Tabla 15.1 Relaciones de clases.

| Relación       | Descripción y representación gráfica                                                                                                                                                                                                            |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dependencia    | Relación entre dos cosas en la que cualquier cambio en un elemento afecta también al otro.<br>Dependencia<br>- - - - - →                                                                                                                        |
| Asociación     | Conjunto de elementos que conectan elementos de un modelo UML. También describe cuántos objetos forman parte de la relación.<br>Asociación<br>← →                                                                                               |
| Generalización | Es una relación que conecta un elemento especializado con un elemento generalizado. Básicamente describe la relación de herencia en el mundo de objetos.<br>Generalización<br>→                                                                 |
| Realización    | Es una relación en la cual dos elementos están conectados. Un elemento describe una responsabilidad que no está implementada y el otro elemento la implementa. Esta relación existe en el caso de las interfaces.<br>Realización<br>- - - - - → |

## 15.2 Dependencia

La relación más débil que puede existir entre dos clases es una relación de *dependencia*. Una dependencia entre clases significa que una clase utiliza, o tiene conocimiento de otra clase, o dicho de otro modo “lo



**Figura 15.2** Relación de dependencia. Ventana depende de la clase EventoCerrarVentana porque necesitará leer el contenido de esta clase para poder cerrar la ventana.

que una clase necesita conocer de otra clase para utilizar objetos de esa clase" (Russ Miles & Kim Hamilton, *Learning UML 2.0*, O'Reilly, pp. 81-82). Normalmente es una relación transitoria y significa que una clase dependiente interactúa de manera breve con la clase destino, pero normalmente no tiene con ella una relación de un tiempo definido. Una *dependencia* es una relación de uso que declara que un elemento utiliza la información y los servicios de otro elemento pero no necesariamente a la inversa.

La dependencia se lee normalmente como una relación "... *usa un* ...". Por ejemplo, si se tiene una clase Ventana que envía un aviso a una clase llamada EventoCerrarVentana cuando está próxima a abrirse. Entonces se dice que Ventana utiliza EventoCerrarVentana.

En un diagrama de clases, la dependencia se representa utilizando una línea discontinua dirigida hacia el elemento del cual se depende. La flecha punteada de dependencia (figura 15.2) que significa "se utiliza simplemente cuando se necesita y se olvida luego de ella".

Otro ejemplo de dependencia se muestra entre la clase *Interfaz* y la clase *EntradaBlog* ya que ambas clases trabajan juntas puesto que *Interfaz* necesitará leer el contenido de las entradas del *blog* para visualizar estas entradas al usuario.

Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro. Una dependencia implica que los objetos de una clase pueden trabajar juntos; por consiguiente, se considera que es la relación directa más débil que puede existir entre dos clases.

### 15.3 Asociación

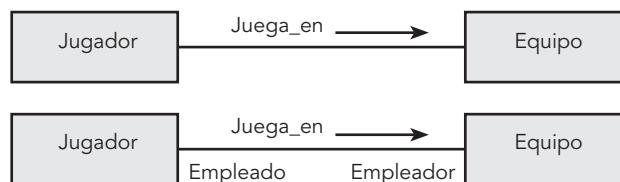
Una **asociación** es más fuerte que la dependencia y normalmente indica que una clase recuerda o retiene una relación con otra clase durante un tiempo determinado. Es decir, las clases se conectan juntas conceptualmente en una asociación. La asociación realmente significa que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo. La asociación se representa utilizando una simple línea que conecta las dos clases, como se muestra en la figura 15.3.

Una *asociación* es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra clase. En general, si se encuentra que una clase *trabaja con* un objeto de otra clase, entonces la relación entre esas clases es una buena candidata para una asociación en lugar de una dependencia.

Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases. Las asociaciones se deben utilizar cuando se deseé representar relaciones estructurales. Los adornos de una asociación son: línea continua, nombre de la asociación, dirección del nombre mediante una flecha que apunta en la dirección de una clase a la otra. La *navegabilidad* se aplica a una relación de asociación que describe qué clase contiene el atributo que soporta la relación.

Si se encuentra que una clase *trabaja con* un objeto de otra clase, entonces la relación entre clases es candidata a una asociación en lugar de a una dependencia. Cuando una clase se asocia con otra clase, cada una juega un rol dentro de la asociación. El rol se representa cerca de la línea próxima a la clase. En la asociación entre un *Jugador* y un *Equipo*, si esta es profesional, el equipo es el *Empleador* y el jugador es el *Empleado*.

Una asociación puede ser bidireccional. Un *Equipo* emplea a *Jugadores*.



**Figura 15.3** Relación de asociación (Jugador-Equipo).

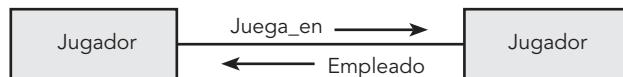


Figura 15.4 Relación de asociación bidireccional.

También pueden existir asociaciones entre varias clases, de modo que varias clases se pueden conectar a una clase.

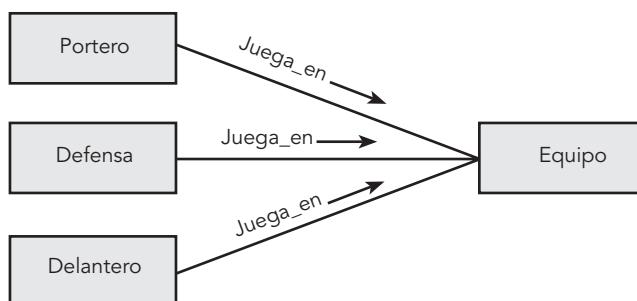


Figura 15.5 Asociación entre varias clases.

Una **asociación** es una conexión conceptual o semántica entre clases. Cuando una asociación conecta dos clases, cada clase envía mensajes a la otra en un diagrama de colaboración. *Una asociación es una abstracción de los enlaces que existen entre instancias de objetos.* Los siguientes diagramas muestran objetos enlazados a otros objetos y sus clases correspondientes asociadas. Las asociaciones se representan de igual modo que los enlaces. La diferencia entre un enlace y una asociación se determina de acuerdo con el contexto del diagrama.

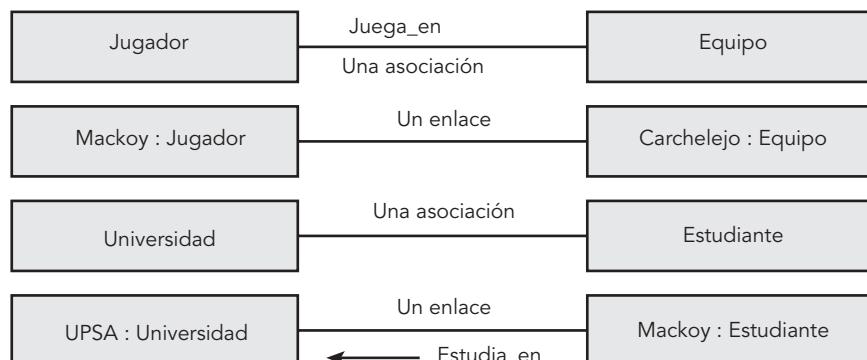


Figura 15.6 Asociación entre clases.

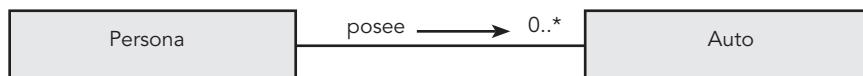
### Regla

El significado más típico en una conexión entre clases es una relación semántica entre clases. Se dibuja con una línea continua entre las dos clases. La asociación tiene un nombre (cerca de la línea que representa la asociación), normalmente un verbo, aunque están permitidos los nombres o frases nominales. Cuando se modela un diagrama de clases, se debe reflejar el sistema que se está construyendo y por ello los nombres de la asociación deben deducirse del dominio del problema, al igual que sucede con los nombres de las clases.



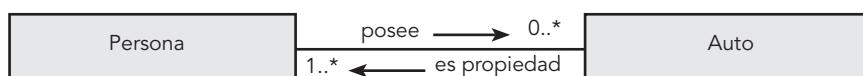
**Figura 15.7** Un programador utiliza una computadora. La clase *Programador* tiene una asociación con la clase *Computadora*.

Es posible utilizar asociaciones navegables añadiendo una flecha al final de la asociación. La flecha indica que la asociación solo se puede utilizar en la dirección de la flecha.



**Figura 15.8** Una asociación navegable representa a una persona que posee (es propietaria) de varios carros, pero no implica que un auto pueda ser propiedad de varias personas.

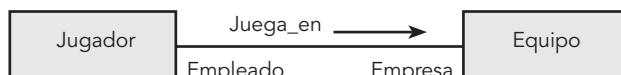
Las asociaciones pueden tener dos nombres, uno en cada dirección.



**Figura 15.9** Una asociación navegable en ambos sentidos, con un nombre en cada dirección.

Las asociaciones pueden ser bidireccionales o unidireccionales. En UML las asociaciones bidireccionales se dibujan con flechas en ambos sentidos. Las asociaciones unidireccionales contienen una muestra que muestra la dirección de navegación.

En las asociaciones se pueden representar los roles o papeles que juegan cada clase dentro de las mismas. La figura 15.10 muestra cómo se representan los roles de las clases. Un nombre de rol puede ser especificado en cualquier lado de la asociación. Los nombres de los roles son especialmente interesantes cuando varias asociaciones conectan dos clases idénticas.

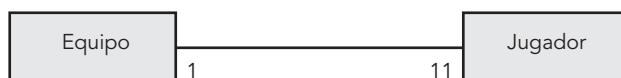


**Figura 15.10** Roles en las asociaciones.

## Multiplicidad

Entre asociaciones existe la propiedad de la *multiplicidad*: número de objetos de una clase que se relaciona con un único objeto de una clase asociada (un equipo de fútbol tiene once jugadores).

En notación moderna, a esta relación se le suele llamar también “*tiene un*”, pero hay que tener cuidado porque este concepto es sutil y de hecho siempre ha representado a la agregación, pero como se verá después UML contempla que la agregación *posee* (... *owns a* ...). Por esta razón nos inclinaremos en considerar la relación “*tiene-un*” (*has-a*) como la relación de agregación.



**Figura 15.11** Multiplicidad en una asociación.

La multiplicidad representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada. La información de multiplicidad aparece en el diagrama de clases a continuación del rol correspondiente. La multiplicidad se escribe como una expresión con un valor mínimo y un valor máximo, que pueden ser iguales; se utilizan dos puntos consecutivos para separar ambos valores. Cuando se indica una multiplicidad en un extremo de una asociación se está especificando cuántos objetos de la clase de ese extremo puede existir por cada objeto de la clase en el otro extremo. UML utiliza un asterisco (\*) para representar *más* y representa *muchos*. La tabla 15.2 resume los valores más típicos de multiplicidad.

**Tabla 15.2** Multiplicidad en asociaciones.

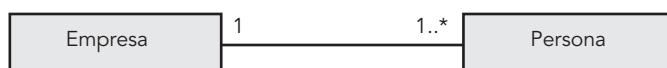
| Símbolo | Significado                                  |
|---------|----------------------------------------------|
| 1       | Uno y solo uno                               |
| 0 .. 1  | Cero o uno                                   |
| m .. n  | De m a n (enteros naturales)                 |
| *       | Muchos (cualquier entero positivo)           |
| 0 .. *  | De cero a muchos (cualquier entero positivo) |
| 1 .. *  | De uno a muchos (cualquier entero positivo)  |
| 2       | Dos                                          |
| 5 .. 11 | Cinco a once                                 |
| 5, 10   | Cinco o diez                                 |

Si no se especifica multiplicidad, es uno (1) por omisión. La multiplicidad se muestra cerca de los extremos de la asociación, en la clase donde es aplicable.



### Ejemplo 15.1

Relación de asociación entre las clases Empresa y Persona



- Cada objeto **Empresa** tiene como **empleados**, 1 o más objetos **Persona**; pero cada objeto **Persona** tiene como **patrón** a un objeto **Empresa**.

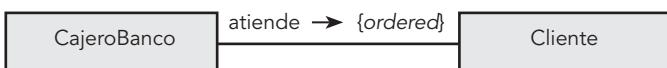
### Restricciones en asociaciones

A veces, una asociación entre dos clases ha de seguir una regla. En este caso, la regla se indica poniendo una restricción cerca de la línea de la asociación que se representa por el nombre encerrado entre llaves.



### Ejemplo 15.2

Un cajero de un banco (humano o electrónico) atiende a clientes. La atención a los clientes se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.



**Figura 15.12** Restricción en una asociación.

- En algunas ocasiones las asociaciones pueden establecer una restricción entre las clases. Las restricciones típicas pueden ser **{ordenado}** (*ordered*) **{or}**.



Figura 15.13 Asociación cualificada.

## Asociación cualificada

Cuando la multiplicidad de una asociación es de uno a muchos, se puede reducir esta multiplicidad de uno a uno con una cualificación. El símbolo que representa la cualificación es un pequeño rectángulo adjunto a la clase correspondiente. Ejemplo: Número de cuenta.

## Asociaciones reflexivas

A veces, una clase es una asociación consigo misma. Esta situación se puede presentar cuando una clase tiene objetos que pueden jugar diferentes roles.

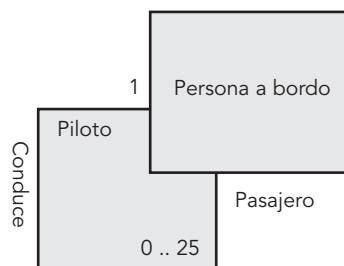


Figura 15.14 Asociación reflexiva.

## Diagrama de objetos

Los objetos se pueden representar en diagramas de objetos. Un diagrama de objetos en UML tiene la misma notación y relaciones que un diagrama de clases, dado que los objetos son instancias de las clases. Así, un diagrama de clases muestra los tipos de clases y sus relaciones, mientras que el diagrama de objetos muestra instancias específicas de esas clases y enlaces específicos entre esas instancias en un momento dado. El diagrama de objetos muestra también cómo los objetos de un diagrama de clases se pueden combinar con cada uno de los restantes en un cierto momento.

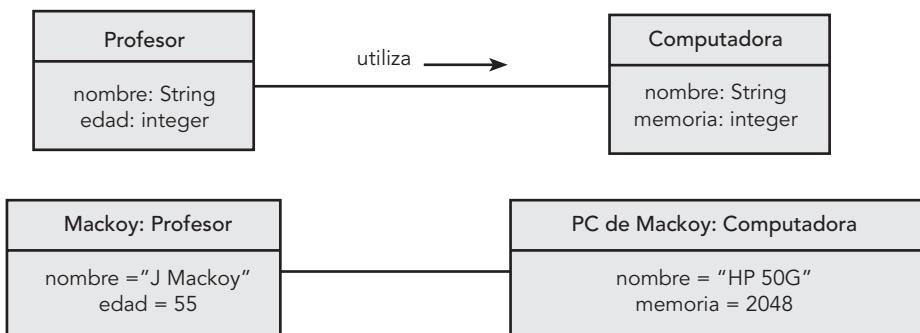


Figura 15.15 Diagrama de clases y diagrama de objetos.

## Enlaces

Al igual que un objeto es una instancia de una clase, una asociación tiene también instancias. Por ejemplo, la asociación de la figura 15.16.

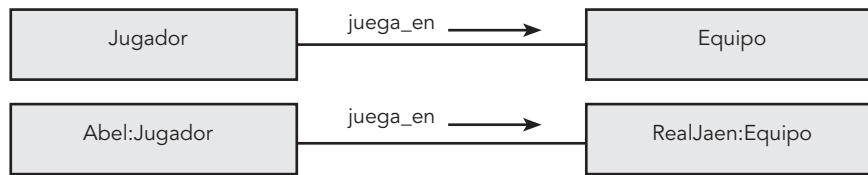


Figura 15.16 Instancia de una asociación.

## Clases de asociación

Es frecuente encontrarse con una asociación que introduce nuevas clases. Una clase se puede conectar a una asociación, en cuyo caso se denomina *clase asociación*. De hecho una asociación puede tener atributos y operaciones como una clase; este es el caso de la clase asociación.

La clase asociación no se conecta a ninguno de los extremos de la asociación, sino que se conecta a la asociación real, a través de una línea punteada. La clase asociación se utiliza para añadir información extra en un enlace, por ejemplo, el momento en que fue creado. Cada enlace de la asociación se relaciona a un objeto de la clase asociación. La clase asociación se utiliza para añadir información extra en un enlace, por ejemplo, el momento en que se crea el enlace. Cada enlace de la asociación se relaciona a un objeto de la clase asociación.

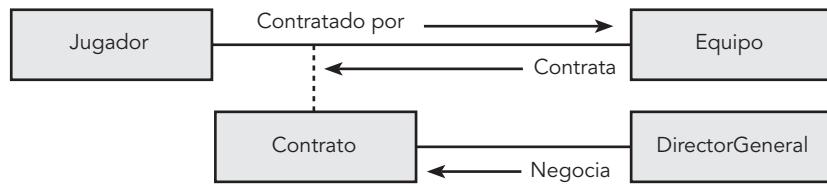


Figura 15.17 La clase asociación Contrato está asociada con la clase DirectorGeneral.

### A recordar

Una clase asociación es una asociación, con métodos y atributos, que es también una clase normal. La clase asociación se representa con una línea punteada que la conecta con la asociación que representa.

Las clases de asociación se pueden aplicar en asociaciones binarias y *n*-arias. De modo similar a como una clase define las características de sus objetos, incluyendo sus características estructurales y sus características de comportamiento, una clase asociación se puede utilizar para definir las características de sus enlaces, incluyendo sus características estructurales y características de comportamiento. Estos tipos de clases se utilizan cuando se necesita mantener información sobre la propia relación.



### Ejemplo 15.3

Clase asociación EquipoFutbol.

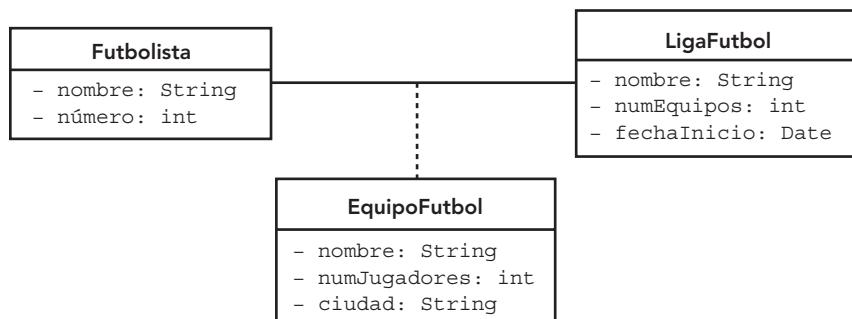


Figura 15.18 Clase asociación EquipoFutbol.

### Criterios de diseño

Cuando se traducen a código, las relaciones con clases asociación se obtienen, normalmente, tres clases: una por cada extremo de la asociación y una por la propia clase asociación.

#### Ejercicio 15.1

Diseñar el control de cinco ascensores (elevadores) de un edificio comercial que tenga presente las peticiones de viaje de los diferentes clientes, en función del orden y momento de llamada.

#### Análisis

El control de ascensores tiene un enlace con cada uno de los cinco ascensores y otro enlace con el botón (pulsador) de llamada de “subida/bajada”. Para gestionar el control de llamadas de modo que responda el ascensor, que cumpla con los requisitos estipulados (situado en piso más cercano, parado, en movimiento, etc.), se requiere una clase **Cola** que almacene las peticiones tanto del **Control Ascensor** como del propio ascensor (los motores interiores del ascensor). Cuando el control del ascensor elige un ascensor para realizar la petición de un pasajero externo al ascensor, un pasajero situado en un determinado piso o nivel, el control del ascensor lee la cola y elige el ascensor que está situado, disponible y más próximo en la **Cola**. Esta elección normalmente se realizará por algún algoritmo inteligente.

En consecuencia, se requieren cuatro clases: **ControlAscensor**, **Ascensor** (elevador), **Botón** (pulsador) y **Cola**. La clase **Cola** será una clase asociación ya que puede ser requerida tanto por el control de ascensores como por cualquier ascensor.

Recuerde que una estructura de datos cola es una estructura en la que cada elemento que se introduce en la cola es el primer elemento que sale de la cola (al igual que sucede con la cola para sacar una entrada de cine, comprar el pan o una cola de impresoras conectadas a una computadora central). En cada enlace entre los ascensores y el control de ascensores hay una cola. Cada cola almacena las peticiones del control del ascensor y el propio ascensor (los botones internos del ascensor).

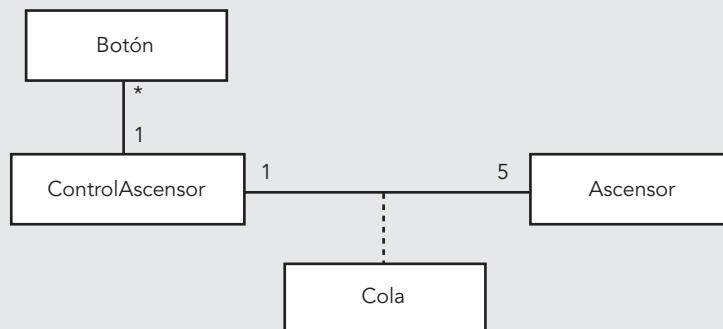


Figura 15.19 Diagrama de clases para control de ascensores.

### Asociaciones ternarias

Las clases se pueden asociar una a una o bien se pueden asociar unas con otras. La asociación ternaria es una asociación especial que asocia a tres clases. La asociación ternaria se representa con la figura geométrica “rombo” y con los roles y multiplicidad necesarias, pero no están permitidos los cualificados ni la agregación. Se puede conectar una clase asociación a la asociación ternaria, dibujando una línea punteada a uno de los cuatro vértices del rombo.

#### Ejercicio 15.2

Dibujar un modelo de seguros de automóviles que represente: compañía de seguros, asegurados, póliza de seguro y el contrato de seguro.

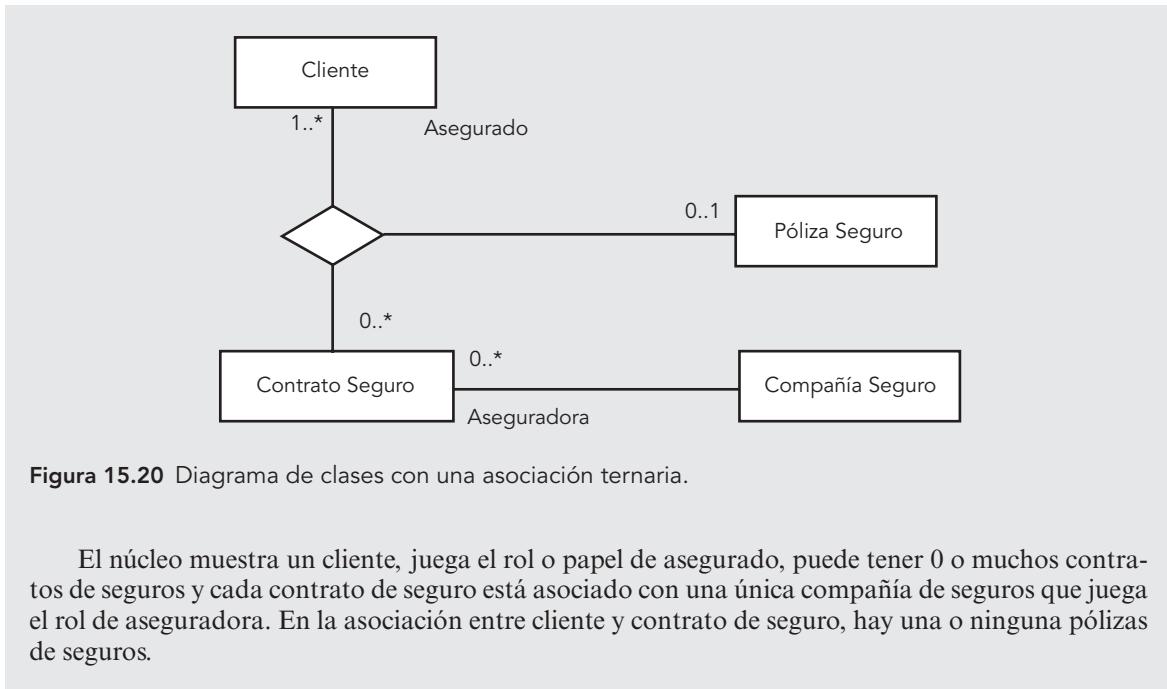


Figura 15.20 Diagrama de clases con una asociación ternaria.

El núcleo muestra un cliente, juega el rol o papel de asegurado, puede tener 0 o muchos contratos de seguros y cada contrato de seguro está asociado con una única compañía de seguros que juega el rol de aseguradora. En la asociación entre cliente y contrato de seguro, hay una o ninguna pólizas de seguros.

### Asociaciones cualificadas

Una asociación cualificada se utiliza con asociaciones una-a-muchas o muchas-a-muchas para reducir su multiplicidad a una, con objeto de especificar un objeto único (o grupos de objetos) desde el destino establecido. La asociación cualificada es muy útil para modelar cuando se busca o navega para encontrar objetos específicos en una colección determinada.

Ejemplos típicos son los sistemas de reservas de pasaje de avión, de entradas de cine, de reservas de habitaciones en hoteles. Cuando se solicita un pasaje, una entrada o una habitación, es frecuente, que nos den un localizador de la reserva (H234JK, o similar) que se ha de proporcionar físicamente a la hora de sacar el pasaje en el aeropuerto, la entrada en el cine o ir a alojarse en el hotel.

El atributo o calificador se conoce normalmente como *identificador* (número de ID). Existen numerosos calificadores como: ID de reserva, nombre, número de la tarjeta de crédito, número de pasaporte, etc. En terminología de proceso de datos, también se conoce como clave de búsqueda. Este identificador o calificador, al especificar una clave única resuelve la relación uno a muchos y lo convierte en uno a uno.

El calificador se representa como una caja o rectángulo pequeño que se dibuja en el extremo correspondiente de la asociación, al lado de la clase a la cual está asociada. El calificador representa un añadido a la línea de la asociación y fuera de la clase. Las asociaciones cualificadas reducen la multiplicidad real en el modelo, de uno-a-muchos a uno-a-uno indicando con el calificador una identidad para cada asociación.



#### Ejemplo 15.4

1. Lista de reservas. Calificador, ID de la reserva.

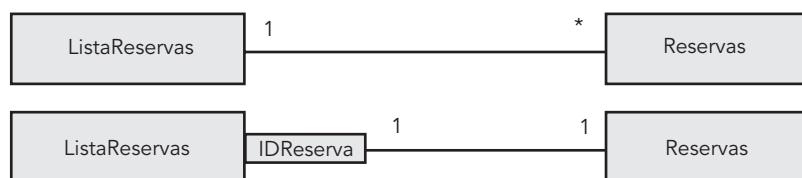


Figura 15.21 Asociación cualificada.

2. Lista de pasajes: vuelo Madrid-Cartagena de Indias con la compañía aérea Iberia.

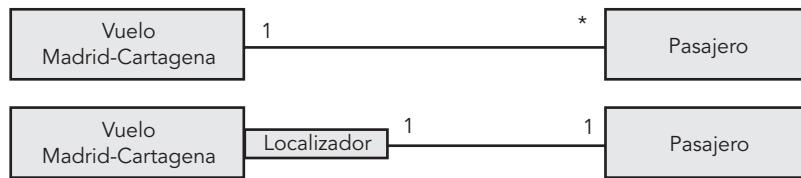


Figura 15.22 Asociación cualificada.

## Asociaciones reflexivas

En ocasiones una clase es una asociación consigo misma. En este caso la asociación se denomina asociación reflexiva. Esta situación se produce cuando una clase tiene objetos que pueden jugar diferentes roles. Por ejemplo, un ocupante de un avión de pasajeros puede ser: un pasajero, un miembro de tripulación o un piloto. Este tipo de asociación se representa gráficamente dibujando la línea de asociación con origen y final en la propia clase y con indicación de los roles y multiplicidades correspondientes.

1. Asociación reflexiva OcupanteAvión.

Ejemplo 15.5

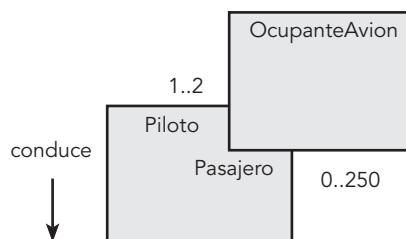


Figura 15.23 Asociación reflexiva.

2. Asociación reflexiva OcupanteAuto.

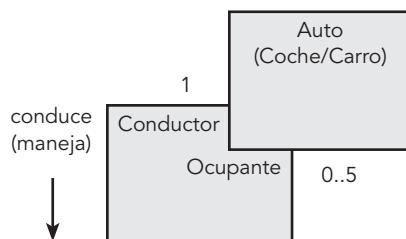


Figura 15.24 Asociación reflexiva.

## Restricciones en asociaciones

Una asociación entre clases, a veces, tiene que seguir una regla determinada. Esta regla se indica poniendo una restricción cerca de la línea de la asociación. Una restricción típica se produce cuando una clase (un objeto) atiende a otra clase (un objeto) en función de un determinado orden o secuencia. Por ejemplo, un vendedor de entradas de cine (taquillero) atiende a los espectadores a medida que se sitúan delante de la ventanilla de entradas. En este caso, esta restricción se representa en el modelo con la palabra *ordered* encerrada entre llaves.

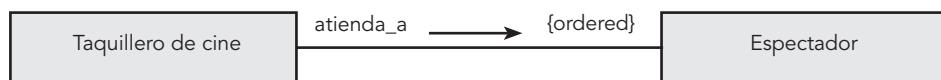


Figura 15.25 Restricciones entre asociaciones.

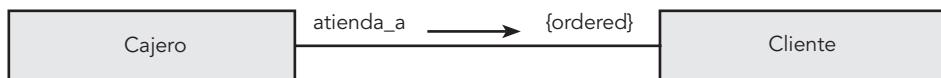


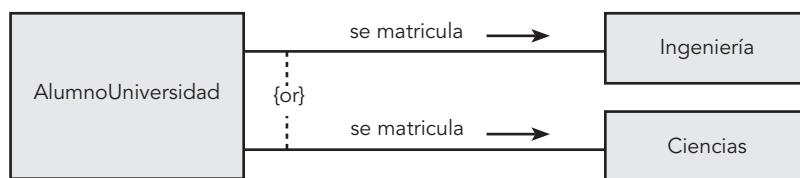
Figura 15.26 Asociación con una restricción (el cajero atiende al cliente por orden de llegada a caja).

Otro tipo de restricciones se pueden presentar y se representan con relaciones *or* o bien *xor*, y se representan gráficamente con una línea de asociación y las palabras *or*, *xor* entre llaves.



### Ejemplo 15.6

Un estudiante se matricula en una universidad en estudios de ingeniería o de ciencias.



Figuras 15.27 Restricción en una asociación.



### Ejemplo 15.7

La relación *xor* implica una u otra asociación y no pueden ser nunca las dos. El caso de una póliza de seguro de una empresa que puede ser o corporativa o de empleado, pero son entre sí, excluyentes.

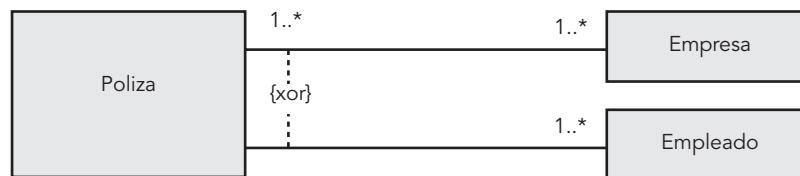


Figura 15.28 Relación *xor* en una asociación.



### Ejemplo 15.8

Un cajero de un banco (humano o electrónico) atiende a clientes. La atención a los clientes se realiza en el orden en que se colocan ante la ventanilla o mostrador, o bien en función del momento de la petición electrónica de acceso al cajero.

### Enlaces

Al igual que un objeto es una instancia de una clase, una asociación también tiene instancia. Por ejemplo la asociación *Juega\_en* y su instancia se muestran en la figura 15.29.

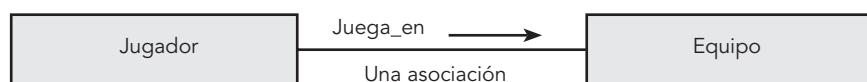


Figura 15.29 Instancia de una asociación.

## 15.4 Agregación

Una **agregación** es un tipo especial de asociación que expresa un acoplamiento más fuerte entre clases. Una de las clases juega un papel importante dentro de la relación con las otras clases. La agregación permite la representación de relaciones como “maestro y esclavo”, “todo y parte de” o “compuesto y componentes”. Los componentes y la clase que constituyen son una asociación que conforma un todo.

Las agregaciones representan conexiones bidireccionales y asimétricas. El concepto de agregación desde un punto de vista matemático es una relación que es transitiva, asimétrica y puede ser reflexiva.



Figura 15.30 Relación de agregación.

La *agregación* es una versión más fuerte que la asociación. Al contrario que la asociación, la agregación implica normalmente propiedad o pertenencia. La agregación se lee normalmente como relación “... posee un ...” o relación “*todo-parte*”, en la cual una clase (“el todo”) representa un gran elemento que consta de elementos más pequeños (“las partes”). La agregación se representa con un rombo a continuación de la clase “propietaria” y una línea recta que apunta a la clase “poseída”. Esta relación se conoce como “*tiene-un*” ya que el todo tiene sus partes; un objeto es parte de otro objeto.

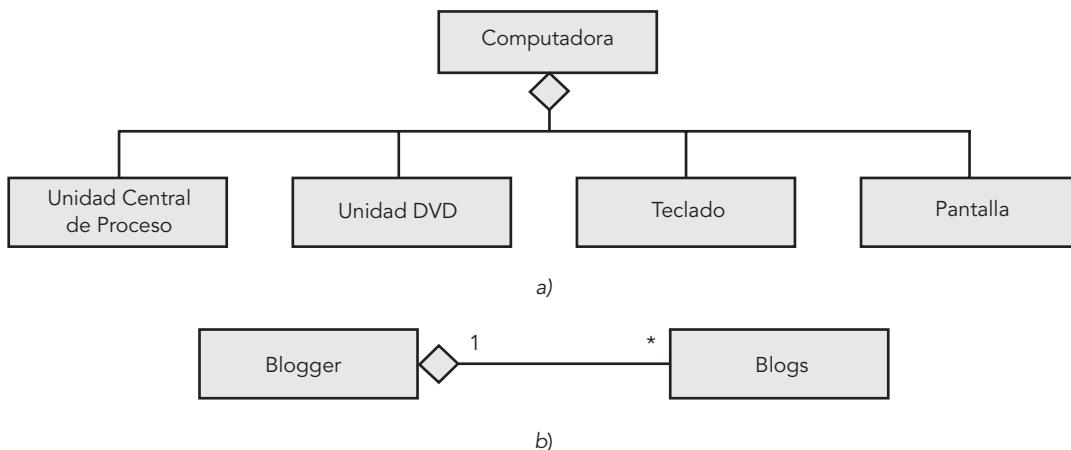


Figura 15.31 Relaciones de agregación: a) Computadora con sus componentes; b) Un Blogger propietario de muchos (\*) Blogs.

Desde el punto de vista conceptual una clase realmente *posee*, pero puede *compartir* objetos de otra clase. Una agregación es un caso especial de asociación. Un ejemplo de una agregación es un automóvil que consta de cuatro ruedas, un motor, un chasis, una caja de cambios, etc. Otro ejemplo es un árbol binario que consta de cero, uno o dos nuevos árboles. Una agregación se representa como una jerarquía con la clase “todo” (por ejemplo, un sistema de computadora) en la parte superior y sus componentes en las partes inferiores (por ejemplo CPU, discos, webcam,...). La representación de la agregación se realiza insertando un rombo vacío en la parte *todo*.

Una computadora es un conjunto de elementos que consta de una unidad central, teclado, ratón, monitor, unidad de CD-ROM, módem, altavoces, escáner, etcétera.

**Ejemplo 15.9**



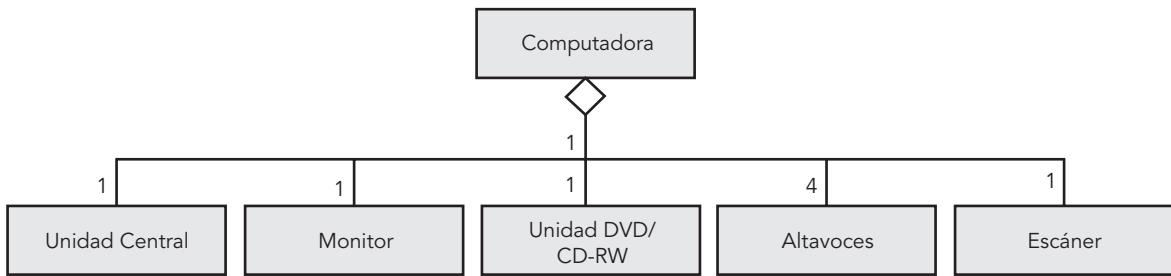


Figura 15.32 Una agregación computadora.

### Restricciones en las agregaciones

En ocasiones el conjunto de componentes posibles en una agregación se establece dentro de una relación O. Así, por ejemplo, el menú del día en un restaurante puede constar de: un primer plato (a elegir entre dos-tres platos), el segundo plato (a elegir entre dos-tres platos) y un postre (a elegir entre cuatro postres). El modelado de este tipo se realiza con la palabra reservada O dentro de llaves con una línea discontinua que conecte las dos líneas que conforman el todo.

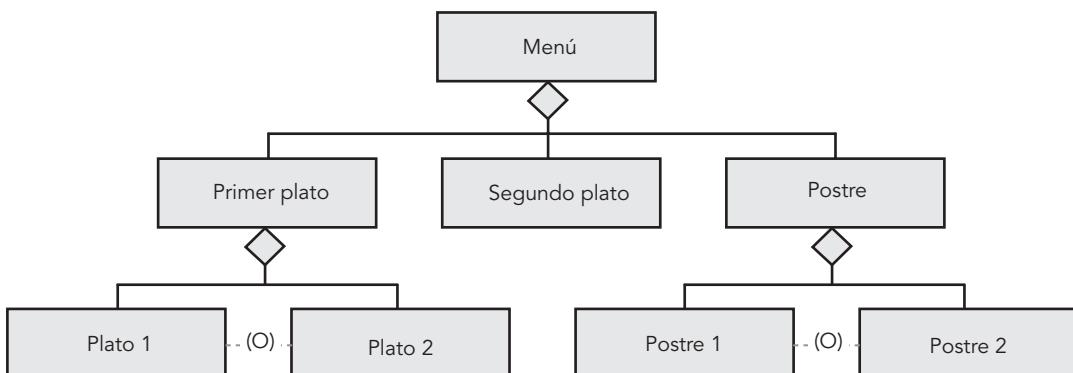


Figura 15.33 Restricción en agregaciones

### Composición

Una *composición* es un tipo especial de agregación que impone algunas restricciones: si el objeto completo se copia o se borra (elimina), sus partes se copian o se suprimen con él. La composición representa una relación fuerte entre clases y se utiliza para representar una *relación todo-parte* (*whole-part*). Cada componente dentro de una composición puede pertenecer tan solo a un todo. El símbolo de una composición es el mismo que el de una agregación, excepto que el rombo está relleno (figura 15.34). Es como una agregación pero con el rombo pintado y no vacío.

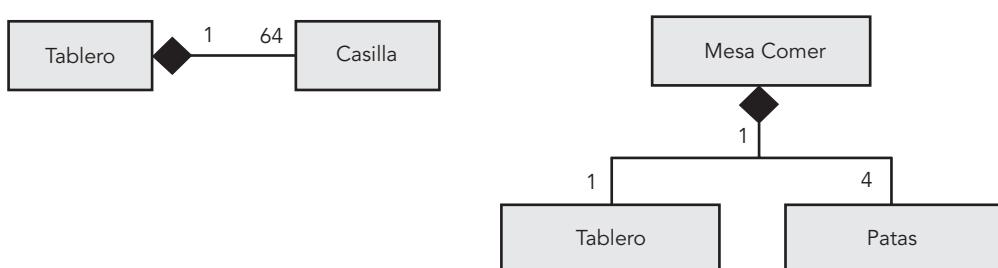


Figura 15.34 Relaciones de composición.

Una relación de composición se lee normalmente como “... es *parte de* ...”, que significa que se necesita leer la composición de la parte al todo. Por ejemplo, si una ventana de una página web tiene una barra de títulos, se puede representar que la clase `BarraTitulo` es *parte de* una clase denominada `Ventana`.

Una mesa para jugar al póker es una composición que consta de una superficie de la mesa y cuatro patas.

### Ejemplo 15.10

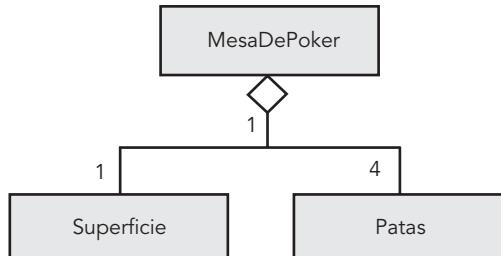


Figura 15.35 Composición.

Un auto tiene un motor que no puede ser parte de otro auto. La eliminación completa del auto supone la eliminación de su motor.

### Ejemplo 15.11

## 15.5 Jerarquía de clases: generalización y especialización

La jerarquía de clases (o clasificaciones) hacen lo posible para gestionar la complejidad ordenando objetos dentro de árboles de clases con niveles crecientes de abstracción. Las jerarquías de clase más conocidas son: **generalización** y **especialización**.

La relación de generalización es un concepto fundamental de la programación orientada a objetos. Una **generalización** es una relación entre un elemento general (llamado *superclase* o “*padre*”) y un caso más concreto de ese elemento (denominado *subclase* o “*hijo*”). Se conoce como *relación es-un* y tiene varios nombres, *extensión*, *herencia*, ... Las clases modelan el hecho de que el mundo real contiene objetos con propiedades y comportamientos. La herencia modela el hecho de que estos objetos tienden a ser organizados en jerarquías. Estas jerarquías representan la relación *es-un*.

La generalización normalmente se lee como “... es un ...” comenzando en la clase específica, derivada o subclase y derivándose a la superclase o *clase base*. Por ejemplo, un `Gato` es-un tipo de `Animal`. La relación de generalización se representa con una línea continua que comienza en la subclase y termina en una flecha cerrada en la superclase.

En UML la relación se conoce como generalización y en programación orientada a objetos como herencia. Al contrario que las relaciones de asociación, las relaciones de generalización no tienen nombre ni ningún tipo de multiplicidad.

Booch, para mostrar las semejanzas y diferencias entre clases, utiliza las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas y pétalos. Se puede constatar que: Una margarita es un tipo (una clase) de flor.

- Una rosa es un tipo (diferente) de flor.
- Las rosas rojas y amarillas son tipos de rosas.
- Un pétalo es una parte de ambos tipos de flores.

Como Booch afirma, las clases y objetos no pueden existir aislados y, en consecuencia, existirán entre ellos relaciones. Las relaciones entre clases pueden indicar alguna forma de compartición, así como algún tipo de conexión semántica. Por ejemplo, las margaritas y las rosas son ambas tipos de

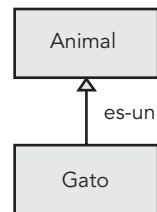


Figura 15.36  
Relaciones de generalización.

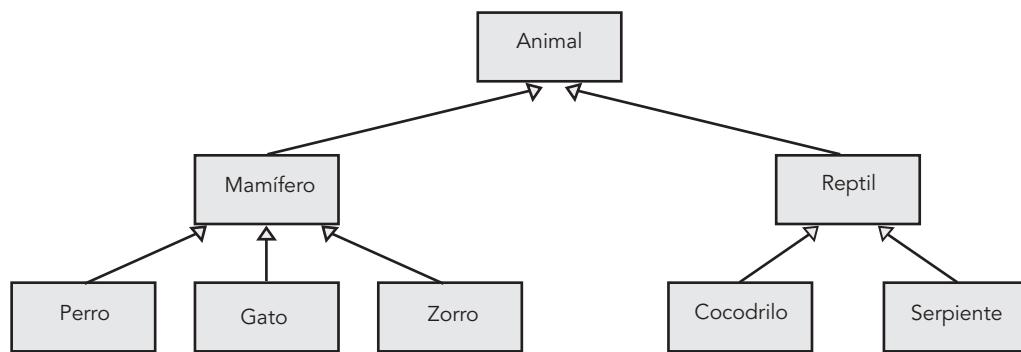


Figura 15.37 Jerarquía de clases.

flores, significando que ambas tienen pétalos coloreados brillantemente, las dos emiten fragancia, etc. La conexión semántica se materializa en el hecho de que las rosas rojas y las margaritas y las rosas están más estrechamente relacionadas entre sí que lo están los pétalos y las flores.

Las clases se pueden organizar en estructuras jerárquicas. La *herencia* es una relación entre clases donde una clase comparte la estructura o comportamiento, definida en una (*herencia simple*) o más clases (*herencia múltiple*). Se denomina *superclase* a la clase de la cual heredan otras clases. De modo similar, una clase que hereda de una o más clases se denomina *subclase*. Una subclase heredará atributos de una superclase más elevada en el árbol jerárquico. La herencia, por consiguiente, define un “tipo” de jerarquía entre clases, en las que una subclase hereda de una o más superclases. La figura 15.37 ilustra una jerarquía de clases **Animal** con dos subclases que heredan de **Animal**, **Mamífero** y **Reptil**.

La herencia es la propiedad por la cual instancias de una clase hija (o subclase) puede acceder tanto a datos como a comportamientos (métodos) asociados con una clase padre (o superclase). La herencia siempre es transitiva, de modo que una clase puede heredar características de superclases de nivel superior. Esto es, si la clase **Perro** es una subclase de la clase **Mamífero** y de **Animal**.

Una vez que una jerarquía se ha establecido es fácil extenderla. Para describir un nuevo concepto no es necesario describir todos sus atributos. Basta describir sus diferencias a partir de un concepto de una jerarquía existente. La herencia significa que el comportamiento y los datos asociados con las clases hija son siempre una extensión (esto es, un conjunto estrictamente más grande) de las propiedades asociadas con las clases padres. Una subclase debe tener todas las propiedades de la clase padre y otras. El proceso de definir nuevos tipos y reutilizar código anteriormente desarrollado en las definiciones de la clase base se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden, a su vez, servir como clases base de otras clases. Esta jerarquía de tipos normalmente toma la estructura de árbol, conocido como *jerarquía de clases* o *jerarquía de tipos*.

La jerarquía de clases es un mecanismo muy eficiente, ya que se pueden utilizar definiciones de variables y métodos en más de una subclase sin duplicar sus definiciones. Por ejemplo, consideremos un sistema que representa varias clases de vehículos manejados por humanos. Este sistema contendrá una clase genérica de vehículos, con subclases para todos los tipos especializados. La clase **Vehículo** contendrá los métodos y variables que fueran propios de todos los vehículos, es decir, número de matrícula, número de pasajeros, capacidad del depósito de combustible. La subclase, a su vez, contendrá métodos y variables adicionales específicos.

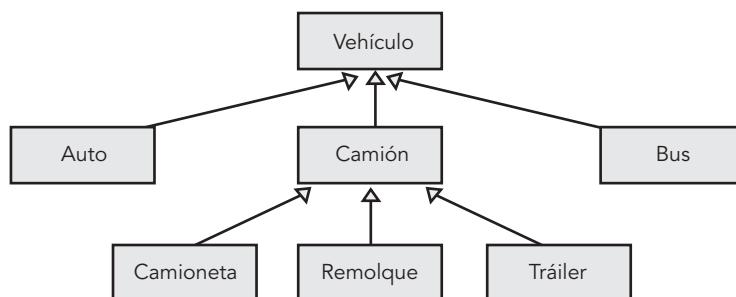


Figura 15.38 Subclases de la clase **Vehículo**.

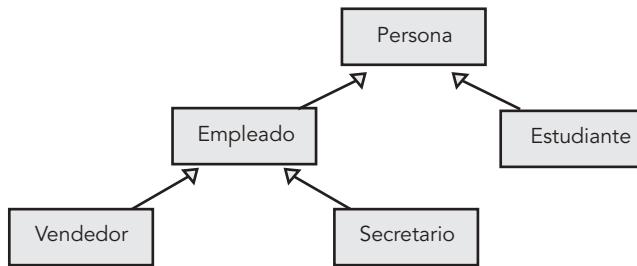


Figura 15.39 Una jerarquía Persona.

La flexibilidad y eficiencia de la herencia no es gratuita; se emplea tiempo en buscar una jerarquía de clases para encontrar un método o variable, de modo que un programa orientado a objetos puede correr más lentamente que su correspondiente convencional. Sin embargo, los diseñadores de lenguajes han desarrollado técnicas para eliminar esta penalización en velocidad en la mayoría de los casos, permitiendo a las clases enlazar directamente con sus métodos y variables heredados, de modo que no se requiera realmente ninguna búsqueda.

### Regla

- Cada objeto es una instancia de una clase.
- Algunas clases, abstractas, no pueden instanciar directamente.
- Cada enlace es una instancia de una asociación.

## Jerarquías de generalización/especialización

Las clases con propiedades comunes se organizan en superclases. Una **superclase** representa una *generalización* de las subclases. De igual modo, una subclase de una clase dada representa una *especialización* de la clase superior (figura 15.40). La *clase derivada* es un tipo de clase de la clase base o superclase.

Una superclase representa una *generalización* de las subclases. Una subclase de la clase dada representa una *especialización* de la clase ascendente (figura 15.41).

En la *modelización* o *modelado* orientado a objetos es útil introducir clases en un cierto nivel que puede no existir en la realidad, pero que son construcciones conceptuales útiles. Estas clases se conocen

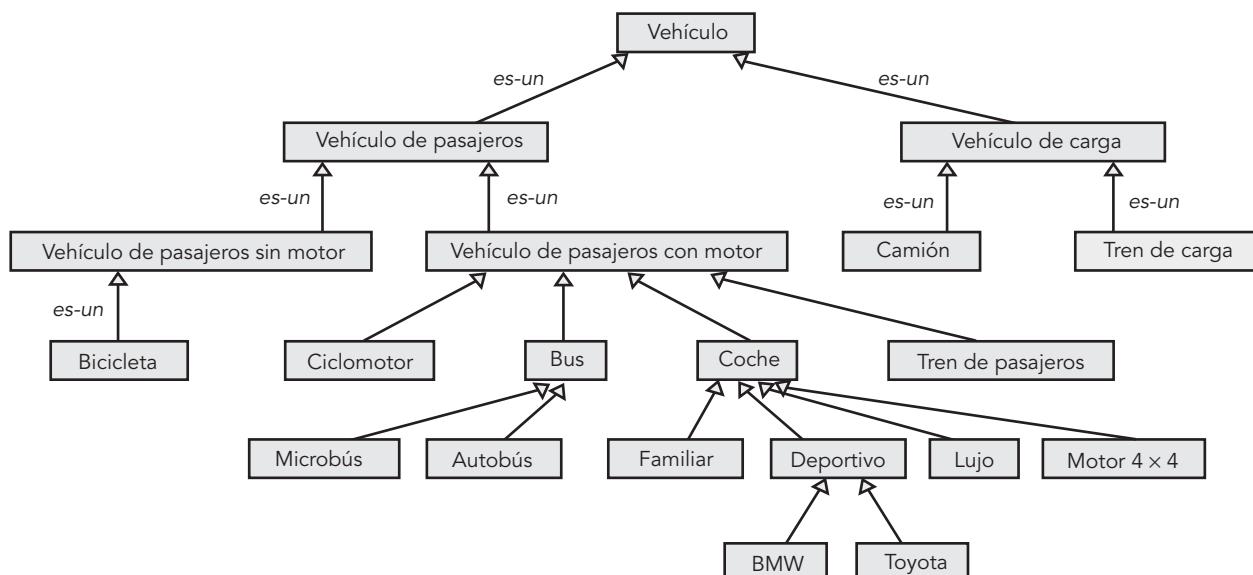


Figura 15.40 Relaciones de generalización.

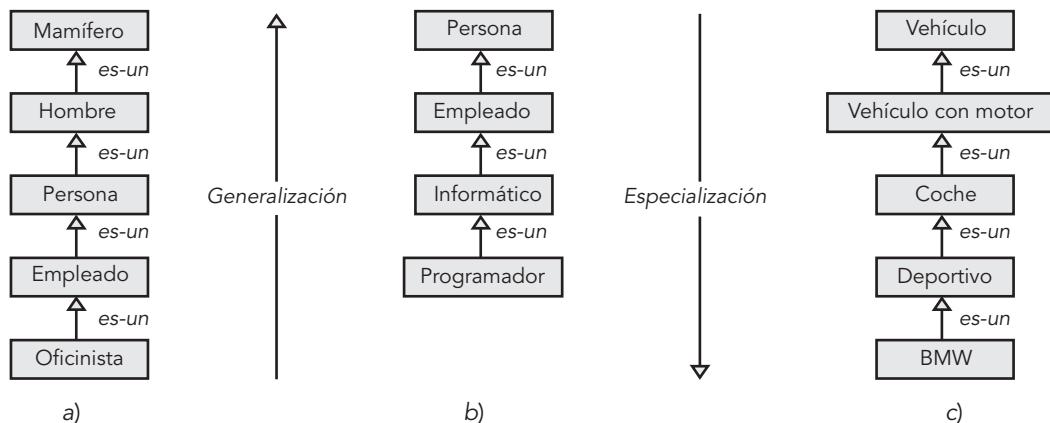


Figura 15.41 Relaciones de jerarquía es-un (is-a).

como **clases abstractas** y su propiedad fundamental es que no se pueden crear instancias de ellas. Ejemplos de clases abstractas son *vehículo de pasajeros* y *vehículo de mercancías*. Por otra parte, de las subclases de estas clases abstractas, que corresponden a los objetos del mundo real, se pueden crear instancias directamente por sí mismas. Por ejemplo, de *BMW* se pueden obtener dos instancias, *Coche1* y *Coche2*.

La generalización, en esencia, es una abstracción en que un conjunto de objetos de propiedades similares se representa mediante un objeto genérico. El método usual para construir relaciones entre clases es definir generalizaciones buscando propiedades y funciones de un grupo de tipos de objetos similares, que se agrupan juntos para formar un nuevo tipo genérico. Consideremos el caso de empleados de una compañía que pueden tener propiedades comunes (nombre, número de empleado, dirección, etc.) y funciones comunes (calcular\_nómina), aunque dichos empleados pueden ser muy diferentes en atención a su trabajo: oficinistas, gerentes, programadores, ingenieros, etc. En este caso, lo normal será crear un objeto genérico o superclase *Empleado*, que definirá una clase de empleados individuales.

Por ejemplo, *Analistas*, *Programadores* y *Operadores* se pueden generalizar en la clase *informático*. Un programador determinado (*Mortimer*) será miembro de las clases *Programador*, *informático* y *Empleado*; sin embargo, los atributos significativos de este programador variarán de una clase a otra.

La jerarquía de generalización/especialización tiene dos características fundamentales y notables. Primero, un tipo objeto no desciende más que de un tipo objeto genérico; segundo, los descendientes inmediatos de cualquier nodo no necesitan ser objetos de clases exclusivas mutuamente. Por ejemplo, los gerentes y los informáticos no tienen por qué ser exclusivos mutuamente, pero pueden ser tratados como dos objetos distintos; es el tipo de relación que se denomina *generalización múltiple*.

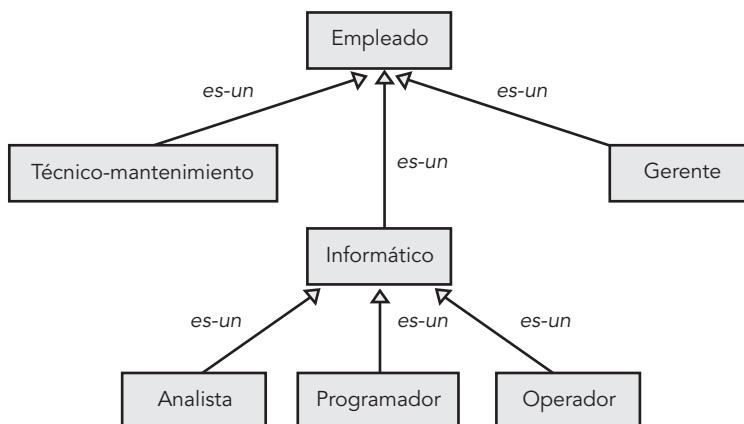


Figura 15.42 Una jerarquía de generalización de empleados.

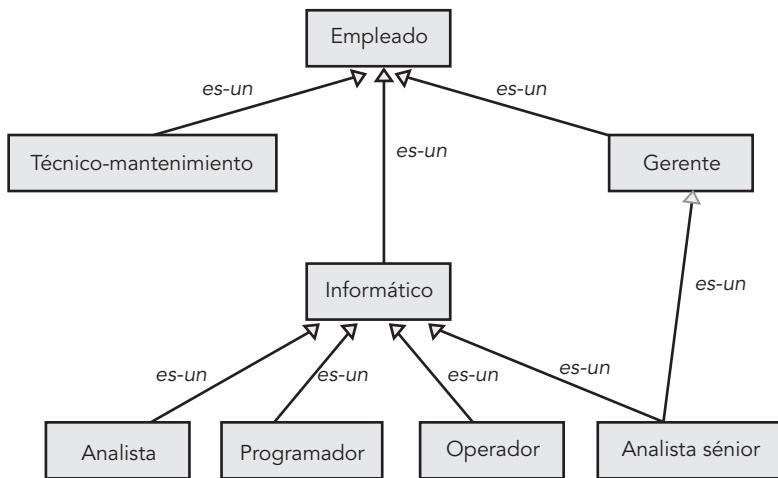


Figura 15.43 Una jerarquía de generalización múltiple.

### A recordar

UML define a la generalización como herencia. De hecho, generalización es el concepto y herencia se considera la implementación del concepto en un lenguaje de programación.

### Síntesis de generalización/Especialización [Muller 97]

1. La generalización es una relación de herencia entre dos elementos de un modelo como clase. Permite a una clase heredar atributos y operaciones de otra clase. En realidad es la factorización de elementos comunes (atributos, operaciones y restricciones) dentro de un conjunto de clases en una clase más general denominada **superclase**. Las clases están ordenadas dentro de una jerarquía; una superclase es una abstracción de sus subclases.
2. La flecha que representa la generalización entre dos clases apunta hacia la clase más general.
3. La especialización permite la captura de las características específicas de un conjunto de objetos que no han sido distinguidos por las clases ya identificadas. Las nuevas características se representan por una nueva clase, que es una subclase de una de las clases existentes. La especialización es una técnica muy eficiente para extender un conjunto de clases de un modo coherente.
4. La generalización y la especialización son dos puntos de vista opuestos del concepto de jerarquía de clasificación; expresan la dirección en que se extiende la jerarquía de clases,
5. Una generalización no lleva ningún nombre específico; siempre significa “es un tipo de”, “es un”, “es uno de”, etc. La generalización solo pertenece a clases, no se puede instanciar vía enlaces y por consiguiente no soporta el concepto de multiplicidad.
6. La generalización es una relación no reflexiva: una clase no se puede derivar de sí misma.
7. La generalización es una relación asimétrica: si la clase B se deriva de la clase A, entonces la clase A no se puede derivar de la clase B.
8. La generalización es una relación transitiva: si la clase C se deriva de la clase B que a su vez se deriva de la clase A, entonces la clase C se deriva de la clase A.

## 15.6 Herencia: clases derivadas

Como ya se ha comentado, la herencia es la manifestación más clara de la relación de generalización/especialización y a la vez una de las propiedades más importantes de la orientación a objetos y posiblemente su característica más conocida y sobresaliente. Todos los lenguajes de programación orientados a objetos soportan directamente en su propio lenguaje construcciones que implementan de modo directo la relación entre clases derivadas.

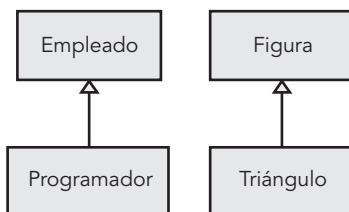


Figura 15.44 Clases derivadas.

La **herencia** o relación **es-un** es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una clase que representa un concepto más general; en este caso la nueva clase puede *heredar* de la clase ya existente. Así, por ejemplo, si existe una clase **Figura** y se desea crear una clase **Triángulo**, esta clase **Triángulo** puede derivarse de **Figura** ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. **Triángulo** *es-un* tipo de **Figura**. Otro ejemplo, puede ser **Programador** que *es-un* tipo de **Empleado**.

### Herencia simple

La implementación de la generalización es la herencia. Una clase hija o subclase puede heredar atributos y operaciones de otra clase padre o superclase. Las clases padre es más general que la clase hija. Una clase hija puede ser, a su vez, una clase padre de otra clase hija. **Mamífero** es una clase derivada de **Animal** y **Caballo** es una clase hija o derivada de **Mamífero**.

En UML, la herencia se representa con una línea que conecta la clase padre con la clase hija. En la parte de la línea que conecta a la clase padre se pone un triángulo abierto (punta de flecha) que apunta a dicha clase padre. Este tipo de conexión se representa como “*es un tipo de*”. **Caballo** *es-un-tipo de* **Mamífero** que a su vez es un tipo de **Animal**.

Una clase puede no tener padre en cuyo caso se denomina *clase base* o *clase raíz*, y también puede no tener ninguna clase hija, en cuyo caso se denomina *clase terminal* o *clase hija*.

Si una clase tiene exactamente un parente, tiene **herencia simple**. Si una clase tiene más de un parente, tiene **herencia múltiple**.

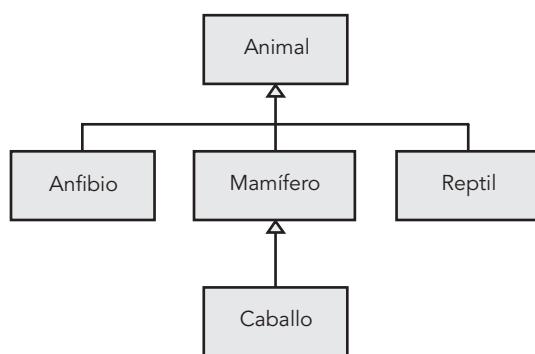


Figura 15.45 Herencia simple con dos niveles.

### Herencia múltiple

**Herencia múltiple** o **generalización múltiple**, en terminología oficial de UML, se produce cuando una clase hereda de dos o más clases padres (figura 15.46).

Aunque la herencia múltiple está soportada en UML y en C++ (no en Java), en general, su uso no se considera una buena práctica en la mayoría de los casos. Esta característica se debe al hecho de que la herencia múltiple presenta un problema complicado cuando las dos clases padre tienen solapamiento de atributos y comportamientos. ¿A qué se debe la complicación? Normalmente a conflictos de atributos o

propiedades derivadas. Por ejemplo, si las clases A1 y A2 tienen el mismo atributo nombre, la clase hija de ambas A1-2, ¿de cuál de las dos clases hereda el atributo?

C++, que soporta herencia múltiple, debe utilizar un conjunto propio de reglas del lenguaje C++ para resolver estos conflictos. Estos problemas conducen a malas prácticas de diseño y ha hecho que lenguajes de programación como **Java** y **C#** no admiten herencia múltiple. Sin embargo como C++ admite esta característica, UML incluye en sus representaciones este tipo de herencia.

### Regla

La generalización es una relación “es-un”. (un **Carro** es-un **Vehículo**; un **Gerente** es-un **Empleado**, etc.). También se utiliza “es-un-tipo-de” (*is a kind of*).

En orientación a objetos la relación “es-un” se conoce como *herencia* y en UML como *generalización*.

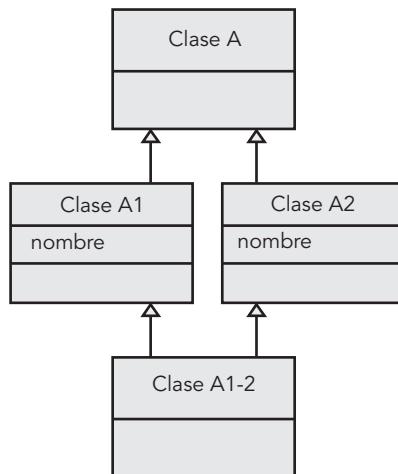


Figura 15.46 Herencia múltiple en la clase A1-2.

### Niveles de herencia

La jerarquía de herencia puede tener más de dos niveles. Una clase hija puede ser una clase padre, a su vez, de otra clase hija. Así, una clase **Mamífero** es una clase hija de **Animal** y una clase padre de **Caballo**.

Las clases hija o subclases añaden sus propios atributos y operaciones a los de sus clases base. Una clase puede no tener clase hija, en cuyo caso es una *clase hija*. Si una clase tiene solo un parente, se tiene *herencia simple* y si tiene más de un parente, entonces se tiene *herencia múltiple*.

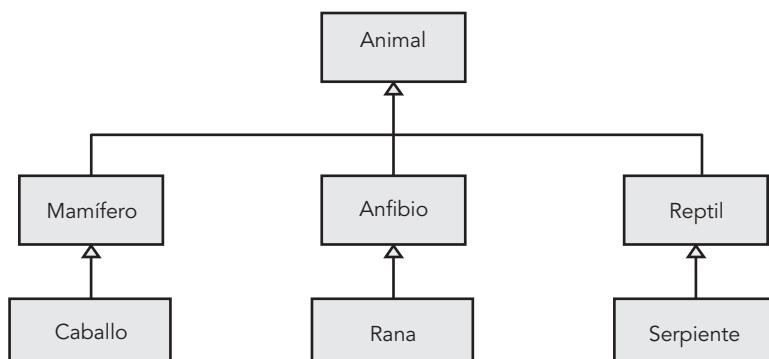


Figura 15.47 Dos niveles en una jerarquía de herencia simple.



### Ejemplo 15.12

Representaciones gráficas de la herencia.

1. Vehículo es una *superclase* (clase base) y tiene como clases derivadas (subclase) Coche (Carro), Barco, Avión y Camión. Se establece la jerarquía Vehículo, que es una jerarquía *generalización-especialización*.

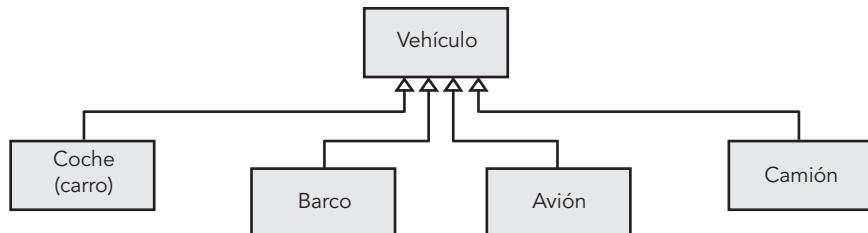


Figura 15.48 Diagrama de clases de la jerarquía Vehículo.

2. Jerarquía Vehículo (segunda representación gráfica, tipo árbol).

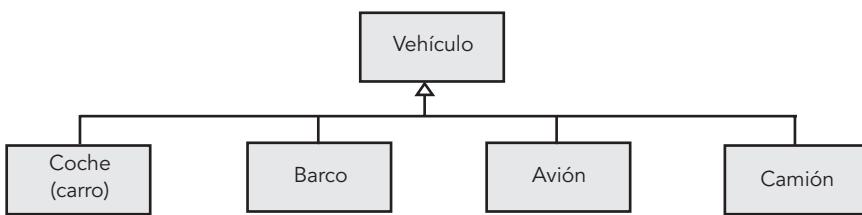


Figura 15.49 Jerarquía Vehículo en forma de árbol.

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el *especificador de acceso* que indica el tipo de herencia (*pública*, *privada* y *protegida*). La primera línea de cada declaración debe incluir el formato siguiente:

```
clase nombre_clase hereda_de tipo_herencia nombre_clase_base
```

#### Regla

En general, se debe incluir la palabra reservada *pública* en la primera línea de la declaración de la clase derivada, y representa *herencia pública*. Esta palabra reservada produce que todos los miembros que son públicos en la clase base permanecen públicos en la clase derivada.



### Ejemplo 15.13

Declaración de las clases Programador y Triangulo.

1. 

```
clase Programador hereda_de Empleado
 publica:
 // miembros públicos
 privada:
 // miembros privados
 fin_clase
```

```
2. clase Triangulo hereda_de Figura
 publica:
 // sección pública
 ...
 privado:
 // sección privada
 ...
```

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

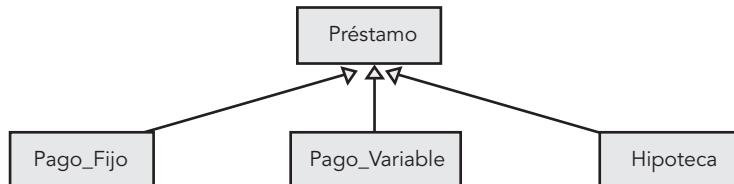
```
clase derivada clase base
 ↙ ↙
clase Director hereda_de Empleado
 publica:
 nuevas funciones miembro
 privada:
 nuevos miembros dato
 fin_clase
```

En la definición de la clase Director solo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase Empleado son heredados automáticamente por la clase Director. Por ejemplo, la función calcular\_salario de Empleado se aplica automáticamente a los directores:

```
Director d;
d.calcular_salario(325000);
```

Considérese una clase Prestamo y tres clases derivadas de ella: Pago\_fijo, Pago\_variable e Hipoteca.

### Ejemplo 15.14



```
clase Prestamo
 protegida:
 real capital;
 real tasa_interes;
 publica:
 Prestamo(float, float);
 entero crearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS] = 0;
 fin_clase
```

Las variables capital y tasa\_interes no se repiten en la clase derivada.

```
clase Pago_fijo hereda_de Prestamo
 privada:
 real pago; // cantidad mensual a pagar por cliente
 publica:
 Pago_Fijo (float, float, float);
 entero CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS]);
 };

clase Hipoteca hereda_de Prestamo
 privada:
```

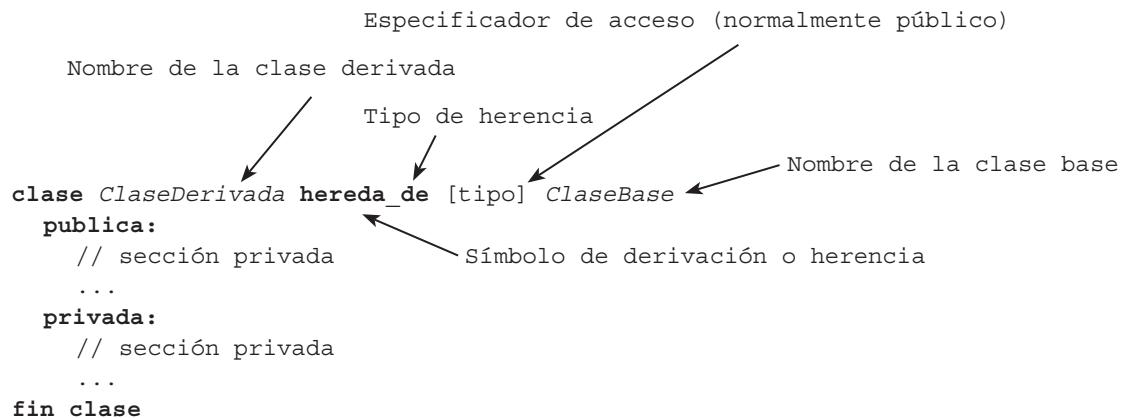
```

 entero num_recibos;
 entero recibos_por_anho;
 real pago;
publica:
 Hipoteca(int, int, float, float, float);
 entero CrearTablaPagos (flota [MAX_TERN] [NUM_COLUMNAS]);
fin_clase

```

## Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:



*Especificador de acceso* `publica` significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

*Herencia pública* es aquella en que el especificador de acceso es `publico` (*público*).

*Herencia privada* es aquella en que el especificador de acceso es `privado` (*privado*).

*Herencia protegida* es aquella en que el especificador de acceso es `protegido` (*protegido*).

El especificador de acceso que declara el tipo de herencia es opcional (`publica`, `privada` o `protegida`); si se omite el especificador de acceso, se considera por defecto (predeterminada) `privada`. La *clase base* (`ClaseBase`) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```

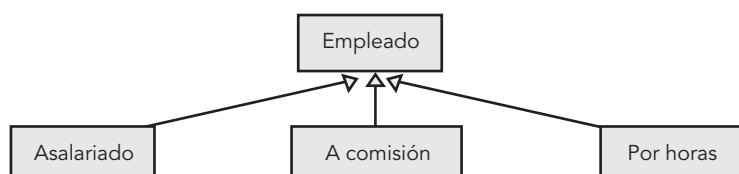
class nombre_clase hereda_de [especificador_acceso] ClaseBase
 lista_de_miembros;
fin_clase

```

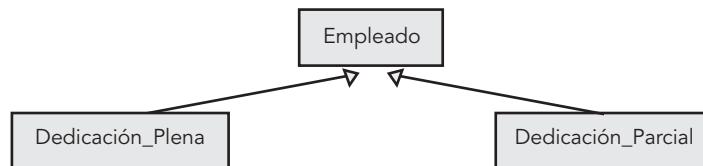
## Consideraciones de diseño

A veces es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

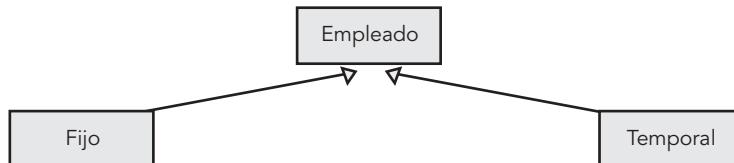
Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas.



Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fijo o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones?, ¿esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada solo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

## 15.7 Accesibilidad y visibilidad en la herencia

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los elementos privados son accesibles solo a los miembros de la clase en que están definidos y los elementos protegidos pueden ser accedidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior existen tres tipos de herencia: *pública*, *privada* y *protegida*. Normalmente el tipo de herencia más utilizada es la herencia pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

### Norma

Por defecto en C++ la herencia es privada. Si accidentalmente se olvida la palabra reservada `public`, los elementos de la clase base serán inaccesibles. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

Tabla 15.3 / Acceso a variables y funciones según tipo de herencia.

| Tipo de herencia | Tipo de elemento | ¿Accesible a clase derivada? |
|------------------|------------------|------------------------------|
| Publica          | publica          | sí                           |
|                  | protegida        | sí                           |
|                  | privada          | no                           |
| Privada          | publica          | no                           |
|                  | protegida        | no                           |
|                  | privada          | no                           |

## Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y privados de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos. La herencia pública se representa con el especificador `publica` en la derivación de clases.

### Formato

```
clase ClaseDerivada hereda_de publica Clase Base
 publica:
 // sección pública
 privada:
 // sección privada
fin_clase
```

## Herencia privada

La herencia privada significa que una clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```
clase ClaseDerivada hereda_de privada ClaseBase
 publica:
 // sección pública
 protegida:
 // sección protegida
 privada:
 // sección privada
fin_clase
```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Con independencia de la visibilidad de la herencia, los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada.

La herencia privada se utiliza con menos frecuencia que la herencia pública. Este tipo de herencia oculta la clase base del usuario y así es posible cambiar la implementación de la clase base o eliminarla toda junta sin requerir ningún cambio al usuario de la interfaz. En C++, cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

## Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que el usuario de la clase ve. La herencia protegida es todavía menos frecuente que la herencia privada.

La tabla 15.4 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.



### Ejercicio 15.3

Declarar una clase base (`Base`) y tres clases derivadas de ella, `D1`, `D2` y `D3`

```
clase Base {
 publica:
 entero i1;
 protegida:
 entero i2;
```

```

 privada:
 entero i3;
};

clase D1 : privada Base {
 nada f();
};

clase D2 : protegida Base {
 nada g();
};

clase D3 : publica Base {
 nada h();
};

```

Ninguna de las subclases tienen acceso al miembro `i3` de la clase `Base`. Las tres clases pueden acceder a los miembros `i1` e `i2`. En la definición de la función miembro `f()` se tiene:

```

void D1::f() {
 i1 = 0; // Correcto
 i2 = 0; // Correcto
 i3 = 0; // Error
};

```

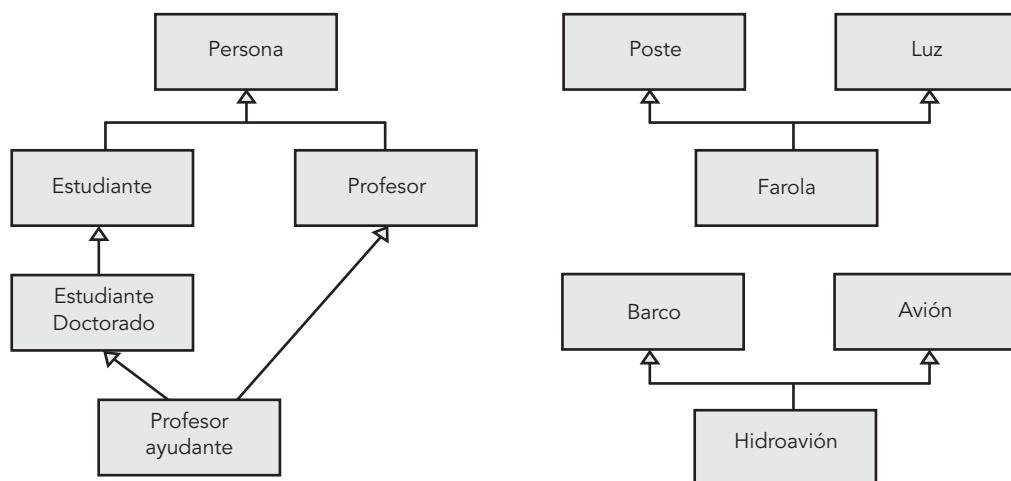
**Tabla 15.4** Tipos de herencia y accesos que permiten.

| Tipo de herencia | Acceso a miembro clase base | Acceso a miembro a clase derivada |
|------------------|-----------------------------|-----------------------------------|
| <b>publica</b>   | publica                     | publica                           |
|                  | protegida                   | protegida                         |
|                  | privada                     | inaccesible                       |
| <b>protegida</b> | publica                     | protegida                         |
|                  | protegida                   | protegida                         |
|                  | privada                     | inaccesible                       |
| <b>privada</b>   | publica                     | privada                           |
|                  | protegida                   | privada                           |
|                  | privada                     | inaccesible                       |

## 15.8 Un caso de estudio especial: herencia múltiple

**Herencia múltiple** es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes* o *padres*) para la clase derivada (*descendiente* o *hija*).

La herencia múltiple entraña un concepto más complicado que la herencia simple, no solo con respecto a la sintaxis sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la figura 15.50 se muestran diferentes ejemplos de herencia múltiple.



**Figura 15.50** Ejemplos de herencia múltiple.

## Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene solo un parente). Herencia múltiple implica múltiples clases bases (una clase derivada tiene varios padres).

En herencia simple, el escenario es bastante sencillo, en términos de concepto y de implementación. En herencia múltiple los escenarios varían ya que las clases bases pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Por qué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en la base del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tomen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
clase Cderivada hereda_de Base1, Base2, ...
publica:
 // sección pública
privada:
 // sección privada
...
fin_clase
```

Funciones o datos miembro que tengan el mismo nombre en `Base1`, `Base2`, `Basen`, ... serán motivo de ambigüedad.

```
clase A hereda de publica B, C {...}
clase D hereda de publica E, F, publica G {...}
```

La palabra reservada `public` ya se ha comentado anteriormente, define la relación “`es-un`” y crea un subtipo para herencia simple. Así, en los ejemplos anteriores, la clase `A` “`es-un`” tipo de `B` y “`es-un`” tipo de `C`. La clase `D` se deriva públicamente de `E` y `G` y privadamente de `F`. Esta derivación hace a `D` un subtipo de `E` y `G` pero no un subtipo de `F`. *El tipo de acceso sólo se aplica a una clase base*.

```
clase Derivada hereda de publica Base1 Base2 { } ;
```

Derivada específica derivación pública de Basel y derivación privada (por defecto u omisión) de BasCo3

### Regla

Asegúrese de especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente `privada` cuando lo necesite para manejar la legibilidad.

```
Class Derivada : publica Base1, privada Base2 { ... }
```

### Ejemplo 15.15

```
clase Estudiante {
 ...
};

clase Trabajador {
 ...
};

clase Estudiante_Trabajador: publica Estudiante, publica Trabajador {
 ...
};
```

## Características de la herencia múltiple

La herencia múltiple plantea diferentes problemas como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

### Ambigüedades

Al contrario que la herencia simple, la herencia múltiple tiene el problema potencial de las ambigüedades.

### Ejemplo 15.16

```
clase Ventana {
 privada:
 ...
 publica:
 nada dimensionar(); // dimensiona una ventana
 ...
};

clase Fuente{
 privada:
 ...
 publica:
 nada dimensionar(); // dimensiona un tipo fuente
 ...
}; fin de clase
```

Una clase `Ventana` tiene una función `dimensionar()` que cambia el tamaño de la ventana; de modo similar, una clase `Fuente` modifica los objetos `Fuente` con `dimensionar()`. Si se crea una clase `Ventana_Fuente` (`VFuente`) con herencia múltiple, se puede producir ambigüedad en el uso de `dimensionar()`.

```
clase VFuente: publica Ventana, publica Fuente {...};
VFuente v;
v.dimensionar(); // se produce un error ¿cuál?
```

La llamada a `dimensionar` es ambigua, ya que el compilador no sabrá a qué función `dimensionar` ha de llamar. Esta ambigüedad se resuelve fácilmente en C++ con el operador de resolución de ámbito (`::`).

```
v.Fuente::dimensionar(); // llamada a dimensionar() de Fuente
v.Ventana::dimensionar(); // llamada a dimensionar de Ventana
```

### Precaución

No es un error definir un objeto derivado con multiplicidad con ambigüedades. Estas se consideran ambigüedades potenciales y solo produce errores en tiempo de compilación cuando se llaman de modo ambiguo.

### Regla

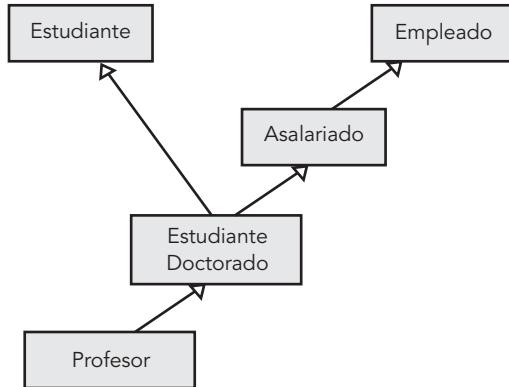
Incluso es mejor solución que la citada anteriormente resolver la ambigüedad en las propias definiciones de la función `dimensionar()`.

```
clase VFuente: publica Ventana, publica Fuente
...
void v_dimensionar() { Ventana::dimensionar(); }
void f_dimensionar() { Fuente::dimensionar(); }
fin_clase
```



### Ejemplo 15.17

Diseñar e implementar una jerarquía de clases que represente las relaciones entre las clases siguientes: estudiante, empleado, empleado asalariado y un estudiante de doctorado que es a su vez profesor de prácticas de laboratorio.



### Nota

Se deja la resolución como ejercicio al lector.

## 15.9 Clases abstractas

Una *clase abstracta* es una clase que no tiene ningún objeto o con mayor precisión, es una clase que no puede tener objetos instancias de la clase base. Una clase abstracta describe atributos y comportamientos comunes a otras clases, y deja algunos aspectos del funcionamiento de la clase a las subclases concretas. Una clase abstracta se representa con su nombre en cursiva.

11

### Ejercicio 15.4

Clase abstracta *Vehículo* con clases derivadas *Coche* (Carro) y *Barco*.

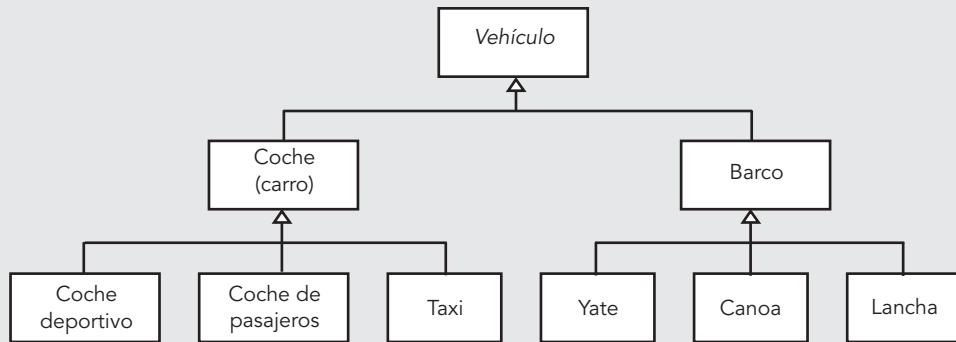


Figura 15.51 Jerarquía con clase base abstracta *Vehículo*.

Una clase abstracta se representa poniendo su nombre en cursiva o añadiendo la palabra `{abstract}` dentro del compartimento de la clase y debajo del nombre de la clase.

11

### Ejercicio 15.5

Clase abstracta *Futbolista* de la cual derivan las clases concretas *Portero*, *Defensa* y *Delantero*.

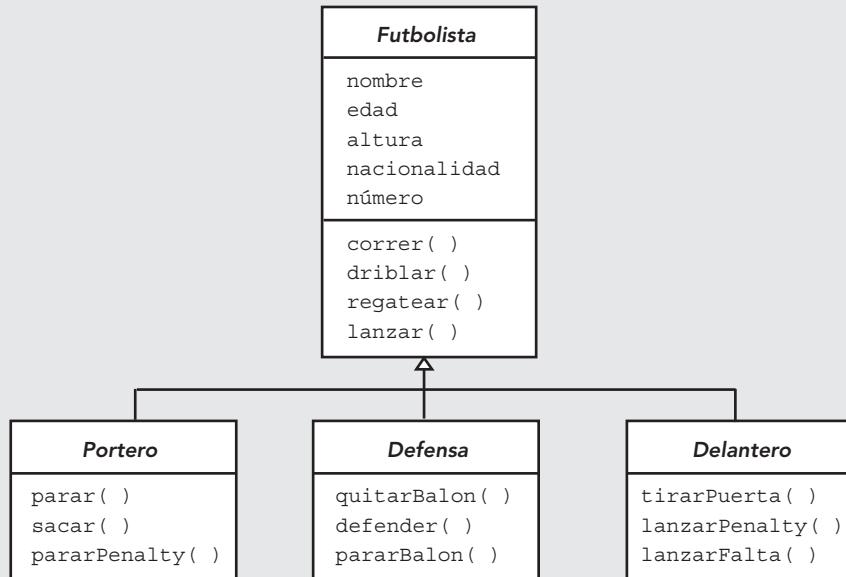


Figura 15.52 Jerarquía con clase base abstracta *Futbolista*.

## Operaciones abstractas

Una clase abstracta tiene operaciones abstractas. Una *operación abstracta* no tiene implementación de métodos, solo la signatura o prototipo. Una clase que tiene al menos una operación abstracta es por definición, abstracta.

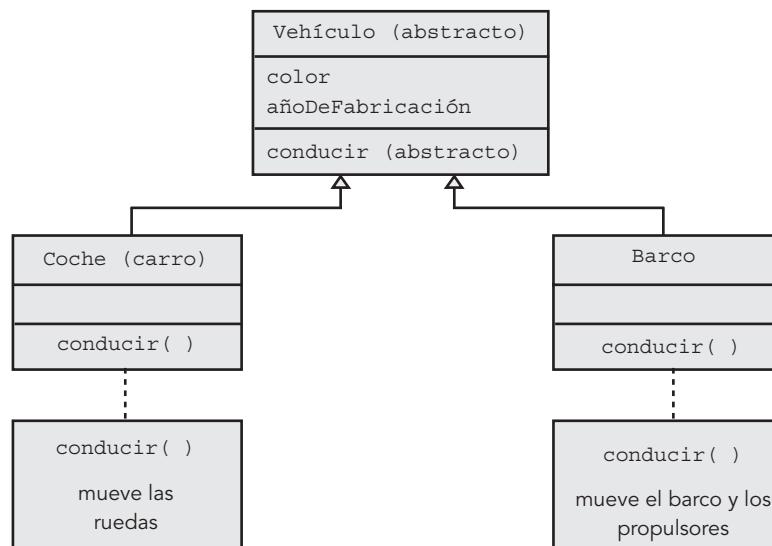
Una clase que hereda de una clase que tiene a su vez una o más operaciones abstractas debe implementar esas operaciones (proporcionar métodos para esas operaciones). Las operaciones abstractas se muestran con la cadena `{abstract}` a continuación del prototipo o signatura de la clase. Las operaciones abstractas se definen en las clases abstractas para especificar el comportamiento que deben tener todas las subclases. Una clase `Vehículo` debe tener operaciones abstractas que especifiquen comportamientos comunes de todos los vehículos (conducir, frenar, arrancar, ...).

Los modeladores suelen proporcionar siempre una capa de clases abstractas como superclases, buscando elementos comunes a cualquier relación de herencia que se puede extender a las clases hijas. En el ejemplo de las clases abstractas, `Coche` y `Barco` representan a clases que requieren implementar las operaciones abstractas “*conducir*” y “*frenar*”.

Una *clase concreta* es una clase opuesta a la clase abstracta. En una clase concreta, es posible crear objetos de la clase que tienen implementaciones de todas las operaciones. Si la clase `Vehículo` tiene especificada una operación abstracta *conducir*, tanto las clases `Coche` como `Barco` deben implementar ese método (o las propias operaciones deben ser especificadas como abstractas). Sin embargo, las implementaciones son diferentes. En un coche, la operación *conducir* hace que las ruedas se muevan; mientras que conducir un barco hace que el barco navegue (se mueva). Las subclases heredan operaciones de una superclase común, pero dichas operaciones se implementan de modo diferente.

Una subclase puede redefinir (modificar el comportamiento de la superclase) las operaciones de la superclase, o bien la heredan de la superclase tal y como está definida. Una operación redefinida debe tener la misma signatura o prototipo (tipo de retorno, nombre y parámetros) que la superclase. La operación que se está redefiniendo puede ser o bien *abstracta* (no tiene implementación en la superclase) o *concreta* (tiene una implementación en la superclase). En cualquier caso, la redefinición en las subclases se utiliza para todas las instancias de esa clase.

Se pueden añadir a las subclases nuevas operaciones, atributos y asociaciones. Un objeto de una subclase se puede utilizar en cualquier situación donde sea posible utilizar objetos de la superclase. En ese caso, la subclase tendrá una implementación diferente dependiendo del objeto implicado.



**Figura 15.53** La clase `Vehículo` (abstracta) define los atributos `color` y `añoDeFabricación`, y la operación `conducir`.



## Resumen

Una asociación es una conexión semántica entre clases. Una asociación permite que una clase conozca los atributos y operaciones públicas de otra clase.

Una *agregación* es una relación más fuerte que una asociación y representa una clase que se compone de otras clases. Una agregación representa la relación todo-parte; es decir una clase es el todo y contiene a todas las partes.

Una *generalización* es una relación de herencia entre dos elementos de un modelo como clases. Permite a una clase heredar atributos y operaciones de otra clase. Su implementación en un lenguaje orientado a objetos es la herencia. La *especialización* es la relación opuesta a la generalización.

La relación **es-un** representa la herencia. Por ejemplo, una rosa es un tipo de flor; un pastor alemán es un tipo de perro, etc. La relación es-un es transitiva. Un pastor alemán es un tipo de perro y un perro es un tipo de mamífero; por consiguiente, un pastor alemán es un mamífero. Una clase nueva que se crea a partir de una clase ya existente,

utilizando herencia se denomina clase derivada o subclase. La clase padre se denomina clase base o superclase.

1. *Herencia* es la capacidad de derivar una clase de otra clase. La clase inicial utilizada por la clase derivada se conoce como *clase base, padre* o *superclase*. La clase derivada se conoce como *derivada, hija* o *subclase*.
2. *Herencia simple* es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. Las relaciones de herencia reducen código redundante en programas. Uno de los requisitos para que un lenguaje sea considerado orientado a objetos es que soporte herencia.
3. La *herencia múltiple* se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflicto de nombres, cosa que se produce cuando nombres idénticos aparecen en más de una clase base.



## Ejercicios

15.1 Definir una clase base *Persona* que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.) Diseñar una jerarquía de clases que contemple las clases siguientes: *Estudiante, Empleado, Estudiante\_Emppleado*. Escribir un programa que lea un archivo de información y cree una lista de personas: *a) general; b) estudiantes; c) empleados; d) estudiantes empleados*. El programa deberá permitir ordenar alfabéticamente por el primer apellido.

15.2 Implementar una jerarquía *Librería* que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etcétera.

15.3 Diseñar una jerarquía de clases que utilice como clase base o raíz una clase *LAN* (red de área local). Las subclases derivadas deben representar diferentes topologías, como *estrella, anillo, bus y hub*. Los miembros datos deben representar propiedades como *soporte de transmisión, control de acceso, formato del marco de datos, estándares, velocidad de transmisión*, etc. **Se desea simular la actividad de los nodos de tal LAN.**

La red consta de **nodos**, que pueden ser dispositivos como computadoras personales, estaciones de trabajo, máquinas fax, etc. Una tarea principal de *LAN* es soportar comunicaciones de datos entre sus nodos.

El usuario del proceso de simulación debe, como mínimo poder:

- Enumerar los nodos actuales de la red *LAN*.
- Añadir un nuevo nodo a la red *LAN*.
- Quitar un nodo de la red *LAN*.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la *LAN*, como tiempo medio que emplea un paquete.

15.4 Implementar una jerarquía *Empleado* de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para *Empleados* de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a *Empleados*. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo

de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o al Ministerio de Hacienda. Utilice la herencia para distinguir entre al menos cuatro tipos diferentes de acceso a la información de `Empleado`.

- 15.5 Implementar una clase `Automovil` (*Carro*) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *vehículo*, un automóvil es también una *comodidad*, un *símbolo de estado social*, un *modo de transporte*, etc. `Automovil` debe tener al menos tres clases base y al menos tres clases derivadas.
- 15.6 Escribir una clase `FigGeometrica` que represente figuras geométricas como *punto*, *línea*, *rectángulo*, *triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.
- 15.7 Implementar una jerarquía de tipos datos numéricos que extienda los tipos de datos fundamentales como `int`, y `float`, disponibles en C++. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etcétera.
- 15.8 Implementar una jerarquía de herencia de animales tal que contenga al menos seis niveles de derivación y doce clases.
- 15.9 Diseñar la siguiente jerarquía de clases :

| <i>Persona</i>      |                     | <i>Profesor</i>     |                 |
|---------------------|---------------------|---------------------|-----------------|
|                     |                     |                     |                 |
|                     | <i>Nombre</i>       |                     |                 |
|                     | <i>edad</i>         |                     |                 |
|                     | <i>visualizar()</i> |                     |                 |
| <i>Estudiante</i>   |                     | <i>Profesor</i>     |                 |
| <i>nombre</i>       | <i>heredado</i>     | <i>nombre</i>       | <i>heredado</i> |
| <i>edad</i>         | <i>heredado</i>     | <i>edad</i>         | <i>heredado</i> |
| <i>id</i>           | <i>definido</i>     | <i>salario</i>      | <i>definido</i> |
| <i>visualizar()</i> | <i>redefinido</i>   | <i>visualizar()</i> | <i>heredada</i> |

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice

- 15.10 Crear una clase base denominada `Punto` que consta de las coordenadas `x` e `y`. A partir de esta clase, definir una clase denominada `Círculo` que tenga las coordenadas del centro y un atributo denominado `radio`. Entre las funciones miembro de la primera clase, deberá existir una función `distancia()` que devuelva la distancia entre dos puntos, donde
- $$\text{Distancia} = ((x_2 - x_1)^2 + (y_2 - y_1)^2)^{1/2}$$
- 15.11 Utilizando la clase construida en el ejercicio 15.10 obtener una clase derivada `Cilindro` derivada de `Círculo`. La clase `Cilindro` deberá tener una función miembro que calcule la superficie de dicho cilindro. La fórmula que calcula la superficie del cilindro es  $S = 2\pi r (h + r)$ , donde  $r$  es el radio del cilindro y  $h$  es la altura.
- 15.12 Crear una clase base denominada `Rectangulo` que contenga como miembros `datos`, `longitud` y `anchura`. De esta clase, derivar una clase denominada `Caja` que tenga un miembro adicional denominado `profundidad` y otra función miembro que permita calcular su `volumen`.
- 15.13 Dibujar un diagrama de objetos que represente la estructura de un coche (carro). Indicar las posibles relaciones de asociación, generalización y agregación.

PARTE

IV

## Programar en C++





## De C a C++

### Contenido

- 16.1 El primer programa C++
- 16.2 Espacios de nombres
- 16.3 Tipo de datos nativos
- 16.4 Operadores
- 16.5 Conversiones de tipos

### 16.6 Estructuras de control

- 16.7 Funciones
  - › Resumen
  - › Ejercicios
  - › Problemas

## Introducción

Existen muchas formas de analizar el código C++. Una es desde el punto de vista del *programador de aplicaciones* que se centra en los tipos y clases necesarias para crear programas utilizando funciones y estructuras de control. Existe otra visión que es la realizada por el *programador de sistemas* que se centra en obtener los nuevos tipos y clases, que son, a su vez, abstracciones de conceptos.

En la práctica real y profesional, los programadores en el siglo XXI, normalmente adoptan ambos enfoques, debido a que el proceso de abstracción del problema conduce, normalmente, al uso frecuente y repetido de las clases y tipos en los programas, teniendo presente las clases a nivel más alto, las relaciones entre ellas, etcétera.

En este capítulo se describirán las herramientas básicas que utiliza el *programador de aplicaciones* utilizando el lenguaje de programación **C++**. En esencia es un curso básico de programación en C++, preparado como curso de introducción para los lectores no iniciados en el lenguaje o para aquellos que provienen de C y desean aprender rápidamente los conceptos fundamentales para escribir código C++ básico de la manera más eficiente.

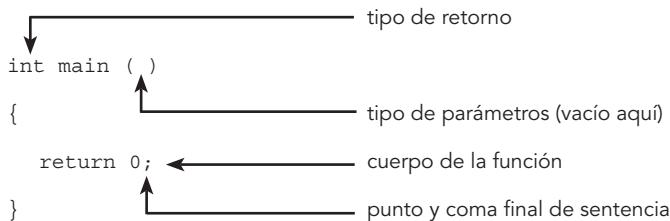
### Conceptos clave

- › cin
- › cout
- › dynamic\_cast
- › main()
- › namespace
- › operador
- › procesador
- › referencia
- › static\_cast
- › std
- › tipos integrales
- › using

### 16.1 El primer programa C++

Cada programa C++ contiene una o más funciones, una de las cuales se denomina `main()`. Una función consta de una o más *sentencias* que ejecutan el trabajo de la fun-

ción. El sistema operativo ejecuta un programa llamando a la función `main()`; esta función ejecuta sus sentencias y devuelve un valor al sistema operativo.



El sistema operativo utiliza el valor devuelto por `main()` para determinar si el programa ha tenido o no éxito. Un valor de retorno 0 indica éxito. La función `main()` debe existir en cada programa C++ y es la única función que llama explícitamente el sistema operativo.

Un programa C++ completo que visualice la frase en español "Hola mundo" es:

```

// programa hola.cpp
// el programa visualiza "Hola mundo"

#include <iostream> // declaraciones
using namespace std;
int main() // función main()
{
 // visualizar "Hola mundo"
 cout<< "Hola mundo!" << endl;
 return 0;
}

```

**Nota:** Entre los compiladores modernos el compilador más idóneo es aquel que utiliza

```
#include <iostream>
```

aunque en compiladores antiguos todavía se utiliza

```
#include <iostream.h>
```

La directiva `using namespace std` define el espacio de nombres `std` y evita tener que utilizar esta directiva.

Si no se utiliza el espacio de nombres `using namespace std` la salida con `cout` requerirá ser precedida por el espacio de nombre `std`.

```

#include <iostream>
int main ()
{
 // visualizar "Hola mundo"
 std::cout<< "Hola mundo" << std::endl;
 return 0;
}

```

El programa `hola.cpp` contiene los siguientes elementos:

- Comentario, indicados por la doble barra `//`
- Directiva del preprocesador `#include`

```
#include <iostream>
```

Se necesita para todas las declaraciones de entrada y salida.

- Cabecera de la función: `int main()`.
- Una directiva `using namespace`.
- Un cuerpo de la función separado por `{` y `}`.
- Sentencias que utilizan `cout` para visualizar un mensaje.
- Una sentencia `return` para terminar la función `main()`.

## Comentarios en C++

Además del estilo de comentarios propios de C++

```
// Esto es un comentario
Es posible escribir en programa comentarios estilo C
/* este es un comentario
en varias líneas de programa */
```

## El preprocesador de C++ y el archivo `iostream`

La primera línea de programa

```
#include <iostream>
```

Se denomina *directiva del preprocesador*, que es una directiva del preprocesador que es una instrucción al compilador. El *preprocesador* es una parte de compilador que procesa un archivo fuente antes de que tenga lugar la compilación. La directiva `#include` es reemplazado por el contenido del archivo indicado. El tipo de archivo incluido normalmente por `#include` se denomina *archivo de cabecera*. Los nuevos archivos de cabecera de ANSI C++ estándar no tienen una extensión `.h` como los antiguos archivos de compilador C++.

```
#include <iostream>
```

## Entrada y salida

C++ no define directamente ninguna sentencia para realizar las operaciones de entrada y salida (E/S). La E/S la proporciona la *biblioteca estándar*. La biblioteca de E/S suministra un conjunto amplio de facilidades. La directiva

```
#include <iostream>
```

añade el contenido del archivo de cabecera `<iostream>` para E/S. `iostream` manipula entrada y salida con formatos. La biblioteca `iostream` tiene dos tipos: `istream` y `ostream` que representan los flujos de entrada y salida. Un flujo (`stream`) es una secuencia de caracteres empleada para leer de un dispositivo o escribir en un dispositivo de E/S (el término flujo se refiere al hecho de que los caracteres se generan o conducen secuencialmente uno a uno).

Las características de E/S se consiguen insertando la sentencia `#include <iostream>`. Con la inclusión de la declaración anterior se definen los siguientes símbolos.

Tabla 16.1

| Símbolo                | Significado                         |
|------------------------|-------------------------------------|
| <code>std::cin</code>  | canal de entrada estándar (teclado) |
| <code>std::cout</code> | canal de salida estándar (pantalla) |
| <code>std::cerr</code> | canal de salida de error (pantalla) |
| <code>std::endl</code> | símbolo para avance de línea        |

La entrada se realiza a través de un canal estándar, que se asigna generalmente al teclado. La salida se hace a través de otro canal estándar que se asigna por lo general a la pantalla. Ambas pueden ser redirigidas por el sistema operativo o utilizando funciones del sistema, de modo que, por ejemplo, la salida estándar se escriba en un archivo.

Los datos se envían al canal de salida utilizando el *operador* `<<` y se pueden encadenar múltiples salidas:

```
#include <iostream>
std:: cout<< "poner un número" << 45 << std:: endl;
```

endl es una instrucción especial de C++ que significa “comienzo de nueva línea”. La inserción de endl en el flujo de salida hace que el cursor de la pantalla se mueva al principio de la siguiente línea. Por ejemplo, cout << "Hola!!" << endl; otra forma de indicar nueva línea es con el carácter '\n', por ejemplo, cout << "\n" equivale a cout << endl.

```
#include <iostream>
int main ()
{
 std::cout << "Introducir dos números:" << std::endl;
 int n1, n2;
 std::cin >> n1 >> n2;
 std::cout << "La suma de " << n1 << "+" << n2
 << "es" << n1 + n2 << std::endl;
 return 0;
}
```

### Ejemplo 16.1

#### Regla

cout << "Introducir dos números"; *visualiza texto*  
 cin >> n1 >> n2; *entrada de dos datos*

## Archivos de cabecera

La directiva #include indica que se desea utilizar la biblioteca `iostream` y que se debe añadir el archivo fuente `iostream` al archivo fuente del programa antes de comenzar la compilación.

Los archivos como `iostream` se denominan también *archivos de inclusión (include)* ya que se adjuntan en otros archivos o bien *archivos de cabecera (header)*, ya que ellos se incluyen al principio de un archivo. Los compiladores de C++ vienen con muchos archivos de cabecera que soportan un amplio conjunto de propiedades y características.

#### Regla

La directiva `#include` se debe escribir en una sola línea; el nombre del archivo de cabecera e `include` deben aparecer en la misma línea. En general, las directivas `#include` aparecerán fuera de cualquier función; normalmente, todas las directivas `#include` de un programa aparecen al principio del archivo.

## 16.2 Espacios de nombres

C++ estándar tiene la característica de *espacios de nombre*, que permite facilidades para nombrar a variables, funciones, estructuras, clases, enumeraciones, etc. y que no se produzcan conflictos en dichos nombres.

Los espacios de nombres proporcionan un mecanismo para controlar la colisión de nombres ya que cada nombre definido en un ámbito dado debe ser único dentro de ese ámbito. Un espacio de nombres comienza con la palabra reservada `namespace` seguida del nombre del espacio.

En C++ existe un espacio de nombres estándar, `std` en la biblioteca `iostream` de entrada y salida. De este modo se utiliza `std::cout` y `std::endl` en lugar de `cout` y `endl`. El prefijo `std::` indica que los nombres `cout` y `endl` están definidos en el espacio de nombres `std`.

El símbolo `::` se denomina operador de *resolución de ámbito o (alcance)*, y se emplea, entre otros casos, para salida (`cout`), entrada (`cin`), y para fin de línea.

std::cout utiliza el nombre cout definido en el espacio de nombre std.

std::cin >> v1 lee la entrada de v1 utilizando cin definido en std.

### Regla 1

Si se usa el archivo <iostream> se necesita utilizar el espacio de nombre std para E/S

```
#include <iostream>
int main ()
{
 std::cout<< "¡Hola mundo!" << std::endl;
 return 0;
}
```

Sin embargo es posible otro estilo de escritura que *requiera* el uso de std:: cuando se utiliza el archivo iostream. Este estilo de escritura requiere el uso de la directiva *using* que permite disponer las definiciones de iostream en sus programas.

Si desea utilizar este método similar al estilo iostream.h debe incluir en sus programas

```
#include <iostream>
using namespace std;
```



### Ejemplo 16.2

```
include <iostream>
using namespace std;
int main ()
{
 cout<< "¡Hola Mundo!" << endl;
 return 0;
}
```

## 16.3 Tipos de datos nativos

C++ proporciona un conjunto de tipos de datos predefinidos y operadores que manipulan los tipos de datos. C++ puede definir un número de tipos de datos, pero no especificar su tamaño. El tamaño dependerá del sistema operativo y de la plataforma sobre la que se ejecute su programa. Los siguientes tipos de datos están predefinidos:

- Tipos de entero.
- Tipos de coma flotante.
- Tipos lógicos.
- Tipos carácter.
- Tipos referencia.
- Tipos enumeraciones.
- Tipos arreglo (*array*).
- Nombres de tipos definidos por el usuario.
- Otros tipos compuestos.

### Tipos de datos básicos/primitivos

C++ define un conjunto de tipos aritméticos: enteros, coma flotante, carácter individual y valores lógicos (booleanos). Además existe un tipo denominado void. El tipo void no tiene valores asociados y solo se puede utilizar en un conjunto limitado de circunstancias; usualmente se utiliza como el tipo de retorno de una función que no devuelve ningún valor.

**Tabla 16.2** / Tipos aritméticos en C++.

| Tipo        | Significado                    | Tamaño mínimo             |
|-------------|--------------------------------|---------------------------|
| bool        | <i>boolean</i> (lógico)        | 1 bit                     |
| char        | carácter                       | 8 bits                    |
| wchar_t     | carácter ancho                 | 16 bits                   |
| short       | entero-corto                   | 16 bits                   |
| int         | entero                         | 16 bits                   |
| long        | largo-entero                   | 32 bits                   |
| float       | real simple precisión          | 6 dígitos significativos  |
| double      | real doble precisión           | 10 dígitos significativos |
| long double | real doble precisión extendido | 10 dígitos significativos |

El tamaño de los tipos aritméticos varía de unas máquinas a otras. Tamaño es el número de bits utilizado para representar el tipo y depende de los compiladores y máquinas sobre las que corren los programas. La tabla 16.2 enumera los tipos incorporados o predefinidos y los tamaños mínimos asociados. Las tablas 16.3 y 16.4 enumeran los tipos de datos enteros y enteros con signo.

El tipo `bool` representan los valores `true` y `false`. Se puede asignar cualquier valor aritmético a `bool`. Un tipo aritmético de valor 0 produce un tipo de `bool` que almacena falso (`false`) y cualquier otro valor, distinto de cero se considera verdadero (`true`).

**Tabla 16.3** / Tipos integrales.

| Tipo  | Significado                                      |
|-------|--------------------------------------------------|
| int   | entero (tamaño de palabra típica de la máquina)  |
| short | valor más pequeño que el normal                  |
| long  | valor mayor que el normal                        |
| bool  | número lógico o <i>boolean</i> (verdadero/falso) |

**Tabla 16.4** / Tipos con signo.

|                          |                       |
|--------------------------|-----------------------|
| int, short, long         | con signo por defecto |
| unsigned int, o unsigned | entero sin signo      |
| unsigned long            | long con signo        |
| char                     | con signo por defecto |
| signed char              | con signo             |
| unsigned char            | char sin signo        |

## Tipos de coma flotantes (reales)

Los tipos `float`, `double` y `long double` representan valores de coma flotante de simple y de doble precisión. Normalmente, `float` representa una palabra (32 bits), `double` representa dos palabras (64 bits) y `long double`, representa tres o cuatro palabras (96 o 128 bits).

El tamaño del tipo determina el número de dígitos significativos que un valor de coma flotante puede contener (normalmente `float`, 6 dígitos significativos y `double`, 10 dígitos significativos).

## Constantes literales

Cada constante literal tiene un tipo de dato básico asociado y no es direccionable.

### 1. Constantes literales enteras (decimal, octal, hexadecimal)

```
26 // decimal
024 // octal comienza con un 0
0x1A // hexadecimal comienza con un 0x o bien 0X
```

### 2. Con signo/sin signo y largo

```
128u /* unsigned int */
1053L /* long*/
5LU /* unsigned long */
```

### 3. Literales de coma flotante (científico, decimal) y de simple, doble precisión

```
1.23e-3F // notación científica, e o bien E, simple precisión
3.141592F // simple precisión, f o F
003f // simple precisión
1E-3F // simple precisión
1.2345E1L // precisión ampliada, l o L
```

### 4. Constante literal carácter

```
'a', 'd', '5', ...
```

### 5. Literales lógicos (true o false)

```
bool prueba = false;
```

### 6. Secuencias de escape para caracteres no imprimibles (un carácter no imprimible, **no visualizable**, es un carácter cuya imagen no es visible, como un retroceso de espacio o un carácter de control);

|                        |    |                       |     |
|------------------------|----|-----------------------|-----|
| nueva línea            | \n | tabulación horizontal | \t  |
| tabulación vertical    | \v | retroceso de espacio  | \b  |
| retorno de carro       | \r | avance fuente         | \f  |
| alerta (timbre)        | \a | barra inclinada       | \\\ |
| signo de interrogación | \? | comilla simple        | \'  |
| doble comillas         | \" |                       |     |

### 7. Literales de cadenas de caracteres

```
"Hola mundo" // literal de cadena
"" // literal de cadena vacía
'A' // carácter
"A" // literal de cadena de caracteres
```

### 8. Literales de cadenas concatenadas

Dos literales cadena que adyacentes una al lado de la otra y separadas solo por espacios, tabulaciones o nuevas líneas se concatenan en un solo literal de cadena.

```
std:: cout << "multilínea"
"cadena literal"
"cero de concatenación"
<< std:: endl;
```

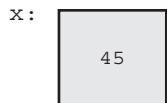
## Tipos apuntadores

Los apuntadores (punteros) referencian a datos. Una variable apuntador contiene una dirección como su valor en memoria y “apunta” a otra variable, objeto o incluso función.

Los operadores `&` y `*` se utilizan en el diseño de apuntadores.

- El operador unitario `&` produce la dirección de un valor o un apuntador a valor.
- El operador unitario `*` **desreferencia** un apuntador, que significa lo que se devuelve y sirve para declarar el apuntador.

1. `int x = 45;` // entero  
`int *xp;` // apuntador a entero



x tiene un valor definido apuntador xp tiene un valor indefinido

2. La sentencia `xp = &x;` // xp apunta a x  
La dirección de x se asigna a xp



3. `*xp = 50`



4. El valor xp se puede obtener mediante

`cout << *xp << endl;`  
o bien con el espacio de nombres std  
`std::cout << *xp << std::endl;`

### Ejemplo 16.3



## Declaración de apuntadores

Los apuntadores (punteros) se declaran situando un asterisco delante del nombre

`int *ptr;`

Muchos programadores C++ adoptan este otro estilo de declaración:

`int* ptr;`

Para significar que `int*` es un tipo, apuntador a `int`

`int* p1, p2;`  
`double *ptr_tasa;`  
`char* cad;`

### Ejemplo 16.4



En C++, todas las manipulaciones de cadenas de caracteres se hacen utilizando apuntadores a caracteres (`char*`). Cada constante cadena de caracteres es de tipo `char*`

`char *cadptr = "Esto es una prueba";`

## Tipos constantes

El calificador `const` aplicado a una variable transforma esta variable en una constante simbólica:

`const tipo variable = valor;`  
`const int MES=12; //MES, constante simbólica de valor 12`

## Referencias

Una *referencia* es un nombre alternativo de un objeto o de una variable. En los programas, las referencias se utilizan principalmente como parámetros formales a funciones. Una referencia es un **tipo compuesto** que se define precediendo a un nombre de variable el símbolo **& (operador de dirección)**. Un tipo compuesto es un tipo que se define en términos de otro tipo.

La variable o referencia debe ser inicializada utilizando una variable del mismo tipo que la referencia.

```
int valR = 1024;
int &valRef = valR; // valRef se refiere a valR
int &valRef2; // error, no está inicializado valRef2
int &valRef3 = 50 // error, el inicializador debe ser un objeto
```



### Ejemplo 16.5

```
double y = 11.12345;
double &refy = y;
refy += 4.54321; // y = 15.66666
```

## Nombres de tipos definidos: `typedef`

Un `typedef` permite definir un sinónimo de un tipo. La definición comienza con la palabra reservada `typedef` seguido por el tipo de dato y el identificador (nombre del `typedef`).

```
typedef double nomina; // nomina, sinónimo de double
typedef int calificación; // calificación, sinónimo de int
```

Un nombre de tipo puede ser utilizado como un especificador de tipos:

```
typedef struct { ... } Profesor;
Profesor consultor;
calificación alumno, doctorando;
```

Los tipos definidos (`typedef`) se utilizan para hacer los programas más legibles y encapsular aspectos del programa dependientes de la máquina.

## Tipos enumeración

Una enumeración se declara con la palabra reservada `enum` y una lista de enumeradores separados por comas y encerrados entre llaves (`{}`).

```
enum modos_archivo {entrada, salida, agregar};
enum colores {azul, verde, blanco, amarillo};
```

Por defecto, al primer enumerador se le asigna el valor cero y a los siguientes 1, 2, 3, etc. Se puede asignar un valor inicial a uno o más enumeradores.

```
enum Estadistica {si, no = 50, todo_lo_contrario}
enum dias_semana {lun, mar, mier, jue, vier, sab, dom = 7};
enum interruptor {abierto, cerrado}
enum horario {am, pm};
```

## Arrays (arreglos)

Un **array (arreglo)** es una colección de diferentes elementos del mismo tipo, dispuestos en secuencia. El número de elementos debe indicarse cuando se crea el array. El acceso a los elementos se realiza mediante el operador `[ ]`:

```
int valores[10]; // array de enteros
valores[0] = 75; // inicializa el primer elemento
valores[9] = valores[0];
```

El índice del array comienza en 0 y llega a longitud - 1.

### Sintaxis

```
nombre_Tipo nombre_Array [longitudArray];
intmes[12]; // crea un array de 12 enteros
```

```
int mes [12];
for (int i=0; i<12; ++i) { // inicializa todos los elementos
 mes [i] = 5000;
}
```

### Ejemplo 16.6

### A recordar

Una definición de un *array* (arreglo) consta de un especificador de tipo, un identificador y una dimensión.

```
float arreglo_float [100];
```

El valor de la dimensión (longitud) del *array* se calcula en tiempo de compilación y debe ser una expresión constante. Los elementos del array se numeran comenzando con 0.

C++ soporta arrays multidimensionales y se declaran especificando dimensiones adicionales.

```
float tablas [5] [10]; // declara un array de dos
 // dimensiones, 5 x 10
double ventas [4] [15]; // array de 4 x 15 elementos
```

Los arreglos multidimensionales se pueden definir especificando los valores de cada fila del arreglo:

```
int amul [3] [4] = {
 {0, 1, 2, 3}, // array de tres elementos
 {4, 5, 6, 7}, // de tamaño cada uno de 4 elementos
 {8, 9, 10, 11} // 3 x 4
};
```

### Tipos carácter

Como ya se ha comentado en C++, el tipo *char* existe como un medio para almacenar un único carácter. Se puede crear un tipo *char* de modo similar a otra variable.

```
char letraInicial;
letraInicial = 'L'; // un solo carácter
```

### Cadenas

Una de las grandes ventajas de C++ sobre C es el tratamiento del tipo de dato cadena (*string*) de caracteres. Una cadena es un valor entre dobles comillas que contiene cualquier combinación de caracteres, números, espacios y símbolos.

Para utilizar cadenas en C++ se debe incluir en primer lugar el archivo de cabecera `string` que define todas las funciones necesarias para su manipulación.

```
#include <string>
```

Al igual que los tipos de E/S, `string` está definido en el espacio de nombres `std`. Para declarar una variable de tipo cadena requiere un sistema similar a otras variables pero con la declaración previa de la biblioteca `string`.

```
#include <string>
using namespace std;

string nombreEmpresa;
nombreEmpresa = "Consultoría Carchelejo y Asociados"
```

### Sintaxis

```
string nombreVariable;
nombreVariable = "cadena inicial";
```



### Ejemplo 16.7

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
 string PrimerNombre, Apellido, NombreCompleto;
 Primer nombre = "Luis";
 Apellido = "Mackoy";
 NombreCompleto = PrimerNombre + " " + Apellido;
 cout << "Hola," << NombreCompleto << ".\n";
 return 0;
}
```

## 16.4 Operadores

C++ proporciona un conjunto muy rico de operadores y define cuáles de estos operadores se aplican a tipos fundamentales. Una **expresión** se compone de uno o más **operandos** que se combinan mediante operadores. La forma más simple de una expresión consta de una sola constante literal o variable. Las expresiones más complejas se forman con un operador y uno o más operandos. Cada expresión produce un resultado.

Existen operadores **unitarios** que actúan sobre un solo operando (operador dirección `&`; operador de referencia `*`) y operadores **binarios** y **terciarios**, que actúan sobre dos o tres operandos (suma `+`, resta `-`). Los operadores se clasifican en función de las tareas que ejecutan: aritméticos, relacionales, lógicos, etc.

### Operadores aritméticos

Los **operadores aritméticos** se aplican a cualquiera de los tipos de datos aritméticos o a cualquier tipo que se puede convertir a un tipo aritmético. La tabla 16.5 recoge los operadores aritméticos fundamentales.

En C++ la tabla 16.5 agrupa a los operadores por su precedencia. Los operadores unitarios tienen la prioridad más alta, a continuación los operadores de multiplicación y división y por último los operadores de suma y resta. Desde el punto de vista de la asociatividad, los operadores aritméticos tienen asociati-

**Tabla 16.5** / Operadores aritméticos.

| Operador | Función        | Ejemplo |
|----------|----------------|---------|
| +        | más unitario   | + expr  |
| -        | menos unitario | - expr  |
| *        | multiplicación | a * b   |
| /        | división       | a / b   |
| %        | resto (módulo) | a % b   |
| +        | suma           | a + b   |
| -        | resta          | a - b   |

vidad a izquierda lo que implica que cuando los niveles de precedencia son iguales se agrupan de izquierda a derecha.

## Operadores relacionales y lógicos

Los operadores relacionales y lógicos tienen operandos de tipos numéricos, `char` y lógicos, y devuelven valores lógicos de tipo `bool`. La tabla 16.6 muestra los operadores relacionales y operadores lógicos en orden de precedencia. Las tablas 16.7 y 16.8 muestran los posibles resultados de aplicar el operador OR (`||`) y AND (`&&`), respectivamente.

**Tabla 16.6** / Operadores relacionales y lógicos.

| Operador | Función           | Ejemplo        |
|----------|-------------------|----------------|
| !        | NOT lógico        | !expr          |
| <        | menor que         | expr1 < expr2  |
| <=       | menor o igual que | expr1 <= expr2 |
| >        | mayor que         | expr1 > expr2  |
| >=       | mayor o igual que | expr1 >= expr2 |
| ==       | igualdad          | expr1 == expr2 |
| !=       | desigualdad       | expr1 != expr2 |
| &&       | AND lógico        | expr1 && expr2 |
|          | OR lógico         | expr1    expr2 |

1. 

```
if (edad >= 18)
 std::cout << "puede votar en España";
```
2. 

```
if ((edad > 18) && (edad < 25))
 std::cout << "eres un trabajador";
```

### Ejemplo 16.8

**Tabla 16.7** / Operador lógico OR (`||`).

|                | expr1 == true | expr1 == false |
|----------------|---------------|----------------|
| expr2 == true  | true          | true           |
| expr2 == false | true          | false          |

Tabla 16.8 / Operador lógico AND (&amp;&amp;).

|                | expr1 == true | expr1 == false |
|----------------|---------------|----------------|
| expr2 == true  | true          | false          |
| expr2 == false | false         | false          |

El operador lógico NOT (!) niega o invierte el valor verdadero de la expresión que sigue. Es decir si expresión es verdadero entonces !expresión es falso.

if (! (x > 8)) equivale a if (x <= 8)

## Operadores de asignación

Un operador de asignación es un operador que se formula por un signo (=).

|              |            |                      |
|--------------|------------|----------------------|
| x = x + 5;   | equivale a | x += 5;              |
| suma += val; | equivale a | suma = suma + valor; |

Tabla 16.9 / Operadores de asignación.

| Operador | Significado                 |
|----------|-----------------------------|
| =        | Asignación simple           |
| *=       |                             |
| /=       |                             |
| %=       |                             |
| +=       | m operador = n ; equivale a |
| -=       | m = m operador n            |
| <<=      |                             |
| >>=      |                             |
| &=       |                             |
| ^=       |                             |

## Operadores incremento y decremento

Los operadores **incremento** (++) y **decremento** (--) proporcionan una notación adecuada para sumar o restar 1 de un objeto. Existen dos funciones de estos operadores: **prefijo** y **posfijo**.

Tabla 16.10 / Operadores de incremento y decremento.

| Operador | Significado              |
|----------|--------------------------|
| ++       | Incremento posfijo (a++) |
| --       | Decremento posfijo (a--) |
| ++       | Incremento prefijo (++a) |
| --       | Decremento prefijo (--a) |



### Ejemplo 16.9

```

1. int i = 0, j;
 j = ++i; // j = 1, i = 1
 j = i++; // j = 1, i = 2
2. x = 45
 cout << ++x; // se visualiza 46
 cout << x // se visualiza 46

```

```
3. x = 45
cout << x++; // se visualiza 45 (primero produce el valor 45, a
 // continuación se incrementa)
cout << x; // se visualiza 46
```

## Operador condicional

El operador condicional es el único operador ternario C++. Se escribe `? :` y se utiliza, con frecuencia, en lugar de la sentencia `if-else`. Su sintaxis es la siguiente:

```
condición ? expresión1 : expresión2
```

donde `condición` es una expresión que se utiliza como condición.

El operador se ejecuta evaluando `condición`. Si `condición` se evalúa a 0 la condición es falsa y cualquier otro valor es verdadero. Si la condición es verdadera entonces se evalúa la `expresión1`, en caso contrario se evalúa `expresión2`.

### Ejemplo 16.10

```
1.
int i = 5, j = 25, k = 40;
// si i > j entonces valorMax = i sino valorMax = j
int valorMax = i > j ? i : j;

2.
if (x < y) {
 z = x;
}
else {
 z = y;
}
```

Es equivalente a:

```
z = x < y ? x : y;
```

(si `x` es menor que `y` se asigna `x`, en caso contrario se asigna `y`)

```
3.
cout << (x == 0 ? "falso" : "verdad") << endl;
si x toma un valor de 0, se escribe falso si no se escribe verdad

4.
int c = a > b ? a : b;
produce el mismo resultado que
```

```
int c;
if (a > b)
 c = a;
else
 c = b;
```

## Operador `sizeof`

El operador `sizeof` devuelve un valor de tipo `size_t`, que es el tamaño en bytes de un objeto o un tipo. El resultado de la expresión `sizeof` es una constante en tiempo de compilación. La sintaxis del operador es:

```
sizeof (nombre_tipo)
sizeof (expr);
sizeof expr;
```

Al aplicar `sizeof` a una expresión `expr` devuelve el tamaño del tipo de resultado de esa expresión.

```
articulo_ventas articulo,*p;
sizeof (articulo_ventas); // tamaño del tipo articulo_ventas
sizeof articulo; // tamaño del tipo de la variable articulo
sizeof *p; // tamaño del tipo al que apunta p
```

## Operador coma

El **operador coma** permite que dos sentencias se combinen en una sola. Una expresión coma es una serie de expresiones separadas por comas. Las expresiones se evalúan de izquierda a derecha. El resultado de una expresión coma es el valor de la expresión más a la derecha. Un uso común del operador coma es en un bucle `for`.



### Ejemplo 16.11

```
1. ++j, --k // dos expresiones unidas en una sola
2. int i, j; // aquí coma es un separador, no operador
3. for (i = 0, j = 5; i < j; --j, ++i)
{
 //sentencias del bucle
}
```

## 16.5 Conversiones de tipos

El tipo del operando u operandos determina si una expresión es legal y, si es legal, determina el significado de la expresión. En C++ existe una gran profusión de tipos y muchos de ellos están relacionados entre sí, de modo que se puede utilizar un objeto o un valor de un tipo en el lugar donde se espera un tipo de operando relacionado.

La relación entre tipos se manifiesta mediante la conversión entre ellos. C++ realiza conversiones automáticas de tipos con el objetivo fundamental de evitar errores en compilación cuando se utilizan tipos de datos diferentes en las mismas expresiones y sentencias.

Algunas de las conversiones automáticas son:

- C++ convierte valores cuando se asigna un valor de un tipo aritmético a una variable de otro tipo aritmético.
- C++ convierte valores cuando se combinan tipos mixtos de expresiones.
- C++ convierte valores cuando se pasan argumentos a funciones.

La tabla 16.11 recoge las reglas de conversión y algunos problemas que pueden surgir.

**Tabla 16.11** Reglas de conversión y algunos problemas potenciales.

| Tipos de conversión                                                                                                              | Problemas potenciales                                                                         |
|----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Tipos mayores de coma flotante se convierten a tipos más pequeños de coma flotante ( <code>double</code> a <code>float</code> ). | Pérdida de precisión; el valor puede quedar fuera del rango del tipo destino.                 |
| Tipos de coma flotante a tipo entero (el tipo <code>double</code> se convierte a <code>int</code> )                              | Pérdida de la parte decimal, el valor original puede quedar fuera del rango del tipo destino. |
| Tipo entero más grande a tipo entero más pequeño como <code>long</code> a <code>short</code> .                                   | El valor original puede quedar fuera del rango del tipo destino.                              |

Cuando los valores `bool` se promueven a `int`, un valor falso se promueve a cero y un valor verdadero no cero (`true`) a uno.

## Conversión en expresiones

Cuando se combinan dos tipos aritméticos diferentes en una expresión, C++ realiza dos clases de conversiones aritméticas. Primero, algunos tipos se convierten automáticamente, por ejemplo, `short` a `int`; segundo, algunos tipos se convierten cuando se combinan con otros tipos en las expresiones.

Los tipos más simples de conversiones se denominan **promociones integrales**. C++ convierte valores `bool`, `char`, `unsigned char`, `signed char` y `short` a `int`. En particular, `true` se promueve a 1 y `false` a 0. De modo similar `wchar_t` se promueve al primero de los siguientes tipos que sea lo suficiente ancho como para aceptar su rango: `int`, `unsigned int`, `long` o `unsigned long`.

### A recordar

Cuando se combine tipos diferentes en una expresión aritmética como un `int` a un `float`, el más pequeño (`int`) se convierte al mayor (`float`).

```
short svalor;
int ivalor;
float fvalor;
long lvalor;
double dvalor;
dvalor + ivalor; // ivalor se convierte a double
dvalor + fvalor; // fvalor se convierte a double
ivalor = dvalor; // dvalor se trunca a int
```

### Ejemplo 16.12

## Conversiones en paso de argumentos

C++ aplica las promociones integrales y realiza las conversiones de tipo en el paso de argumento a las funciones.

## Conversiones explícitas

C++ permite forzar conversiones explícitas de tipos mediante el mecanismo denominado *moldeo* (*cast*) de tipos. C++ soporta la notación típica de *moldeo* de tipos mediante los siguientes dos formatos:

```
(nombretipo) valor; // notación estilo C
nombretipo (valor); // notación estilo función
```

Convertir un valor `int` almacenado en una variable llamada `mesa` a tipo `long`. Se puede utilizar cualquiera de las siguientes expresiones:

```
(long) mesa
long (mesa)
```

### Ejemplo 16.13

## Operador `new`

C++ proporciona el operador `new` para obtener bloques de memoria. El operador `new` asigna un bloque de memoria cuyo tamaño es del tipo de dato al que se aplica. El dato u objeto dato puede ser un `int`, un `float`, una estructura, un array o cualquier otro tipo de dato. El operador `new` devuelve un apuntador, que es la dirección del bloque asignado de memoria. El apuntador se utiliza para referenciar el bloque de memoria.

### Sintaxis completa del operador new

```
1. apuntador = new nombreTipo // no arreglos (arrays)
 o bien:
 apuntador = new nombreTipo (inicializador)
2. apuntador = new nombreTipo [dimensión] // arreglos (arrays)
```

Cada vez que se ejecuta una expresión que invoca el operador new, el compilador realiza una verificación de tipo para asegurar que el tipo del apuntador especificado en el lado izquierdo del operador es el tipo correcto de la memoria que se asigna a la derecha. Si los tipos no coinciden, el compilador produce un mensaje de error.

El campo *apuntador* (*puntero*) es el nombre del apuntador al que se asigna la dirección del objeto dato, o NULL, si falla la operación de asignación de memoria. En el segundo formato, al *apuntador* se le asigna la dirección de memoria de un bloque lo suficientemente grande para contener un array con *dimensiones* elementos.



#### Ejemplo 16.14

El efecto de la siguiente sentencia es crear una variable entera sin nombre e inicializada a 8, accesible solo a través de un apuntador *pEnt*.

```
int *pEnt = new int(8);
```

A continuación se crea un arreglo de 100 elementos de tipo char. La dirección de inicio del arreglo la tiene *p*.

```
char * p = new char [100];
```

Ahora se crea un arreglo de 100 enteros:

```
int *BloqueMem;
BloqueMem = new int[100];
```

o bien, con una sentencia

```
int *BloqueMem = new int[100];
```

También se pueden crear matrices:

```
char**p;
p = new (char*)[12];
for (int i = 0; i < 12; i++)
 p[i] = new char[10];
```

O bien, de forma directa con la sentencia new char[12][10]

### Precaución

El almacenamiento libre no es una fuente inagotable de memoria. Si el operador new se ejecuta con falta de memoria, devuelve un apuntador NULL. Es responsabilidad del programador comprobar siempre el apuntador para asegurar que es válido antes de que se asigne un valor al apuntador. Supongamos, por ejemplo, que se desea asignar un arreglo de 8 000 enteros:

```
int *ptr_lista = new int[8000];
if (ptr_lista == NULL)
{
 cout << "Falta memoria" << endl;
 return -1; // Hacer alguna acción de recuperación
}
```

### Operador delete

El operador `delete` garantiza un uso seguro y eficiente de la memoria; libera la memoria reservada con `new`.

```
int * ps = new int; // asigna memoria con new
delete ps; // libera la memoria
```

### Ejemplo 16.15

La acción de `delete` elimina la memoria a la cual apunta `ps` pero no borra el apuntador `ps` que se puede seguir utilizando; es decir, se puede utilizar `ps`, por ejemplo, para apuntar a otra asignación de `new`. Es importante que exista siempre una correspondencia `new-delete`. No se puede utilizar `delete` para liberar memoria creada por declaración de variables ordinarias.

```
int * p = new int; // correcto
delete p; // correcto
int num = 10; // correcto
int * pn = & num; // correcto
delete pn; // incorrecto, memoria no asignado por new
```

### Precaución

Cuando se termina de utilizar un bloque de memoria previamente asignado por `new`, se puede liberar el espacio de memoria y dejarlo disponible para otros usos, mediante el operador `delete`. 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 es `delete apuntador`.

En las declaraciones

### Ejemplo 16.16

1. `int *ad;`  
`ad = new int`  
 o bien,  
`int *ad = new int`
2. `char *adc = new char[100];`  
 el espacio asignado se puede liberar con las sentencias:  
`delete ad;`  
`delete [ ] adc;`

Obsérvese que si se utiliza `new` sin corchetes se debe utilizar `delete` sin corchetes. Si se utiliza `new` con corchetes se debe utilizar `delete` con corchetes.

## 16.6 Estructuras de control

Las sentencias de control de un lenguaje de programación determinan el flujo de ejecución a través de un programa. Existen tres categorías: **Secuencia**, **Selección** y **Repetición**.

### Secuencia y sentencias compuestas

Una **sentencia compuesta** es un grupo de sentencias que se ejecutan en secuencia y se escriben encerradas entre símbolos tipo llave. Las sentencias se ejecutan en el orden en que aparecen.

```

{
 double m = 4.96;
 double n = 3.141516;
 double z = m * n;
 int i = int(z);
 i++;
}

```

## Selección y repetición

Las sentencias o estructuras de control no secuenciales se clasifican en **sentencias de selección** (if-else y switch) y **sentencias de repetición** (iterativas o bucles, for, while y do\_while).

Las sentencias for, while y do\_while implementan la realización de bucles (lazos, loop) que contienen sentencias que se ejecutan repetidamente.

### Sentencia if

Una sentencia if se utiliza para comprobar una condición determinada. La sentencia if tiene dos formatos: if e if-else

| Sintaxis de if                          | Sintaxis de if-else                                          |
|-----------------------------------------|--------------------------------------------------------------|
| <pre>if (condición)     sentencia</pre> | <pre>if (condición)     sentencia1 else     sentencia2</pre> |

Una condición **verdadera** hace que el programa ejecute la sentencia y una condición **falsa** hace que el programa salte dicha sentencia.

Si la condición verdadera (true, o distinto de cero), el programa ejecuta sentencia1 y una condición falsa (cero) el programa ejecuta la sentencia2.

**Nota.** En ambos casos la sentencia puede ser una sentencia compuesta.

La **condición** es una **expresión** de tipo lógico (bool) entre paréntesis. La expresión de tipo bool, normalmente puede implicar, comparaciones escritas que utilizan operadores de igualdad (==, !=) y operadores relacionales (<, <=, >, >=).

Las condiciones también pueden contener operadores lógicos ! (not, complemento), && (and) y || (or) que se combinan en expresiones lógicas.



### Ejemplo 16.17

1. if (n < 25)
 cout << "es menor que 25" << endl;
else
 cout << "es mayor que 25" << endl;
2. Visualizar el menor de dos números
 if (x < y)
 min = x;
 else
 min = y;
 cout << "el valor número es" << min;

## Sentencia switch

La sentencia `switch` se utiliza para seleccionar entre múltiples valores constantes; por ejemplo, crear un menú en la pantalla que solicite al usuario seleccionar una de cinco elecciones posibles ('Almorzar', 'Desayunar', 'Cenar', 'Aperitivo' y 'Merienda'). La sintaxis de `switch` es:

```
switch (expresión entera o selector)
{
 case etiqueta1: sentencia(s);
 break;
 case etiqueta2: sentencia(s);
 break;
 ...
 default: sentencias;
}
```

La sentencia `switch` indica a la computadora la línea de código a ejecutar. El programa ejecuta la sentencia `switch` y salta a la línea etiquetada en el valor correspondiente al valor de `expresión entera` (selector). Por ejemplo, si `expresión_entera` toma el valor 5, es decir `case 5`. Las etiquetas deben ser expresiones de constantes enteras. La mayoría de las etiquetas suelen ser simples valores `int` o `char`, como '3', '4', o '5'. Si la `expresión_entera` toma un valor que no coincide con ninguna de las etiquetas, el programa salta a línea etiquetada con `default`. La etiqueta `default` es opcional, si se omite y no hay coincidencia de etiquetas, el programa salta a la sentencia siguiente a `switch`. La sentencia `break` produce una salida de la sentencia `switch`; sin ella, la ejecución del programa continuaría en la siguiente sentencia `case`. Una de las mejores aplicaciones de `switch` es sustituir a `if` anidadas y en la realización de menús.

```
switch (operador)
{
 case '+': resultado = m + n;
 suma++;
 break;
 case '-': resultado = m - n;
 diferencia--;
 break;
 case '*': resultado = m * n;
 producto--;
 break;
 case '/': resultado = m / n;
 cociente--;
 break;
 default: cout << "respuesta no válida";
}
```

### Regla

`switch` e `if-else`. Si se puede utilizar una sentencia u otra, la regla usual es emplear `switch` si existen tres o más alternativas.

## Sentencia while

Uno de los conceptos más fundamentales en cualquier lenguaje de programación es el de bucle (`lazo`, `loop`). Un bucle es un conjunto de sentencias (puede no tener sentencias, bucle vacío) que se ejecutan repetidamente hasta que se cumple una determinada condición o bien se alcanza un número fijo de iteraciones o repeticiones de todas las sentencias. `while`, `do-while` y `for` son las sentencias típicas de realización de bucles.

El bucle `while` es un bucle `pre-test` (`pre-condición`) donde la condición que debe cumplirse en el bucle para su continuación se comprueba antes de cada parada (`iteración`) del bucle.

### Sintaxis

```
1. while (condición)
 sentencia
2. while (condición) {
 sentencias
}
```

while evalúa condición y si es verdadera se ejecuta la/s sentencia/s. La diferencia con if es que while repite la operación una y otra vez hasta que condición es falsa.

```
cin >> n; // leer un valor n del teclado
i = 1;
while (i <= n){ // mientras i sea menor o igual que n
 cout<< i << " ";
 i = i + 1; // incrementar i en 1
}
```

### Sentencia do-while

El bucle do-while es un bucle post-test (post-condición). La condición del bucle se comprueba después de cada ejecución. Como resultado el bucle do-while se ejecuta al menos una vez.

### Sintaxis

```
do
 sentencia
while (condición)
```

### Ejemplo

```
do {
 sentencias
} while (condición)
```



#### Ejemplo 16.18

x se incrementa al menos una vez y luego se sale del bucle

```
do {
 ++x;
 cout<< "x: " << x << endl;
} while (x < 10);
```

### Sentencia for

Un bucle for es una sentencia iterativa que normalmente se utiliza con una variable que se incrementa o decremente.

### Sintaxis

```
for (inicialización; expresión-prueba; expresión-actualización)
{
 sentencias
}
```

La sentencia de *inicialización* puede ser o bien una declaración o una expresión. Normalmente se utiliza para inicializar una variable; sin embargo, puede ser nulo. La *expresión-prueba* sirve como control del bucle. Las iteraciones se ejecutan mientras que *expresión-prueba* se evalúe a verdadero. En cada iteración, se ejecuta la sentencia que puede ser una sola sentencia o una sentencia compuesta. Si la evaluación de *expresión-prueba* es falsa, la ejecución de sentencias no se realiza nunca. La *expresión\_actualización* se evalúa después de cada iteración del bucle; normalmente se utiliza para modificar la variable inicializada en la sentencia de inicialización. Si la primera evaluación de *expresión-prueba* es falsa, la *expresión\_actualización* nunca se evalúa.

```

1. for (i = 0; i < 10; ++i){
 cout << "i toma el valor: " << i << endl;
}

2. for (i = 1; i <= 10; i++)
 cout << i << " ";

```

### Ejemplo 16.19

En la manipulación de un *array* (**arreglo**) se pueden utilizar dos índices para intercambiar valores.

```

// intercambio de valores en un array
int array [50]; // array de 50 valores enteros

for (int i = 0, int j = 49; i < j; ++i, --j) {
 int aux = array[i];
 array[i] = array[j];
 array[j] = aux;
}

```

### Ejemplo 16.20

Al principio del bucle anterior, a *i* se asigna el valor 0 y a *j* el valor 49 (primer y último elementos del *array*). Mientras *i* sea menor que *j*, se ejecutan las sentencias del bucle. Después de cada iteración *i* se incrementa en 1 y *j* se decrementa en 1.

## Sentencias **break** y **continue**

Dos sentencias se pueden utilizar en los bucles para romper la secuencia normal de los mismos: **break** (ya utilizada en la sentencia **switch**) y **continue**.

**break** aborta un bucle inmediatamente (al igual que sucede en la sentencia **switch**); realiza un salto incondicional al final del bucle. Se puede utilizar **break** en cualquiera de los bucles, haciendo que la ejecución del programa salte a la siguiente sentencia después del bucle. La sentencia **continue** se utiliza en los bucles y hace que comience inmediatamente una nueva iteración del bucle saltando el resto del cuerpo del bucle que viene a continuación.

## Estructura de **break** y **continue**

El siguiente segmento de programa permite leer una línea de texto con la función `cin.get(car)`. El bucle repite cada carácter y utiliza **break** para terminar el bucle si el carácter es un salto de línea.

```

1. while (cin.get (car))
{
 sentencias
 if (car == '\n')
 continue;
 sentencias
}

```

**continue** salta el resto del bucle y comienza una nueva iteración.

```
2. while (cin.get (car)) {
 sentencias
 if (car == '\n')
 break; _____
 sentencias
}
sentencia _____
```

**break** salta el resto del bucle y va a la siguiente sentencia.

## 16.7 Funciones

Una **función** es un grupo de sentencias que realizan una operación o tarea específica. En la resolución de problemas tanto por métodos estructurados como por métodos orientados a objetos, se recurre a un proceso de refinamiento sucesivo que implica la descomposición de un problema o proceso, en subproblemas o subprocesos más pequeños y de menor complejidad.

Las funciones se utilizan para capturar y representar los procesos y subprocesos. El programa principal (`main`) de C++ es una secuencia de llamadas a funciones que pueden llamar a otras funciones.

C++ proporciona mecanismos para crear funciones definidas por el usuario así como proporcionar una biblioteca estándar de funciones, como la biblioteca estándar `stdio.h` de ANSI C, y las específicas de C++, como biblioteca de flujos, `stream`, `iostream`, etc.

## Funciones de biblioteca

La función `sqrt( )` devuelve la raíz cuadrada de un número y pertenece a la biblioteca estándar de funciones de C/C++. Para poder utilizar una función de biblioteca se debe incluir el archivo de cabecera `cmath` (`math.h` en los compiladores antiguos que siguen ANSI C o los primeros compiladores de C++). Una vez incluido el archivo de cabecera correspondiente a la biblioteca, se necesita proporcionar el prototipo. Cada función de la biblioteca C++ tiene un prototipo en uno o más archivos de cabecera. No se debe confundir el prototipo de la función con la definición de la función. El prototipo solo describe la interfaz de la función.

```
double sqrt (double);
```

es decir, describe la información enviada a la función y la información enviada de retorno. La **definición**, sin embargo, incluye el código que calcula la raíz cuadrada de un número. C++ separa estas dos características, prototipo y definición, de las funciones de biblioteca. Los archivos de biblioteca contienen el código compilado de las funciones mientras que los archivos de cabecera contienen prototipos.

La práctica usual es situar los prototipos antes de la definición de la función `main()`; de modo similar a:

```
#include <iostream>
#include <cmath> // o bien math.h, en sistemas antiguos
```

Una vez incluida la directiva correspondiente; ya se puede utilizar `sqrt( )`. Por ejemplo, la sentencia

```
x = sqrt(27.04) // devuelve el valor 5.2 y lo
 // asigna a x
```

La sintaxis a emplear es la siguiente: `sqrt (27.04)` **invoca** o **llama** a la función `sqrt()`. La expresión `sqrt(27.04)` se denomina **llamada a la función**, la función invocada se denomina **función llamada** y la función que contiene la llamada de la función se denomina **función llamadora**.

## Función Llamadora

```
int main ()
{
 ...
 ...
 x = sqrt (2
 ...
}
```

## Función llamada

```
 código de sqrt()
|
| ...
| ...
| ... función
| ...
| ...
```

```
// raizcua.cpp
#include <iostream>
#include <cmath>
using namespace std;

int main ()
{
 double superficie;
 cout << "Introducir la superficie de un cuadrado";
 cin >> superficie;
 double lado;
 lado = sqrt(superficie);
 cout << "el lado del cuadrado es " << lado;
 return 0;
}
```

### Ejemplo 16.21



## Definición de una función (función definida por el usuario)

Una función se representa por un nombre y un conjunto de tipos de operandos, denominados **parámetros**, que se escriben encerrados entre paréntesis separados por comas.

Las acciones que debe ejecutar la función se especifica en un bloque, conocido como **cuerpo de la función**. Cada función tiene un **tipo de retorno** asociado.

### Sintaxis

```
tipo nombre_función (lista_de_argumentos)
{
 Sentencias
}
```

Para poder utilizar una función se debe especificar su prototipo. Para que el compilador pueda compilar una llamada a una función primero debe conocer la declaración de esa función. Las declaraciones de las funciones normalmente aparecen antes de la función `main()` en un programa, pero también se pueden situar en archivos de inclusión.

Una función **mcd** (máximo común divisor de dos números).

### Ejemplo 16.22



```
// devuelve el mcd de dos números a y b
int mcd (int a, int b)
{
 while (b)
 {
 int aux = b;
 b = a % b;
 a = aux;
 }
 return a
}
```

Para **llamar a una función** se utiliza el operador de llamada que es el nombre de la función con una lista de argumentos, solo los nombres.

```
// obtener dos valores para deducir su mcd
cout << "Introducir dos números: \n";
int m, n;
cin >> m >> n;
// obtener el mcd
cout << "mcd: " << mcd(m,n) << endl;
```

La llamada de una función realiza dos cosas: inicializa los parámetros de la función a partir de los argumentos correspondientes y transfiere el control a la función invocada. La ejecución de la función **llamadora** se detiene y comienza la ejecución de la función **llamada**. La ejecución de una función comienza con la definición e inicialización de sus parámetros. En el caso de `mcd`, se crean las variables de tipo entero `m` y `n` y se inicializa con los valores introducidos por el operador `cin`. A continuación, se ejecuta el cuerpo del bucle y termina cuando se encuentra una sentencia `return` que produce el resultado especificado en ella. Después de ejecutar `return`, la función llamadora reanuda su ejecución en el punto de llamada.

El tipo de retorno de una función puede ser un tipo incorporado, como `int` o `double`, un tipo clase, o un tipo compuesto como `string`. También puede ser el tipo de retorno, en cuyo caso significa que no se devuelve ningún valor.

### Regla

Sintaxis del prototipo de una función:

**Tipo nombre (lista de argumentos);**

## Argumentos de la función: paso por valor y por referencia

En C++ los parámetros de una función son llamados **por valor** o **por referencia**. La llamada o paso por valor significa que el valor del argumento, es decir, una copia del objeto que representa el argumento, se pasa a la función donde se asigna a una nueva variable, pero aunque la función modifique el valor del parámetro, la modificación no se mantendrá después del retorno de la función. La declaración de un argumento que se pasa por valor se realiza, simplemente, declarando el tipo de argumento.

### Paso por valor

```
double cubo(double x);
int main ()
{
 ...
 double lado = 10;
 double volumen = cubo(lado); → paso por valor
 ...
}
double cubo(double n)
{
 return n*n*n;
}
```

### Paso por referencia

El paso por referencia de los argumentos permite a una función llamada acceder a variables de la función llamadora. C solo podría pasar argumentos por valor. En el paso por referencia la función puede modificar el valor del parámetro.

Un parámetro se declara que se pase por referencia declarando dicho parámetro como un tipo de referencia.

La función típica de intercambio de los valores de dos variables utilizada en algoritmos de ordenación.

### Ejemplo 16.23

```
void intercambio (int &m, int&n)
{
 int aux = m;
 m = n;
 n = aux;
}
```

La declaración de la función **intercambio** es

```
void intercambio (int&, int&);
```

En este caso cuando se hace una llamada

```
intercambio(a,b)
```

Los valores de **a** y **b**, se modifican dado que el valor de **a** se cambia por el de **b** y viceversa. En el caso del paso por valor de la función **mcd**, los parámetros **m** y **n** se intercambiarían en la ejecución de la función pero **a** y **b** mantendrían sus valores originales después de la llamada a la función.

### La sentencia return

La sentencia **return** termina la función que se está ejecutando actualmente y devuelve el control a la función que la llamó. Existen dos formatos de la sentencia **return**.

```
return;
return expresión;
```

Una sentencia **return** sin ningún valor, solo se puede utilizar en una función que tiene un tipo de retorno **void**. Las funciones que devuelven **void** no requieren contener una sentencia **return**, aunque cuando una función utiliza **return** es para producir una terminación prematura de la función.

El segundo formato de la sentencia **return** proporciona el resultado de la función. Cada retorno de una función con un tipo distinto de **void** debe devolver un valor. El valor devuelto debe tener el mismo tipo que el tipo de retorno de la función, o debe tener un tipo que pueda ser convertido implícitamente a este tipo.

Calcular el importe total de un conjunto de artículos de un supermercado, cada precio ha de ser incrementado con las tasas de impuestos (IVA).

### Ejemplo 16.24

```
double precio_total (int numero, double precio)
{
 const double IVA = 0.16; // 16% de impuesto
 double subtotal;
 subtotal = precio * numero;
 return (subtotal + subtotal * IVA);
```

### La sentencia using y el espacio de nombres std

La línea **using namespace std** indica al compilador que ponga todos los nombres del espacio de nombres predefinido **std** disponibles al programa. Un *espacio de nombres* es una colección de nombres o identificadores que se definen juntos. Un programa que contiene esta línea podrá acceder a todas las funciones e identificadores declarados en la biblioteca estándar.

Sin el uso de la directiva `using` se pueden referenciar los nombres declarados en un espacio de nombres especificando el espacio de nombres real antes de cada uso de un nombre de ese espacio de nombres. Por ejemplo, se puede utilizar `std::cout` para referenciar el nombre `cout`.

### Directiva `#include`

La directiva `#include` tiene uno de los dos formatos:

1. `#include <cabecera>`

2. `#include "nombre-archivo"`

El formato

`#include <cabecera>`

Se reserva para incluir definiciones que están en la biblioteca estándar y el formato

`#include "nombre-archivo"`

Se utiliza para incluir un archivo de inclusión definido por el usuario.



## Resumen

- Cada programa C++ contiene una o más funciones, una de las cuales se denomina `main()`. El sistema operativo ejecuta un programa llamando a la función `main()`.
- El *preprocesador* es una parte de compilador que procesa un archivo fuente antes de que tenga lugar la compilación. La directiva `#include archivo` es reemplazada por el contenido del archivo indicado.
- Un flujo (`stream`) es una secuencia de caracteres empleada para leer de un dispositivo o escribir en un dispositivo de E/S. Para manipular la entrada se utiliza un objeto del tipo `istream` (flujo de entrada) denominado `cin`. Para salida, se utiliza un objeto del tipo `ostream` denominado `cout`.
- C++ define varios tipos de datos, pero no especifica su tamaño. El tamaño dependerá del sistema operativo y de la plataforma sobre la que se ejecute su programa.
- C++ define un conjunto de tipos aritméticos: enteros, coma faltante, carácter individual y valores lógicos (booleanos). Además existe el tipo `void` que no tiene valores asociados.
- Una **referencia** es un nombre alternativo de un objeto. Las referencias se utilizan principalmente como pará-

metros formales a funciones. Una referencia es un **tipo compuesto** que se define precediendo a un nombre de variable el símbolo `&` (**operador de dirección**).

- C++ proporciona un conjunto muy rico de operadores y define cuáles de estos operadores se aplican a tipos fundamentales. Existen operadores **unitarios** que actúan sobre un solo operando (operador dirección `&`; operador de referencia `*`) y operadores **binarios** y **terciarios**, que actúan sobre dos o tres operandos (suma `+`, resta `-`...).
- Las funciones C++ son de dos tipos: aquellas que devuelven valores y aquellas que no devuelven valores. En C los parámetros de una función siempre se pasan por valor. En C++ los parámetros de una función son **por valor o por referencia**.
- La mayoría de los compiladores se ejecutan (**corren**) desde un entorno integrado de desarrollo (EID, IDE “Integrated Development Environment”) que contiene además del compilador, herramientas de análisis, desarrollo y depuración de programas. La mayoría de los entornos incluyen una interfaz que permite al programador escribir un programa y utilizar diferentes menús para compilar y ejecutar el programa.



## Ejercicios

16.1 ¿Qué salida producirá el siguiente código cuando se inserta en un programa completo?

```
int x = 2;
cout << "Arranque\n";
if (x <= 3)
 if (x != 0)
 cout << "Hola desde el segundo if.\n";
 else
 cout << "Hola desde el else.\n";
cout << "Fin\n";
cout << "Arranque de nuevo\n";
if (x > 3)
if (x != 0)
 cout << "Hola desde el segundo if.\n";
else
 cout << "Hola desde el else.\n";
cout << "De nuevo fin\n";
```

16.2 ¿Cuál es el error de este código?

```
cout << "Introduzca n:";
cin >> n;
if (n < 0)
 cout << "Este número es negativo. Pruebe
 de nuevo .\n";
 cin >> n;
else
 cout << "conforme. n= " << n << endl;
```

16.3 ¿Qué salida producirá el siguiente código, cuando se inserta en un programa completo?

```
int x = 5;
cout << "Arranque\n";
if (x <= 3)
 if (x != 0)
 cout << "Hola desde el segundo if.\n";
 else
 cout << "Hola desde el else.\n";
cout << "Fin\n";
cout << "Arranque de nuevo\n";
if (x)
 if (x != 0)
 cout << "Hola desde el segundo if.\n";
else
 cout << "Hola desde el else.\n";
cout << "De nuevo fin\n";
```

16.4 Seleccione y escriba el bucle adecuado que mejor resuelva las siguientes tareas:

- Suma de series como  $1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/50$
- Lectura de la lista de calificaciones de un examen de Historia

c) Visualizar la suma de enteros en el intervalo 11... 50.

16.5 ¿Cuál es la salida de este bucle?

```
int i = 1 ;
while (i * i < 10)
{
 int j = i;
 while (j * j < 100)
 {
 cout << i + j << " ";
 j *= 2;
 }
 i++;
 cout << endl;
}
cout << "\n*****\n";
```

16.6 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, 1) devuelve el valor 20.6.

16.7 ¿Cuál es la salida del código siguiente? (Se supone insertado en un programa completo y correcto.)

```
char simbolo[3] = { 'a', 'b', 'c', 'd' } ;
cout a[0] << " " << a[1] << " " <<
a[2] << endl ;
a[1] = a[2] ;
cout << a[0] << " " << a[1] << " " << a[2]
<< endl ;
```

16.8 ¿Cuál es la salida del siguiente código?

```
char simbolo[3] = {'a', 'b', 'c'} ;
for (int indice=0 ; indice < 3; indice++)
 cout << simbolo[indice] ;
```

16.9 Consideré la siguiente definición:

```
void triple (int &n)
{
 n = 3*n ;
}
```

¿Cuál de las siguientes llamadas es válida?

```
int a[3] = {4, 5, 6}, numero = 2 ;
triple (numero) ;
triple (a[2]) ;
triple (a[3]) ;
triple [a[numero]) ;
triple (a) ;
```

- 16.10 ¿Cuál es la salida del siguiente código?

```
int i, aux[10] ;
for (i=0 ; i<10 ; i++)
 aux[i] = 2*i ;
for (i=0 ; i<10 ; i++)
```

```
cout << aux[i] << " " ;
cout << endl ;
for (i=0 ; i<10 ; i = i + 2)
 cout << aux[i] << " " ;
```

## Problemas

16.1 Escribir y ejecutar un programa en C++ que simule un calculador simple. Lee dos enteros y un carácter. Si el carácter es un +, se visualiza la suma; si es un -, se visualiza la diferencia; si es un \*, se visualiza el producto; si es un /, se visualiza el cociente; y si es un % se imprime el resto.

16.2 Escribir un programa en C++ que lea un límite máximo entero positivo, una base entera positiva, y visualice todas las potencias de la base, menores que el valor especificado límite máximo.

16.3 Escribir un programa en C++ que lea dos números  $x$  y  $n$  y calcule la suma de la progresión geométrica:

$$1 + x + x^2 + x^3 + \dots + x^n$$

16.4 Escribir un programa que calcule los valores de la función definida de la siguiente forma:

```
funciony(0) = 0,
funciony(1) = 1
funciony(2) = 2
funciony(n) = funciony(n - 3) + 3*funciony(n - 2) - funciony(n - 1) si n > 2.
```

16.5 Diseñar un programa C++ 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).

16.6 Escribir una función con un argumento de tipo apuntador a `double` y otro argumento de tipo `int`. El primer argumento se debe de corresponder con un arreglo y el segundo con el número de elementos del arreglo. La función ha de ser de tipo apuntador a `double` para devolver la dirección del elemento menor.

16.7 Se dice que un vector es simétrico si el elemento que ocupa la posición  $i$ -ésima coincide con el que ocupa la posición  $n-i$ -ésima, siempre que el número de elementos que almacene el vector sea  $n$ . Por ejemplo el vector que almacena los valores 2, 4, 5, 4, 2 es simétrico. Escribir una función que decida si el vector de  $n$  datos que recibe como parámetro es simétrico.

16.8 Escribir una función que utilice apuntadores (punteros) para buscar la dirección de un entero dado en un arreglo dado. Si se encuentra el entero dado, la función devuelve su dirección; en caso contrario, devuelve `NULL`.

16.9 Una malla de números enteros representa imágenes, cada entero expresa la intensidad luminosa de un punto. Un punto es un elemento “ruido” cuando su valor se diferencia en dos o más unidades del valor medio de los ocho puntos que le rodean. Escribir un programa que tenga como entrada las dimensiones de la malla, reserve memoria dinámicamente para una matriz en la que se lean los valores de la malla. Diseñar una función que reciba una malla, devuelva otra malla de las mismas dimensiones donde los elementos “ruido” tengan el valor 1 y los que no lo son valor 0.

16.10 Escribir una función que decida si una matriz cuadrada de orden  $n$  es mayoritaria. “Se dice que una matriz cuadrada de orden  $n$  es mayoritaria si existe un elemento en la matriz cuyo número de repeticiones sobrepasa a la mitad de  $n \times n$ ”. El tratamiento debe hacerse con apuntadores.



# Clases y objetos. Sobrecarga de operadores

## Contenido

- 17.1 Clases y objetos
- 17.2 Definición de una clase
- 17.3 Constructores
- 17.4 Destructores
- 17.5 Sobrecarga de funciones miembro
- 17.6 Funciones amigas
- 17.7 Sobrecarga de operadores

- 17.8 Sobrecarga de operadores unitarios
- 17.9 Sobrecarga de operadores binarios
- 17.10 Conversión de datos y operadores de conversión de tipos
- 17.11 Errores de programación frecuentes
  - › Resumen
  - › Ejercicios
  - › Problemas

## Introducción

Hasta ahora hemos aprendido el concepto de estructuras, con lo que se ha visto un medio para agrupar datos. También se han examinado funciones que sirven para realizar acciones determinadas a las que se les asigna un nombre. En este capítulo se tratarán las clases, un nuevo tipo de dato cuyas variables serán objetos. Una **clase** es un tipo de dato que contiene código (funciones) y datos y permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. La sobrecarga de operadores hace posible manipular objetos con operadores estándar como `+`, `-`, `*`, `[ ]`. En el capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales y en capítulos posteriores se verá cómo definir y utilizar jerarquías y otras relaciones entre clases; también se mostrará la sobrecarga de operadores unitarios y binarios.

## Conceptos clave

- › `const`
- › Constructores
- › Conversión de tipos
- › Destructores
- › Encapsulamiento
- › Especificadores de acceso `public`, `protected`, `private`
- › Funciones miembro
- › Función operador
- › Miembros dato
- › Moldeado
- › Objetos
- › Ocultación de la información
- › Tipo de clase

## 17.1 Clases y objetos

Las tecnologías orientadas a objetos han evolucionado mucho pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas son los elementos clave sobre los que se articula la orientación a objetos.

## ¿Qué son objetos?

Martin y Odell definen un objeto como “cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos”. Para realizar esa actividad se añaden a cada objeto de la clase los propios datos asociados con sus propias *funciones miembro* que pertenecen a la clase.

Cualquier programa orientado a objetos puede manejar muchos objetos. Por ejemplo, un programa que maneja el inventario de un almacén de ventas al por menor, utiliza un objeto de cada producto manipulado en el almacén. El programa manipula los mismos datos de cada objeto, incluyendo el número de producto, descripción del producto, precio, número de artículos del *stock* y el momento de nuevos pedidos.

Cada objeto conoce también cómo ejecutar acciones con sus propios datos. El objeto producto del programa de inventario, por ejemplo, conoce cómo crearse a sí mismo y establecer los valores iniciales de todos sus datos, cómo modificar sus datos y cómo evaluar si hay artículos suficientes en el *stock* para cumplir una petición de compra. En esencia, la cosa más importante de un objeto es reconocer que consta de datos, y las acciones que pueden ejecutar.

Un objeto de un programa de computadora no es algo que se pueda tocar. Cuando un programa se ejecuta, la mayoría existe en memoria principal. Los objetos se crean por un programa para su uso mientras el programa se está ejecutando. A menos que se guarden los datos de un objeto en un disco, el objeto se pierde cuando el programa termina (este objeto se llama *transitorio* para diferenciarlo del objeto *permanente* que se mantiene después de la terminación del programa).

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una *interfaz* se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón “enviar”. El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el periodo de garantía puede anularla.

## ¿Qué son clases?

En términos prácticos, una *clase* es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como “un conjunto de objetos que comparten una estructura y comportamiento comunes”.

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios*, *métodos* o *funciones miembro*. El término *función miembro* se utiliza, específicamente, en C++. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos* o *variables*. El término *atributo* se utiliza en *análisis y diseño orientado a objetos* y el término *variable* se suele usar en programas orientados a objetos.

Las clases son la espina dorsal de la mayoría de los programas C++. C++ soporta la programación orientada a objetos con un modelo de objetos basado en clases. Es decir, una clase define el comportamiento y el estado de los objetos que son instancias de la clase.

Las clases definen nuestros propios tipos de datos que son personalizados a los problemas a resolver, obteniendo aplicaciones que son más fáciles de escribir y de comprender. Los tipos de clases bien diseñadas pueden ser tan fáciles de utilizar como los tipos incorporados.

Una clase define *miembros dato* y funciones. Los miembros dato almacenan el estado asociado con los objetos del *tipo de clase* y las funciones ejecutan operaciones que dan significado a los datos. Las clases pueden separar la interfaz y la implementación. Solo los implementadores de la clase necesitan conocer los detalles de la implementación; el usuario normalmente necesita conocer los detalles de la interfaz.

Los tipos de clases se conocen como tipos abstractos de *objetos*. Un TAD trata los datos (el estado) y las operaciones sobre el estado como única unidad: se precisa en modo abstracto lo que hace la clase en lugar de pensar en cómo funciona la clase internamente.

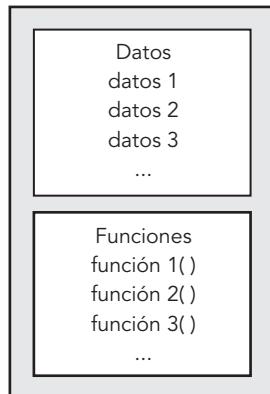


Figura 17.1 Las clases contienen datos y funciones.

Una *clase* es un medio para traducir la abstracción de un tipo definido por el usuario. Combina la representación de los datos miembros (*atributos*) y las funciones miembro (*métodos*) que manipulan esos datos en una única entidad o paquete que se trata como tal. La colocación de datos y funciones juntas en una sola entidad (*clase*) es una idea central en programación orientada a objetos. La figura 17.1 muestra la representación gráfica de una clase.

## 17.2 Definición de una clase

La primera operación que se debe realizar con una clase es definirla. La *definición* o *especificación* tiene dos partes:

- *Declaración de la clase*. Describe el componente datos, en términos de miembros dato y la interfaz pública, en términos de funciones miembro, también denominados métodos.
- *Definiciones de métodos de las clases*. Describen la implementación de las funciones miembro.

La declaración de una clase utiliza la palabra reservada `class`, seguida de la lista de miembros dato y métodos de la clase, encerrados entre llaves.

### Sintaxis

```
class NombreClase // Identificador válido
{
 declaraciones de datos // atributos
 declaraciones de funciones // métodos
} ; ←Punto y coma obligatorio
```

### Regla práctica

- Los atributos (*datos*) son variables simples (de tipo entero, estructuras, arrays, `float`, etc.).
- Los métodos (*funciones*) son funciones simples que actúan sobre los atributos (*datos*).
- La omisión del punto y coma detrás de la llave de cierre es un error frecuente, difícil de detectar.

En una clase, por convenio de dominio, los miembros están ocultos al exterior; es decir, los datos y funciones miembro son *privados* por omisión. La *visibilidad* permite controlar el acceso a los miembros de la clase, ya que solo se puede acceder a los miembros que son visibles desde el exterior y queda prohibido el acceso a aquellos miembros ocultos. Los *especificadores* o *duplicadores de acceso* son: `public` (*público*), `protected` y `private` (*privado*) (tabla 17.1).

**Tabla 17.1** Secciones pública y privada de una clase.

| Sección                          | Comentario                                                                                |
|----------------------------------|-------------------------------------------------------------------------------------------|
| <code>public (público)*</code>   | Se permite el acceso desde el exterior del objeto de la clase (visible desde el exterior) |
| <code>private (privado)**</code> | Se permite el acceso solo desde el interior del objeto (oculto al exterior)               |

\* A los miembros públicos se puede acceder desde cualquier objeto de la clase.

\*\* A los miembros privados solo se puede acceder desde métodos de la propia clase.

### Nota

Si la clase se define con la palabra reservada `struct`, por omisión todos los miembros de la estructura son visibles desde el exterior con el objeto de mantener la compatibilidad con C. Esta es la única diferencia existente en C++ entre palabras reservadas `class` y `struct`.

La sintaxis completa de una clase es:

```
class NombreClave
{
 public:
 Sección pública // declaración de miembros públicos
 private:
 Sección privada // declaración de miembros privado
};
```

El acceso restringido a los datos de la clase es una propiedad de la programación orientada a objetos: *la ocultación de datos*. El mecanismo principal para conseguir la ocultación de datos es ponerlos todos en una clase y hacerlos privados. A los datos o funciones privados solo se puede acceder desde el interior de la clase y a los datos y funciones públicas se puede acceder desde el exterior de la clase. La ocultación de datos es una técnica de programación que facilita a los programadores el diseño evitándoles errores que en programación estructurada suelen ser frecuentes. La figura 17.2 muestra gráficamente el funcionamiento de las secciones pública y privada de una clase.

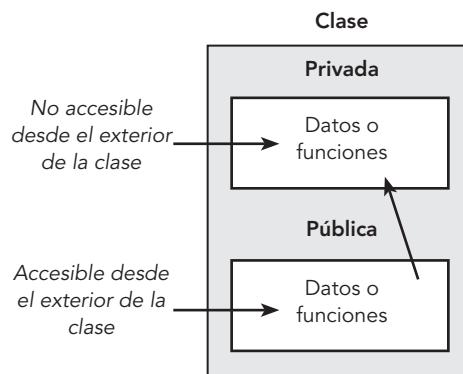


Figura 17.2 Secciones pública y privada de una clase.



### Ejemplo 17.1

Declaración de una clase Semaforo

```
clase Semaforo
{
 public:
```

```

void cambiarColor();
//...
private:
 enum Color {VERDE, ROJO, AMBAR};
 Color c;
};

```

En la clase `Semaforo` se observa que la sección pública declara la definición de los servicios (acción de cambio de color del semáforo) y en la sección privada, en último lugar, los detalles internos del semáforo (diferentes tipos de colores del semáforo).

### Reglas prácticas en la declaración de una clase

1. Las declaraciones (cabeceras de las funciones) de los métodos, normalmente, se colocan en la sección pública y las declaraciones de los datos (atributos), normalmente, se sitúan en la sección privada.
2. Es indiferente colocar primero la sección pública o la sección privada; pero la situación más usual es colocar la sección pública primero para resaltar las operaciones que forman parte de la interfaz pública del usuario.
3. Las palabras clave `public` y `private` seguidas de dos puntos, señalan el comienzo de las respectivas secciones públicas y privadas; aunque no se suele utilizar, una clase puede tener varias secciones pública y privada.
4. *Recuerde* poner el punto y coma después de la llave de cierre de la declaración de la clase.

### ¿Control de acceso a los miembros: público o privado?

Los miembros de una clase se pueden declarar públicos o privados. Para ello, el cuerpo de una clase contiene dos palabras clave reservadas: `private` y `public`; tanto los datos como las funciones miembro se pueden declarar en una u otra sección.

Sin embargo, uno de los principios fundamentales de la programación orientada a objetos es el *encapsulamiento (ocultación de la información)*; por esta razón, normalmente, los datos deben permanecer ocultos y por ello se deben poner en la sección privada. Por el contrario, las funciones miembro que constituyen la interfaz de la clase van en la sección pública, ya que en caso contrario no se podría llamar a estas funciones desde un programa. Normalmente, se utilizan funciones miembro privadas para manejar detalles de la implementación que no forman parte de la interfaz pública.

### Diversas clases con diferentes secciones

### Ejemplo 17.2

```

class DemoUno
{
 private
 float precio;
 char nombre[20];
 public:
 int calanlar(int)
 ...
 };
};

class DiaDelAnio
{
 public:
 void leer();
 void escribir();
 int mes;
 int dia;
 };

```

No se necesita utilizar la palabra reservado `private` en las declaraciones de la clase ya que en forma predeterminada el control de acceso para los elementos declarados a continuación de la llave de apertura es privado. Sin embargo, con frecuencia utilizaremos la etiqueta `private` con el objetivo de enfatizar el concepto de ocultación de datos.

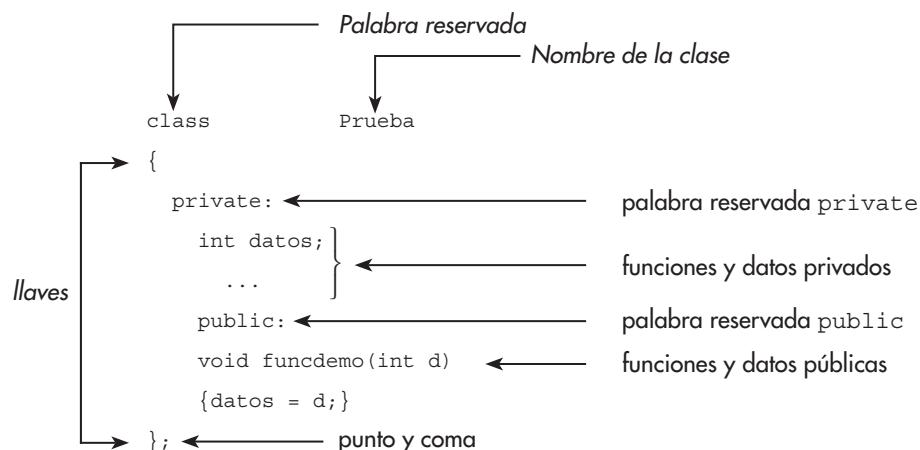


### Ejemplo 17.3

```
class DemoUno
{
 float precio;
 char nombre[20];
public:
 void calcular(int)
 ...
};
```

#### Las funciones son públicas y los datos privados

Normalmente, los datos dentro de una clase son privados y las funciones son públicas. Esto se debe al modo de uso de las clases. Los datos están ocultos para asegurarse frente a manipulaciones accidentales, mientras que las funciones que operan sobre los datos son públicos de modo que se pueda acceder a ellas desde el exterior de la clase. Sin embargo, no es una regla estricta y en algunas circunstancias se puede necesitar utilizar funciones privadas y datos públicos.



Para utilizar una clase se necesitan tres cosas:

1. ¿Cuál es su nombre?
2. ¿Dónde está definida?
3. ¿Qué operaciones admite?

El nombre de la clase, por ejemplo, `Semaforo`, está definida en un archivo de cabecera, que al igual que otros archivos tiene dos partes: un nombre de archivo y el sufijo (extensión) del archivo. De manera usual el nombre del archivo de la clase es el mismo que el nombre de la clase. La extensión, normalmente, es `.h`, pero algunos compiladores y programas pueden usar `.H`, `.hpp` o `.hxx`. En el ejemplo anterior, el archivo se denomina `Semaforo.h`.

Excepto en algunos casos especiales que se verán más adelante, en la declaración de la clase solo se colocan las cabeceras de los métodos. Sus definiciones se colocan, normalmente, en un archivo independiente llamado *archivo de implementación* cuyo nombre es el del archivo de cabecera correspondiente, pero su extensión es `.cpp` en lugar de `.h`. Por ejemplo, `Semaforo.cpp`. Los dos archivos constituyen una biblioteca de clases y los programas que utilizan la clase definida en la biblioteca se denominarán *programas cliente*. Estos programas deben incluir la cabecera de la biblioteca con la directiva del compilador:

```
#include "NombreClase.h"
#include "Semaforo.h"
```

**Nota**

Observe la diferencia entre

```
#include <iostream>
#include "Semaforo.h"
```

El primer caso indica al compilador de C++ que `iostream` es una biblioteca estándar; y el segundo formato con el nombre del archivo entre comillas, es una biblioteca definida por el programador.

Definición de una clase llamada `Punto` que contiene las coordenadas `x` y `y` de un punto en un plano.

**Ejemplo 17.4**

```
class Punto {
public:
 int Leerx(); // devuelve el valor de x
 void Fijarx(int); // establece el valor de x
private:
 int x; // coordenada x
 int y; // coordenada y
};
```

La definición de una clase no reserva espacio en memoria. El almacenamiento se asigna cuando se crea un objeto de una clase (*instancia* de una clase). Las palabras reservadas `public` y `private` se llaman *especificadores de acceso*.

Declaración de la clase `edad` (almacenada en el archivo `edad.h`).

**Ejemplo 17.5**

```
class edad
{
private:
 int edadHijo, edadMadre, edadPadre; //datos
public:
 edad();
 void iniciar (int, int, int); //función miembro
 int leerHijo();
 int leerMadre();
 int leerPadre();
};
```

Una *declaración* de una clase consta de una palabra reservada `class` y el nombre de la clase. Una declaración de la clase se utiliza cuando el compilador necesita conocer una determinada clase definida totalmente en alguna parte del programa. Por ejemplo,

```
class Punto; // definida en algún lugar
```

**Objetos de clases**

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*.

**Formato**

```
nombre_clase identificador;
```

Así, la definición de un objeto Punto es:

```
Punto P;
```

Cada clase define un tipo. El nombre del tipo es el mismo que el nombre de la clase. Por consiguiente, la clase Semaforo define un tipo denominado Semaforo. Al igual que sucede con los tipos de datos incorporados (propios) del lenguaje, se puede definir una variable de un tipo:

```
Semaforo S1;
```

en donde se indica que S1 es un objeto del tipo Semaforo.

Un programa que puede utilizar datos del objeto Semaforo puede ser:

```
#include <iostream>
using namespace std;

#include "Semaforo.h"
int main ()
{
 Semaforo S1;
 ...
 cin >> S1;
 ...
 cout << S1 << endl;
 return 0;
}
```

Un objeto tiene la misma relación con su clase que una variable tiene con un tipo de dato. Se dice que un objeto es una instancia (un *ejemplar*) de una clase, de igual modo que un Seat Ibiza es una instancia de un vehículo (carro).

El valor de una variable de un tipo clase se llama *objeto* (en general, cuando se hable de una variable de un tipo clase, normalmente nos estamos refiriendo a un objeto). Un objeto tiene miembros función y miembros dato. En programación con clases, un programa se visualiza como una colección de objetos que interactúan. Los objetos pueden interactuar ya que son capaces de acciones, es decir, invocaciones de funciones miembro. Las variables de un tipo clase se declaran de igual modo que las variables de tipos predefinidos y de igual modo que las variables estructura.

### A recordar

La definición de objetos significa crear esos objetos. Este proceso se llama también *instanciación*. El término *instanciar* (del inglés *instantiate*) se debe a que se crea una instancia (un ejemplar, un caso específico de una clase). Los objetos se denominan, a veces, *variables de instancia* (*instance variables*).

El *operador de acceso* a un miembro (.) selecciona un miembro individual de un objeto de la clase. Las siguientes sentencias, por ejemplo, crean un punto P, que fija su coordenada x y la visualiza a continuación.

```
Punto p;
p.Fijarx (100);
cout << " coordenada x es " << p.Leerx();
```

El operador punto se utiliza con los nombres de las funciones miembro para especificar que son miembros de un objeto.

### Ejemplo

```
class DiaSemana; // contiene una función Visualizar
 DiaSemana Hoy; // Hoy es un objeto
 Hoy.visualizar(); // ejecuta la función visualizar
```

Se puede asignar un objeto de una clase a otro; por defecto, C++ realiza una copia bit a bit de todos los miembros dato. En otras palabras, todos los miembros físicamente contenidos en el área de datos del objeto fuente se copian en el objeto receptor. Por ejemplo, el siguiente código crea un punto (Punto) llamado P2 y lo inicializa con el contenido de P:

```
Punto P;
// ...
Punto P2;
P2 = P;
```

## Acceso a miembros de la clase: encapsulamiento

Un principio fundamental en programación orientada a objetos es la *ocultación de la información* que significa que a determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos privados. A los datos o funciones privados solo se puede acceder desde dentro de la clase. Por el contrario, los datos o funciones públicos son accesibles desde el exterior de la clase.

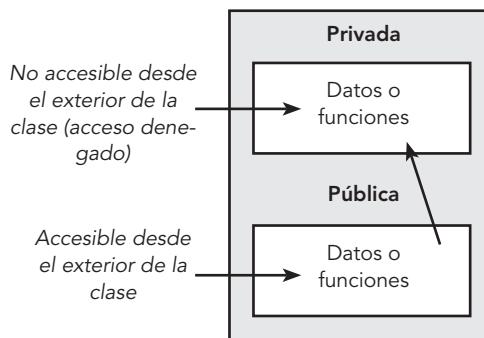


Figura 17.3 Secciones pública y privada de una clase.

Se utilizan tres diferentes *especificadores de acceso* para controlar el acceso a los miembros de la clase son: `public`, `private` y `protected`. Se utiliza el siguiente formato general en definiciones de la clase, situando primero las funciones miembro públicas, seguidas por los miembros protegido y privado (este orden, sin embargo, no es obligatorio).

### Formato

```
class nombre_clase {
public:
 / / miembros públicos
protected:
 / / miembros protegidos
private:
 / / miembros privados
};
```

El especificador `public` define miembros públicos, que son aquellos a los que se puede acceder por cualquier función. A los miembros que siguen al especificador `private` solo se puede acceder por funciones miembro de la misma clase o por funciones y clases amigas.<sup>1</sup> A los miembros que siguen al especificador `protected` se puede acceder por funciones miembro de la misma clase o de clases derivadas de la misma, así como por *amigas*. Los especificadores `public`, `protected` y `private` pueden aparecer en cualquier orden.

<sup>1</sup> Las funciones y clases amigas se estudian más adelante.

**Tabla 17.2 / Visibilidad.**

| Tipo de miembro | Miembro de la misma clase | Amiga | Miembro de una clase derivada | Función no miembro |
|-----------------|---------------------------|-------|-------------------------------|--------------------|
| private         | x                         | x     |                               |                    |
| protected       | x                         | x     | x                             |                    |
| public          | x                         | x     | x                             | x                  |

En la tabla 17.2 cada “x” indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

Si se omite el especificador de acceso, el acceso es privado. En la siguiente clase `Estudiante`, por ejemplo, todos los datos son privados, mientras que las funciones miembro son públicas.

```
class Estudiante {
 long numId;
 char nombre[40];
 int edad;
public:
 long LeerNumId();
 char * LeerNombre();
 int LeerEdad();
};
```

El mismo especificador de acceso puede aparecer más de una vez en una definición de una clase, pero, en este caso, no es fácil de leer.

```
class Estudiante{
private:
 long numId;
public:
 long LeerNumId();
private:
 char nombre[40];
 int edad;
public:
 char * LeerNombre();
 int LeerEdad();
};
```

### A recordar

El especificador de acceso se aplica a *todos* los miembros que vienen después de él en la definición de la clase (hasta que se encuentra otro especificador de acceso).

Aunque las secciones públicas y privadas pueden aparecer en cualquier orden, en C++, los programadores suelen seguir algunas reglas en el diseño que citamos a continuación, y de entre las cuales puede elegir la que considere más eficiente.

1. Poner la sección privada primero, debido a que contiene los atributos (datos).
2. Poner la sección pública primero debido a que las funciones miembro y los constructores son la interfaz del usuario de la clase.

La regla 2 presenta realmente la ventaja de que los datos son algo secundario en el uso de la clase y con una clase definida en forma adecuada, realmente no se suele necesitar nunca ver cómo están declarados los atributos.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con

un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, las funciones miembro y los miembros datos de la *sección pública* forman la interfaz externa del objeto, mientras que los elementos de la *sección privada* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

### A recordar

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

### Nota

C++ proporciona un especificador de acceso, `protected`. La explicación detallada de este tercer especificador se explicará en el capítulo 18 ya que se requiere el conocimiento del concepto de herencia.

## Datos miembros (miembros dato)

El concepto de **miembro de datos** o **dato miembro** es el mismo que el de atributo del objeto. Los miembros dato pueden ser de cualquier tipo válido definido, con excepción del propio tipo de la clase que se está definiendo, ya que está incompleto; es posible, sin embargo, emplear punteros (apuntadores) o referencias al tipo.

Los miembros dato se pueden declarar constantes. Si un miembro dato es `const` se indica que su valor, establecido durante la constitución del objeto, no puede ser modificado con posterioridad.

Los datos de una clase pueden existir en cualquier número, al igual que pueden existir cualquier número de elementos dato en una estructura. Normalmente, los datos de una clase son privados y van situados a continuación de la palabra reservada `private`, de modo que se pueden acceder a ellos desde el interior de la clase pero no desde el exterior.

Las *funciones miembro* son funciones que están incluidas dentro de una clase (en algunos lenguajes de programación como Smalltalk, a estas funciones se les denomina *métodos* y por analogía, a veces, en C++ también se les llama *métodos*).

La definición de una función miembro debe incluir el nombre de la clase ya que puede darse el caso de que dos o más clases tengan funciones miembro con el mismo nombre. Suponga la clase

```
class DiaDeSemana {
public:
 void salida ();
 int semana;
 int dia;
};
```

En un programa puede existir una sola clase como `DiaDeSemana`, pero en otros programas puede haber muchas definiciones de clases y más de una clase puede tener funciones miembro con el mismo nombre. La definición de una función miembro es similar a la definición de una función ordinaria excepto que se debe especificar el nombre de la clase en la cabecera de la definición de la función. Así en el caso de la función `salida`, la cabecera de su definición es:

```
void DiaDeSemana :: salida ()
```

Las funciones miembro se definen de modo similar a las funciones ordinarias. Al igual que cualquier otra función, una función miembro consta de cuatro partes:

- Un tipo de retorno de la función.
- El nombre de la función.
- Una lista de parámetros (puede ser vacía) separadas por comas.
- El cuerpo de la función que está contenido entre una pareja de símbolos llave (`{ }` ).

Las tres primeras partes constituyen el prototipo de la función. Este prototipo define toda la información de los tipos relativos a la función: su tipo de retorno, nombre de la función y el tipo de argumen-

tos que se pueden pasar a la misma. El prototipo de la función *debe estar definido* dentro del cuerpo de la clase. Sin embargo, la definición de la función miembro puede hacerse dentro de la propia clase o fuera del cuerpo de la clase.



### Ejemplo 17.6

Declaración de una clase `Articulo_Ventas`

```
class Articulo_Ventas {
public:
 // operaciones sobre objetos de la clase
 double precio_medio() const;
 bool articulo_igual (const Articulo_Ventas & art) const
 {return iva == art.iva; }
 // miembros privados
private:
 // ...
};
```

Se deben declarar todos los miembros de una clase dentro de las llaves que delimitan la definición de la clase. No existe ningún medio para añadir miembros a la clase. Los miembros que son funciones deben estar definidos y declarados. Se puede definir una función miembro dentro o fuera de la clase. En el ejemplo de la clase `Articulo_Ventas`, la función `articulo_igual` se define dentro de la clase, mientras que la función `precio_medio` se declara dentro de la clase pero se define en otro sitio.

Una función miembro que se declara dentro de una clase se considera implícitamente como función en línea. Es posible *declarar* una función dentro de una clase pero *definirla* en cualquier otro sitio. Las funciones definidas fuera de la clase no están, normalmente, en línea.



### Ejemplo 17.7

```
class DemoFunc
{
public:
 void leerdatos (int d)
 {datosprueba = d; }
 void verdatos ()
 {cout << "\n los datos son: " << datosprueba; }
};
```

Obsérvese que las funciones miembro `leerdatos( )` y `verdados( )` son *definiciones* en las cuales el código real de la función está contenido en la definición de la clase. (Recuerde que las funciones no se definen al estilo de las variables que reservan espacio de memoria automáticamente.)

#### Definición de una función miembro fuera de la clase

Las funciones miembro definidas fuera de la definición de la clase deben indicar que son miembros de la clase. En este caso, como ya se ha comentado, la función se declara dentro de la clase con una sintaxis similar a:

```
tipo nombre (parámetros);
```

Este prototipo indica al compilador que la función es un miembro de la clase pero que se definirá fuera de la declaración de la clase, en alguna parte del listado. La sintaxis de la definición de la función requiere el operador de resolución de ámbito (:) y es la siguiente:

```
tipo_devuelto Nombre_clase :: Nombre_función (lista_parámetros)
{
 sentencias del cuerpo de la función
}
Lista_parametros : lista de declaración de parámetros
```

```
void DiaDeSemana :: salida ()
{
 switch (semana)
 {
 case 1:
 cout << "Lunes"; break;
 case 2:
 cout << "Martes"; break;
 ...
 case 7:
 cout << "Domingo"; break;
 default:
 cout << "Error en DiaDeSemana -- ";
 }
 cout << dia;
}
```

### Ejemplo 17.8



## Funciones miembro

En la figura 17.4 se muestra la definición completa de una clase.

```
class Producto {
private:
 int num_prod;
 char nombre_prod [30], descrip_prod [80];
 float precio_produc, num_unidades;
public:
 Producto (int, char [], float, float);
 void verProducto();
 float obtenerPrecio();
 void actualizarProd(int);
};
```

Nombre de la clase

Acceso para almacenamiento de datos

Declaraciones para almacenamiento de datos

Acceso para funciones

Declaración de funciones

Figura 17.4 Definición típica de una clase.

La clase `Punto` define las coordenadas de un punto en un plano. Por cada dato se proporciona una función miembro que devuelve su valor y una que fija su valor.

### Ejemplo 17.9



```
class Punto {
public:
 int LeerX() { return x; }
 int LeerY() { return y; }
 void FijarX (int valx) { x = valx; }
}
```

```

 void FijarY (int valy) { y = valy; }
private:
 int x;
 int y;
};
```



### Ejemplo 17.10

```

void DiaAnyo:: visualizar()
{
 cout << "mes = " << mes
 << ", dia = " << dia << endl;
}
```

Las declaraciones de las funciones en el ejemplo 17.9 son también definiciones ya que se ha incluido el cuerpo de cada función. Si un cuerpo de la función consta de una única sentencia, muchos programadores sitúan el cuerpo de la función en la misma línea que el nombre de la función. Por otra parte, si una función contiene múltiples sentencias, cada una debe ir en una línea independiente. Por ejemplo:

```

class Punto {
public:
 void FijarX(int valx)
 {
 if ((valx >= -100) && (valx <= 100))
 x = valx;
 else
 cerr << "Error-: Fijarx() argumento fuera de rango";
 }
 // ...
};
```

11

### Ejercicio 17.1

Definir una clase que represente una radio que emita con tecnología MP3.

```

class radioMP3 {
private:
 int frecuencia;
 int volumen;
public:
 void iniciar(void);
 void aumentarFrecuencia(void);
 void disminuirFrecuencia(void);
 void bajarVolumen(void);
 void subirVolumen(void);
};

void radioMP3 :: iniciar(void)
{
 // inicializar atributos de frecuencia y volumen
 frecuencia = 99.99;
 volumen = 45;
}

void radioMP3 :: SubirVolumen(void)
{
```

```

 // Incrementar en una unidad el volumen
 volumen++;
}

void radioMP3 :: bajarVolumen(void)
{
 // Disminuir en una unidad el volumen
 volumen--;
}

void radioMP3 :: disminuirFrecuencia(void)
{
 // disminuir frecuencia 1 MHz
 frecuencia--;
}

void radioMP3 :: aumentarFrecuencia(void)
{
 // aumentar la frecuencia 1 MHz
 frecuencia++;
}

```

Un programa `main` que utilice las funciones de `radioMP3` podría ser:

```

int main (void)
{
 radioMP3 miEstacion;
 miEstacion.iniciar();
 miEstacion.subirVolumen();
 miEstacion.bajarVolumen();
 miEstacion.aumentarFrecuencia();
 miEstacion.disminuirFrecuencia();
 return 0;
}

```

## Llamadas a funciones miembro

Las funciones miembro de una clase se invocan o llaman mediante el operador punto (.) con la siguiente sintaxis:

`nombreObjeto.nombreFunción (valores de los parámetros)`

```

class Demo
{
private:
 // ...
public
 void func1(int P1)
 {...}
 void func2(int P2)
 {...}
};

Demo d1, d2; / / definición de los objetos d1 y d2
...
d1.func (2005);
d2.func (2010);

```

» **Ejemplo 17.11** 

La función `func1()` es una función miembro de la clase `Demo` y se debe llamar siempre en conexión con un objeto de esa clase. No tiene sentido una llamada directa `func1(2005)`, además produce un error. Una función miembro se llama para actuar sobre un objeto específico y no sobre la clase en general. Para utilizar una función miembro, el operador punto `(.)` (denominado operador de acceso a miembros de una clase) conecta el nombre del objeto y la función miembro.

### Regla

A las funciones miembro de una clase solo se puede acceder por un objeto de esa clase.

### Mensajes

Algunos lenguajes orientados a objetos denominan a las invocaciones a las funciones miembro como *mensajes*. Por consiguiente la llamada

```
d1.func1();
```

representa el *envío de un mensaje* al objeto `d1` para que ejecute una tarea determinada. El término *mensaje* no es un término formal en C++, pero es útil para interpretar la tecnología orientadas a objetos. En el caso de C++ se enfatiza en que los objetos son entidades discretas y nos comunicamos con ellos llamando a sus funciones miembro.



### Ejercicio 17.2

Demostración del uso de objetos

```
#include <iostream>
using namespace std;

class demoObj {
private:
 int undato; //datos de la clase
public:
 void fijardatos(int d) //establecer datos
 {
 undato = d;
 }
 void mostrardatos() //visualizar datos
 {
 cout << "El dato es: " << undato << endl;
 }
};

int main()
{
 demoObj d1, d2;
 d1.fijardatos(2005);
 d2.fijardatos(2010);

 d1.mostrardatos();
 d2.mostrardatos();
 return 0;
}
```

### Tipos de funciones miembro

Las funciones miembro que pueden aparecer en la definición de una clase se clasifican en función del tipo de operación que representan.

- *Constructores y destructores* son funciones miembro a las que se llama automáticamente cuando un objeto se crea o se destruye.
- *Selectores* que devuelven los valores de los miembros dato.

- *Modificadores o mutadores* que permiten a un programa cliente cambiar los contenidos de los miembros dato.
- *Operadores* que permiten definir operadores estándar C++ para los objetos de las clases.
- *Iteradores* que procesan colecciones de objetos, como arreglos y listas.

## Funciones en línea y fuera de línea

Hasta este momento, todas las funciones miembro se han definido dentro del cuerpo de la definición de la clase. Se denominan definiciones de funciones en *línea* (*inline*). Para el caso de funciones más grandes, es preferible codificar solo el prototipo de la función dentro del bloque de la clase y codificar la implementación de la función en el exterior. Esta forma permite al creador de una clase ocultar la implementación de la función al usuario de la clase proporcionando solo el código fuente del archivo de cabecera, junto con un archivo de implementación de la clase precompilada.

En el siguiente ejemplo, *FijarX* de la clase *Punto* se declara pero no se define en la definición de la clase:

```
class Punto {
public:
 void FijarX(int valx);
private:
 int x;
 int y;
};
```

### Ejercicio 17.3

Definir una clase *DiaAnyo* que contiene los atributos *mes* y *día* y una función miembro *visualizar*. El mes se registra como un valor entero en *mes* (1, enero, 2, febrero, etc.). El día del mes se registra en la variable entera *día*. Escribir un programa que haga uso de la clase y ver su salida.

```
// Uso de la clase DiaAnyo
#include <iostream>
using namespace std;

class DiaAnyo
{
public:
 void visualizar();
 int mes;
 int dia;
};

// programa que usa DiaAnyo
int main()
{
 DiaAnyo hoy, cumpleanyos;
 cout << "Introduzca fecha del dia de hoy\n";
 cout << "Introduzca el numero del mes:";
 cin >> hoy.mes;
 cout << "Introduzca el dia del mes: ";
 cin >> hoy.dia;
 cout << " Introduzca su fecha de nacimiento:\n";
 cout << " Introduzca el numero del mes:";
 cin >> cumpleanyos.dia;
 cout << "Introduzca el dia del mes: ";
 cin >> cumpleanyos.mes;
```

```

cout << " La fecha de hoy es ";
hoy.visualizar(); // llamada a la función visualizar
cout << " su fecha de nacimiento es ";
cumpleanos.visualizar();

if(hoy.mes == cumpleanos.mes
 && hoy.dia == cumpleanos.dia)
 cout << "¡Feliz cumpleaños! \n";
else
 cout << "¡Feliz día! \n";
return 0;
}

```

### Ejecución del programa

```

Introduzca fecha del día de hoy:
Introduzca el número del mes : 08
Introduzca el día del mes : 10
Introduzca su fecha de nacimiento:
Introduzca el número del mes : 2
Introduzca el día del mes : 25
La fecha de hoy es mes = 8, día = 10
Su fecha de nacimiento es mes = 2, día = 25
¡Feliz día!

```

La implementación de una función miembro externamente a la definición de la clase, se hace en una definición de la función *fueras de línea*. Su nombre debe ser precedido por el nombre de la clase y el signo de puntuación :: denominado *operador de resolución de ámbito*. El operador :: permite al compilador conocer que `FijarX` pertenece a la clase `Punto` y es, por consiguiente, diferente de una función global que pueda tener el mismo nombre o de una función que tenga ese nombre que puede existir en otra clase. La siguiente función global, por ejemplo, puede coexistir dentro del mismo ámbito que `Punto::FijarX`:

```

void FijarX(int valx)
{
 / / ...
}

```

### Formato

El símbolo :: (operador de resolución de ámbitos) se utiliza en sentencias de ejecución que accede a los miembros estáticos de la clase. Por ejemplo, la expresión `Punto::x` se refiere al miembro dato estático `x` de la clase `Punto`.

### La palabra reservada `inline`

La decisión de elegir funciones en línea y fuera de línea es una cuestión de eficiencia en tiempo de ejecución. Una función en línea se ejecuta normalmente más rápida, ya que el compilador inserta una copia “fresca” de la función en un programa en cada punto en que se llama a la función. La definición de una función miembro en línea no garantiza que el compilador lo haga realmente en línea; es una decisión que el compilador toma, basado en los tipos de las sentencias dentro de la función y cada compilador de C++ toma esta decisión de modo diferente.

Si una función se compila en línea, se ahorra tiempo de la CPU al no tener que ejecutar una instrucción *call* (llamar) para bifurcar a la función y no tener que ejecutar una instrucción de *return* para retornar al programa llamador. Si una función es corta y se llama cientos de veces, se puede apreciar un incremento en eficiencia cuando actúa como función en línea.

Una función localizada fuera del bloque de la definición de una clase se puede beneficiar de las ventajas de las funciones en línea si está precedida por la palabra reservada `inline`:

```
inline void Punto::FijarX (int valx)
{
 x = valx;
}
```

Dependiendo de la implementación de su compilador, las funciones que utilizan la palabra reservada `inline` se pueden situar en el mismo archivo de cabecera que la definición de la clase. Las funciones que no utilizan `inline` se sitúan en el mismo módulo de programa, pero no en el archivo de cabecera. Estas funciones se sitúan en un archivo `.cpp` (`.cc`, `.cxx`, `.c`, etc.).

## Nombres de parámetros de funciones miembro

Al igual que sucede con los prototipos de funciones globales, se pueden omitir los nombres de parámetros de una declaración de funciones miembro e identificar solo los tipos de parámetros. Por ejemplo:

```
class Punto {
public:
 void FijarX(int);
 // ...
};
```

Sin embargo, esta situación no siempre es deseable, ya que como la definición de la clase es también la interfaz de la clase, una función miembro sin más información que los tipos de datos de parámetros no proporcionará información suficiente sobre cómo llamar a la función:

```
class Demo {
public :
 FuncionDemo(int, float, char * ,int);
 // ...
};
```

### Regla

Si los nombres de los parámetros aparecen tanto en la declaración como en la implementación de la función, no es necesario que los nombres sean idénticos pero su orden y tipo sí deben serlo:

```
class Punto{
public:
 void Girar (int valx, int valy);
 //...
};

void Punto::Girar (int x, int y)
{
 //...
}
```

### Consejo

Es conveniente incluir comentarios de una o dos líneas en la declaración de una función que indiquen los que hace la función y cuáles son los valores de entrada/salida correspondientes.

## Implementación de clases

El código fuente para la implementación de funciones miembro de una clase una vez compilado se convierte en código ejecutable. Se almacena, por consiguiente, en archivos de texto con extensiones `.cp` o `.cpp`. Normalmente, se sitúa la implementación de cada clase en un archivo independiente.

Cada implementación de una función tiene la misma estructura general. Obsérvese que una función comienza con una línea de cabecera que contiene, entre otras cosas, el nombre de la función y su cuerpo está acotado entre una pareja de signos llave.

Las clases pueden proceder de diferentes fuentes:

- El programador puede declarar e implementar sus clases propias. El código fuente siempre estará disponible.
- Se pueden utilizar clases que hayan sido escritas por otras personas o incluso que se han comprado. En este caso, se puede disponer del código fuente o estar limitado a utilizar el código objeto de la implementación.
- Se pueden utilizar clases de las bibliotecas del programa que acompañan a su software de desarrollo C++. La implementación de estas clases se proporcionan normalmente como código objeto.

De cualquier forma, se debe disponer de las versiones de texto de las declaraciones de clase para que pueda utilizarlas su compilador.

## Archivos de cabecera y de clases

Las declaraciones de clases se almacenan normalmente en sus propios archivos de código fuente, independientes de la implementación de sus funciones miembro. Estos son los *archivos de cabecera* que se almacenan con una extensión .h o bien .hpp en el nombre del archivo. Aunque como ya conoce el lector, la versión estándar ANSI/ISO no requiere, normalmente, las extensiones .h ni .hpp en los nombres de los archivos de cabecera.

El uso de archivos de cabecera tiene un beneficio muy importante: “Se puede tener disponible la misma declaración de clases a muchos programas sin necesidad de duplicar la declaración”. Esta propiedad facilita la reutilización en programas C++.

Se utiliza una directiva del compilador, la directiva #include, para incorporar el archivo de cabecera con la declaración de la clase. Esta declaración de inclusión se pone en el archivo que implementa las funciones de la clase; también en los archivos de programa que utilizan la clase para crear objetos.

```
//Declaración de una clase almacenada en Demo1.h
class Demo1
{
public:
 Demo1();
 void ejecutar();
};
```

```
// Declaración de la clase edad almacenada en edad.h
class Edad
{
private:
 int edadHijo, edadPadre, edadMadre;
public:
 Edad();
 void iniciar(int, int, int);
 int obtenerHijo();
 int obtenerPadre();
 int obtenerMadre();
};
```

Figura 17.5 Listado de declaraciones de clases.

La directiva que mezcla el contenido de un archivo de cabecera en un archivo que contiene el código fuente de una función es:

```
#include "nombre-archivo"
```

### Opciones de compilación

La mayoría de los compiladores soporta dos versiones ligeramente diferentes de esta directiva. La primera instruye al compilador a que busque el archivo de cabecera en un directorio de disco que ha sido designado como el depósito de archivos de cabecera.

#### Ejemplo

```
#include <iostream>
utiliza la biblioteca de clases que soporta E/S
```

La segunda versión se produce cuando el archivo de cabecera está en un directorio diferente; entonces, se pone el nombre del camino entre dobles comillas.

#### Ejemplo

```
#include "\mi_cabecera\cliente.h"
```

## 17.3 Constructores

Un *constructor* es una función miembro de propósito específico que se ejecuta automáticamente cuando se crea un objeto de una clase. Un constructor sirve para inicializar los miembros dato de una clase.

Un constructor tiene el mismo nombre que la propia clase. Cuando se define un constructor no se puede especificar un valor de retorno, ni incluso `void` (un constructor nunca devuelve un valor). Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más).

Un objeto comienza a existir cuando se crea, y en un momento determinado deja también de existir. Normalmente, las funciones miembro de una clase se utilizan para dar valores a los elementos dato de un objeto. Sin embargo, es conveniente, a veces, que un objeto se pueda inicializar por sí mismo cuando se crea, sin necesidad de una llamada independiente a una función miembro. La inicialización automática se ejecuta utilizando una función miembro llamada *constructor*.

Un **constructor** es una función miembro que se ejecuta automáticamente siempre que se crea un objeto. Dicho de otro modo, en el momento que se crea un objeto se invoca automáticamente al constructor y en el momento que deja de existir se invoca a otra función miembro llamada *destructor*.

Las función constructor se utiliza normalmente para **inicializar** los atributos, mientras que la función destructor se suele utilizar para liberar memoria que haya sido reservada con anterioridad.

### Reglas

- El constructor tiene el mismo nombre que la clase.
- Puede tener cero, o más parámetros.
- No devuelve ningún valor.

La clase `Rectángulo` tiene un constructor con cuatro parámetros.

#### Ejemplo 17.12

```
class Rectángulo
{
 private:
 int izdo;
 int superior;
 int dcha;
 int inferior;
 public:
 // Constructor
 Rectángulo(int i, int s, int d, int inf);
 // definiciones de otras funciones miembro
};
```

Cuando se define un objeto, se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal de la función:

```
Rectangulo Rect(25, 25, 75, 75);
```

Esta definición crea una instancia del objeto `Rectangulo` e invoca al constructor de la clase pasándole los parámetros con valores especificados.

### Caso particular

Se pueden también pasar los valores de los parámetros al constructor cuando se crea la instancia de una clase utilizando el operador `new`:

```
Rectangulo *Crect = new Rectangulo(25, 25, 75, 75);
```

El operador `new` invoca automáticamente al constructor del objeto que crea (esta es una ventaja importante de utilizar `new` en lugar de otros métodos de asignación de memoria como la función `malloc`).

### Constructor por defecto

Un constructor que no tiene parámetros se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato asignándoles valores por defecto.



#### Ejemplo 17.13

El constructor por defecto de `Punto` inicializa `x` y `y` a 0.

```
class Punto {
public:
 Punto() // constructor
 {
 x = 0;
 y = 0;
 }
private:
 int x;
 int y;
};
```

Una vez que se ha declarado un constructor, cuando se declara un objeto `Punto` sus miembros dato se inicializan a 0. Esta es una buena práctica de programación.

```
Punto P1; // P1.x = 0, P1.y = 0
```

Si `Punto` se declara dentro de una función, su constructor se llama tan pronto como la ejecución del programa alcanza la declaración de `Punto`:

```
void FuncDemoConstructorD()
{
 Punto Pt; // llamada al constructor
 // ...
}
```

### Regla

C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Sin embargo, tal constructor no inicializa los miembros dato de la clase a un valor previsible, de modo que siempre es conveniente al crear su propio constructor por defecto, darle la opción de inicializar los miembros dato con valores previsibles.

### Precaución

Tenga cuidado con la escritura de la siguiente sentencia:

```
Punto P ();
```

Aunque parece que se realiza una llamada al constructor por defecto, lo que se hace es declarar una función de nombre `P` que no tiene parámetros y devuelve un resultado de tipo `Punto`.

## Constructores alternativos

Como ya se ha visto anteriormente es posible pasar argumentos a un constructor asignando valores específicos a cada miembro dato de los objetos de la clase. Un constructor con parámetros se denomina *constructor alternativo*.

### Ejemplo

```
Punto P(50, 250); // define e inicializa P
```

Un constructor se puede llamar directamente, creando un objeto temporal.

### Ejemplo

```
Punto x = Punto(50, 250);
```

se construye un punto y se asigna a `x`.

Otro constructor alternativo de la clase `Punto`, que visualiza un mensaje después de ser llamado.

### Ejemplo 17.14

```
Punto::Punto(int valx, int valy)
{
 FijarX(valx);
 FijarY(valy);
 cout << "Llamado el constructor de Punto .\n";
}
```

## Constructores sobrecargados

Al igual que se puede sobrecargar una función global, se puede también sobrecargar el constructor de la clase o cualquiera otra función miembro de una clase excepto el destructor (posteriormente se describirá el concepto de destructor, pero no se puede sobrecargar). De hecho, los constructores sobrecargados son bastante frecuentes; proporcionan medios alternativos para inicializar objetos nuevos de una clase.

Solo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores hayan sido definidos.

Clase `Punto` con dos constructores sobrecargados.

```
class Punto {
public:
 Punto(); // constructor
 Punto(int valx, int valy); // constructor sobrecargado
 // ...
private:
 int x;
 int y;
};
```

### Ejemplo 17.15

La declaración de un objeto `Punto` puede llamar a cualquier constructor

```
Punto P; // llamada al constructor por defecto
Punto Q(25,50); // llamada al constructor alternativo
```

## Constructor de copia

Existe un tipo especializado de constructor denominado *constructor de copia*, que crea automáticamente el compilador. El constructor de copia se llama automáticamente cuando un objeto se pasa por valor: se construye una copia local del objeto que se construye. El constructor de copia se llama también cuando un objeto se declara e inicializa con otro objeto del mismo tipo.



**Ejemplo 17.16** Se crean objetos `Punto` llamando al constructor de copia.

```
Punto P;
Punto T(P);
Punto Q = P;
```

Por defecto, C++ construye una copia bit a bit de un objeto. Sin embargo, se puede también implementar el constructor de la copia, el cual se utiliza para notificar al usuario de que una copia se ha realizado, normalmente como una ayuda de depuración.

### Regla

Si existe un constructor alternativo, C++ no generará un constructor por defecto. Para prevenir a los usuarios de la clase un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

### Ejemplo

```
class Punto {
public:
 Punto (int valx, int valy);
private:
 Punto ();
 / / ...
};

Punto T; / / error; el constructor no está accesible
```

## Inicialización de miembros en constructores

No está permitido inicializar un miembro dato de una clase cuando se define. Por consiguiente, la siguiente definición de la clase genera errores:

```
class C {
private:
 int T = 0; // Error
 const int CInt = 25; // Error
 int &Dint = T; // Error
// ...
};
```

No tiene sentido inicializar un miembro dato dentro de una definición de la clase, dado que la definición de la clase indica simplemente el *tipo* de cada miembro dato y no reserva realmente memoria. En su lugar, se desea inicializar los miembros dato cada vez que se crea una *instancia específica* de la clase. El sitio lógico para inicializar miembros dato, por consiguiente, está dentro del constructor de la clase. El constructor de la clase inicializa los miembros dato utilizando *expresiones de asignación*. Sin embargo, ciertos tipos de datos, específicamente, constantes y referencias, no pueden ser valores asignados. Para resolver este problema, C++ proporciona una característica de constructor especial conocido como *lista inicializadora de miembros* que permite inicializar (en lugar de asignar) a uno o más miembros dato.

Una lista inicializadora de miembros se sitúa inmediatamente después de la lista de parámetros en la definición del constructor; consta de un carácter dos puntos, seguido por uno o más *inicializadores de miembro*, separados por comas. Un inicializador de miembros consta del nombre de un miembro dato seguido por un valor inicial entre paréntesis.

Constructor de la clase C con lista de inicialización.

### Ejemplo 17.17

```
class C {
private:
 int T;
 const int CInt;
 int &Dint;
 // ...
public:
 C (int Param): T(Param), CInt(25), Dint (T)
 {
 // código del constructor
 }
 // ...
};
```

La definición siguiente crea un objeto

```
C CObjeto (0);
```

con los miembros dato T y CInt inicializadas a 0 y 25, y el miembro dato Dint se inicializa de modo que referencia a T.

### Ejercicio 17.4

Diseñar y construir una clase contador que cuente cosas (cada vez que se produzca un suceso el contador se incrementa en 1). El contador puede ser consultado para encontrar la cuenta actual.

El programa define dos objetos tipo Contador incrementa el atributo de cuenta de cada objeto y lo visualiza

```
#include <iostream>
using namespace std;

class Contador {
private:
 unsigned int cuenta; // contar
public:
 Contador() {cuenta = 0;} // constructor
 void inc_cuenta(){cuenta++;} // cuenta
 int leer_cuenta(){return cuenta;} // devuelve cuenta
};
```

```

void main()
{
 Contador c1, c2// define e inicializa
 cout << "\nc1 = " << c1.leer_cuenta();
 cout << "\nc2 = " << c2.leer_cuenta();
 c1.inc_cuenta(); // incrementa c1
 c2.inc_cuenta(); // incrementa c2
 c2.inc_cuenta(); // incrementa c2
 cout << "\nc1 = " << c1.leer_cuenta();// visualiza de nuevo
 cout << "\nc2 = " << c2.leer_cuenta();
}

```

La clase `Contador` tiene un elemento dato: `cuenta`, del tipo `unsigned int` (la cuenta siempre es positivo). Tiene tres funciones miembro: `Contador( )`; `inc_cuenta( )`, que añade 1 a `cuenta`; y `leer_cuenta( )`, que devuelve el valor actual de `cuenta`.

## 17.4 Destructores

En una clase se puede definir también una función miembro especial conocida como *destructor*, que se llama automáticamente siempre que se destruye un objeto de la clase. El nombre del destructor es el mismo que el nombre de la clase, precedida con el carácter `~`. Al igual que un constructor, un destructor se debe definir sin ningún tipo de retorno (ni incluso `void`); al contrario que un constructor, no puede aceptar parámetros. Solo puede existir un *destructor*.



### Ejemplo 17.18

```

class Demo
{
private:
 int datos;
public:
 Demo() {datos = 0;} // constructor
 ~Demo() { } // destructor
};

```

#### Regla

- Los destructores no tienen valor de retorno.
- Tampoco tienen argumentos.

El uso más frecuente de un destructor es liberar memoria que fue asignada por el constructor. Si un destructor no se declara explícitamente, C++ crea uno vacío de manera automática.

Si un objeto tiene ámbito local, su destructor se llama cuando el control pasa fuera de su bloque de definición. Si un objeto tiene ámbito de archivo, el destructor se llama cuando termina el programa principal (`main`). Si un objeto se asignó dinámicamente (utilizando `new` y `delete`), el destructor se llama cuando se invoca el operador `delete`.



### Ejemplo 17.19

El destructor notifica cuándo se ha destruido un `Punto`

```

class Punto {
public:

```

```
~Punto()
{
 cout << "Se ha llamado al destructor de Punto \n";
}
// ...
i
```

## Clases compuestas

Una clase *compuesta* es aquella que contiene miembros dato que son asimismo objetos de clases. Antes de activar el constructor de la clase compuesta, se construyen los miembros objetos individualizados en su orden de declaración.

La clase `Estudiante` contiene miembros dato de tipo `Expediente` y `Dirección`:

```
class Expediente { // ...};
class Direccion { // ...};
class Estudiante {
public :
 Estudiante()
 {
 PonerId(0);
 PonerNotaMedia(0.0);
 }
 void PonerId(long);
 void PonerNotaMedia(float);
private:
 long id;
 Expediente exp;
 Direccion dir;
 float NotMedia;
};
```

Aunque `Estudiante` contiene `Expediente` y `Dirección`, el constructor de `Estudiante` no tiene acceso a los miembros privados o protegidos de `Expediente` o `Dirección`. Cuando un objeto `Estudiante` sale de alcance, se llama a su destructor. El cuerpo de `~Estudiante` se ejecuta antes que los destructores de `Expediente` y `Dirección`. En otras palabras, el orden de las llamadas a destructores a clases compuestas es exactamente el opuesto al orden de llamadas de constructores.

## 17.5 Sobrecarga de funciones miembro

Al igual que sucede con las funciones no miembro de una clase, las funciones miembro de una clase se pueden sobrecargar. A excepción de los operadores sobrecargados, que tienen reglas especiales, una función miembro se puede sobrecargar pero solo en su propia clase. Una función miembro de una clase tiene como ámbito la clase, está relacionada y por ello no se puede sobrecargar, normalmente, con funciones no miembro o funciones declaradas en otras clases.

Las mismas reglas utilizadas para sobrecargar funciones ordinarias se aplican a las funciones miembro: dos miembros sobrecargados no puede tener el mismo número y tipo de parámetros. La sobrecarga permite utilizar un mismo nombre para una función y ejecutar la función definida más adecuada a los parámetros pasados durante la ejecución del programa. La ventaja fundamental de trabajar con funciones miembro sobrecargadas es la comodidad que aporta a la programación.

Para ilustrar la sobrecarga, veamos la clase `Producto` donde aparecen diferentes funciones miembro con el mismo nombre y distintos tipos de parámetros

```
class Producto
```

```

public:
 int producto (int m, int n); //funcion 1
 int producto (int m, int p, int q); //funcion 2
 int producto (float m, float n); //funcion 3
 int producto (float m, float n, float p); //funcion 4
}

```

### Requisitos para la sobrecarga

No pueden existir dos funciones en el mismo ámbito con igual signatura (lista de parámetros).

Los constructores suelen ser las funciones que se sobrecargan con mayor frecuencia, como ya se ha visto en ejemplos anteriores.

## 17.6 Funciones amigas

En ocasiones una función global (no miembro de una clase) necesita poder acceder a los miembros privados de una clase; al ser función externa de una clase no puede referirse a los miembros dato privados de la clase. En estos casos, es posible declarar la función *amiga* (*friend*) de la clase. La declaración de una función como *amiga* garantiza que dicha función accede a sus miembros no públicos.

Una declaración amiga comienza con la palabra reservada *friend* y continúa con el prototipo de la función. La declaración de la función *amiga* debe aparecer dentro de la definición de la clase. Aunque las funciones amigas se declaran dentro de una clase, no son funciones miembro, por consiguiente no están afectadas por la sección *pública*, *privada* o *protegida* en que están declaradas dentro del cuerpo de la clase.



### Ejemplo 17.20

La clase *Cadena* declara *friend* de la clase a dos funciones globales.

```

class Cadena
{
 friend bool igual(const Cadena&, const Cadena&);
 friend int numVocales(const Cadena&);

public:
 // resto de la clase Cadena
};

```



### Ejercicio 17.5

La clase *Punto3D* declara *friend* de la clase a la función global *suma2P* que suma dos objetos tipo *Punto3D*.

```

#include <iostream>
using namespace std;
class Punto3D
{
public:
 Punto3D(int xx = 0, int yy = 0, int zz = 0):x(xx),y(yy),z(zz){;}
 void imprimirPunto() const;
 friend Punto3D suma2P (const Punto3D &a, const Punto3D &b);
private:
 int x, y, z;
};

```

```
void Punto3D::imprimirPunto () const
{
 cout << x << ',' << y << ',' << z << endl;
}
```

Definición de la función `friend` (*accede a miembros privados*). La función tiene dos argumentos que son objetos `Punto3D`; se suman los miembros privados de ambos y se crea otro objeto.

```
Punto3D suma2P (const Punto3D &a, const Punto3D &b);
{
 Punto3D sm(a.x + b.x, a.y + b.y, a.z + b.z);
 return sm;
}
```

El siguiente programa crea objetos `Punto3D`, llama a la función `suma2P` y muestra el resultado de la suma.

```
int main()
{
 Punto3D u(3,1,-6), v(1,2,4), s;
 s = suma2P(u,v);
 s.imprimirPunto();
 return 0;
}
```

Gracias al prefijo `friend` de la declaración de la función `suma2P`, esta función puede acceder directamente a los miembros `x`, `y`, `z`. Aunque una función amiga no es miembro de la clase, como ya se ha comentado, se puede definir dentro de la clase.

### Nota

En una clase `A`, se puede especificar que todas las funciones miembros de una clase `B` son funciones amigas, declarando la clase `B` como amiga de `A`. Se escribe así:

```
friend class B
```

La clase `Cartilla` define miembros dato privados; se requiere que todas las funciones miembro de la clase `Empleado` tengan acceso a los miembros privados de `Cartilla`, para ello se declara a `Empleado` `friend` de la clase `Cartilla`.

```
class Empleado
{
private:
 string nombre;
 ...
public:
 void darAlta(Cartilla &);
 void anotar(Cartilla&);
 ...
};

class Cartilla
{
 friend class Empleado;
private:
```

### Ejemplo 17.21



```

 // miembros privados de Cartilla
public:
 // miembros de Cartilla
};

```

## 17.7 Sobrecarga de operadores

En C++ hay un número determinado de operadores predefinidos que se pueden aplicar a tipos estándar incorporados o integrados. Por ejemplo, el operador `+` sirve para sumar dos variables de tipo `int`, el operador `<` puede comparar dos valores de tipo `double`. Estos operadores predefinidos, realmente, están sobrecargados ya que cada uno de ellos se puede utilizar para diferentes tipos de datos. Por ejemplo, un operador `+` se utiliza para tipo `int` y otro para tipo `double`. Sin embargo, cuando los tipos de datos son clases no hay prácticamente operadores predefinidos.

Al trabajar con clases; por ejemplo, la clase `Racional` que representa a los números racionales y sus posibles operaciones, es muy importante que se puedan realizar operaciones como suma, resta, producto, etc., de igual forma que se realizan las operaciones de números enteros, reales, etc. Es decir, si `r1` y `r2` son números de tipo `racional` o de tipo `complejo`, se trata de definir que el operador `+` pueda servir para escribir `r1+r2` y que el compilador entienda correctamente la operación. En esencia lo que se busca es que los datos de tipo clases (`String`, `Fecha`, `Complejo`, etc.), puedan ser tratados como si fueran tipos de datos simples.

Casi todos los operadores predefinidos para tipos estándar se pueden sobrecargar. La tabla 17.3 muestra dichos operadores.

Tabla 17.3 Operadores que se pueden sobrecargar.

|                    |                     |                         |                       |                       |                        |                        |                     |                    |  |
|--------------------|---------------------|-------------------------|-----------------------|-----------------------|------------------------|------------------------|---------------------|--------------------|--|
| <code>%</code>     | <code>^</code>      | <code>&amp;</code>      | <code>~</code>        | <code> </code>        |                        |                        |                     |                    |  |
| <code>+</code>     | <code>-</code>      | <code>*</code>          | <code>/</code>        |                       |                        |                        |                     |                    |  |
| <code>!</code>     | <code>=</code>      | <code>&lt;</code>       | <code>&gt;</code>     | <code>+=</code>       | <code>-=</code>        | <code>*=</code>        | <code>/=</code>     | <code>%=</code>    |  |
| <code>^=</code>    | <code>&amp;=</code> | <code> =</code>         | <code>&lt;&lt;</code> | <code>&gt;&gt;</code> | <code>&gt;&gt;=</code> | <code>&lt;&lt;=</code> | <code>==</code>     | <code>!=</code>    |  |
| <code>&lt;=</code> | <code>&gt;=</code>  | <code>&amp;&amp;</code> | <code>  </code>       | <code>++</code>       | <code>--</code>        | <code>,</code>         | <code>-&gt;*</code> | <code>-&gt;</code> |  |
| <code>()</code>    | <code>[]</code>     | <code>new</code>        | <code>delete</code>   |                       |                        |                        |                     |                    |  |

Una función operador es aquella cuyo nombre consta de la palabra reservada `operator` seguida por un operador unitario o binario.

### Formato

```
operator operador
```

Los siguientes operadores no se pueden sobrecargar:

- `.` acceso a miembro
- `.*` indirección de acceso a miembro
- `::` resolución de ámbitos
- `?:` if aritmético

Excepto para el caso de los operadores `new`, `delete` y `->`, las funciones operador pueden devolver cualquier tipo de dato. La precedencia, agrupamiento y número de operandos no se puede cambiar. Con excepción del operador `()`, las funciones operador no pueden tener argumentos por defecto.

Una función operador debe ser o bien una función miembro no estática o una función no miembro que tenga al menos un parámetro cuyo tipo es una clase, referencia a una clase o una clase enumeración, o referencia a una enumeración. Esta regla evita cambiar el significado de operadores que operan sobre tipos de datos intrínsecos. Por ejemplo, el siguiente código no será válido:

```
int operator +(int x, int y)
{
 return x * y;
}
```

Por contra, la siguiente función `operator` no miembro es válida ya que al menos un parámetro es un tipo clase:

```
class Integer
{
 // ...
public:
 int valor;
};

int operator + (const Integer &x, int y)
{
 return x.valor + y;
}
```

Cuando se declaran operadores para clases lo que se hace es escribir una función con el nombre `operator x` en donde `x` es el símbolo de un operador. Así, si se escribe `operator +`, la función que se declara tendrá el nombre `operator +`. Existen dos medios de construir nuestros propios operadores, bien formulándolos como funciones amigas de la clase, aunque lo usual será como funciones miembro.

### Reglas

- Las funciones con el nombre `operator x`, en donde `x` es el símbolo del operador.
- La mayoría de los operadores predefinidos pueden ser sobrecargados para tipos clase.
- Puede tener un operando (unitario) o dos operandos (binario).
- El número de operandos y la prioridad son la misma que la del correspondiente operador predefinido.
- Se puede construir como una función miembro o como una función amiga.

Sobrecarga de los operadores binario `+` y `-` junto con el operador unitario `-`

### Ejemplo 17.22

```
#include <iostream>
using namespace std;
class vector {
public:
 vector(float xx = 0, float yy = 0):x(xx), y(yy) {}
 void imprimirvec() const;
 vector operator+(const vector &b) const; // + binario
 vector operator-(); // - unitario
 vector operator-(const vector &b) const; // - binario
private:
 float x, y;
};

void vector::imprimirvec() const
{
 cout << x << ',' << y << endl;
}
```

La definición de las función `operator +`:

```
vector vector::operator + (const vector &b) const // + binario
```

```

{
 return vector(x + b.x, y + b.y);
}

```

La definición de la función `operator -` como operador unario:

```

vector vector::operator - () // - unitario
{
 return vector(-x, -y);
}

```

La definición de la función `operator -` como operador binario:

```

vector operator - (const vector &b) const
{
 // utiliza la sobrecarga del operador + y del operador unario - u
 return *this + (-b);
}

```

El siguiente programa crea objetos tipo `vector` y realiza operaciones

```

int main()
{
 vector u(3,1), v(1,2), suma, neg, differ;
 suma = u + v;
 suma.imprimirvec(); // salida 4,3
 neg = -suma;
 neg.imprimirvec(); // salida -4, -3
 differ = u - v;
 differ.imprimirvec(); // salida 2, -1
 return 0;
}

```

## 17.8 Sobrecarga de operadores unitarios

Un operador unitario se puede definir bien *como un amigo* con un argumento, o bien como una función miembro sin argumento, que es el caso más frecuente.

En el siguiente ejemplo se sobrecarga el operador unitario `++` para incrementar objetos de la clase `Tiempo`.

```

#include <iostream>
using namespace std;
class Tiempo
{
private:
 int hrs; // horas
 int mins; // minutos
 int segs; // segundos
public:
 Tiempo()
 {
 segs = 0;
 mins = 0;
 hrs = 0;
 } // inicializa tiempo a 00:00:00
 void verHora(void) // visualizar tiempo
 {
 cout << hrs << ":" << mins << ":" << segs;
 }
}

```

```

void leerTiempo (void) // leer hora del teclado
{
 cin >> hrs >> mins >> segs;
}
Tiempo & void operator ++() // sumar 1 segundo a esta hora
{
 segs++;
 if (segs > 59)
 {
 segs -= 60;
 ++mins;
 }
 if (mins > 59)
 {
 mins -= 60;
 ++hrs;
 }
 return * this;
}

```

El siguiente programa crea un objeto tipo Tiempo, se introducen datos para el objeto y se aplica el operador sobrecargado ++.

```

int main()
{
 Tiempo t1; // declara variable tiempo
 cout << "\n Introduzca hora (hh:mm:ss) : ";
 t1.leerTiempo(); // leer t1
 cout << "visualizar t1";
 t1.verHora();
 ++t1; // incrementar t1,
 cout << "\n después de incrementar, t1= "; // ver de nuevo t1
 t1.verHora();
 return 0;
}

```

Se ha utilizado el operador sobrecargado ++ para aumentar el tiempo en 1 segundo.

Tiempo & operator++

Se puede entonces incrementar t1, un objeto de la clase tiempo, con la sentencia ++t1;

Un ejemplo de salida de este programa:

```

Introduzca hora (hh:mm:ss) :10:59:59
Después de incrementar, t1 = 11:00:00

```

## Sobrecargar un operador unitario como función miembro

El siguiente ejemplo muestra cómo se puede sobrecargar un operador unitario como una función miembro de una clase:

```

#include <iostream>
using namespace std;

class Unitaria
{
 int x;
public:

```

```

Unitaria() {x = 0;}
Unitaria(int a) {x = a;}
Unitaria& operator--()
{
 --x;
 return *this;
}
void visualizar() {cout << x << "\n";}
};

```

El siguiente programa crea un objeto de tipo `Unitaria` al que se le aplica el operador sobrecargado `--`.

```

int main()
{
 Unitaria ejemplo = 65;
 for(int i = 5; i > 0; i--)
 {
 --ejemplo; // equivale a ejemplo.operator--()
 ejemplo.visualizar();
 }
 return 0;
}

```

La función operador unitario `--()` está declarada sin argumentos, al ser una función miembro el sistema pasa el puntero `this` implícitamente. Por consiguiente, el objeto que se liga a la función miembro se convierte en el operando de este operador.

## Sobrecarga de un operador unitario como una función amiga

El siguiente ejemplo sobrecarga el operador unitario `++` como una función *amiga* de la clase:

```

#include <iostream>
using namespace std;
class Unitaria
{
 int x;
public:
 Unitaria() {x = 0;}
 Unitaria(int a) {x = a;}
 friend Unitaria& operator++(Unitaria& y);
 void visualizar() {cout << x << "\n";}
};

```

La definición de la función `operator++`:

```

Unitaria& operator++(Unitaria& y)
{
 y.x = y.x + 1;
 return y;
}

```

El siguiente programa crea un objeto de tipo `Unitaria` al que se le aplica el operador sobrecargado `++`.

```

int main()
{
 Unitaria ejemplo = 65;
 for(int i = 5; i > 0; i--)
 {
 ++ejemplo;
 }
}

```

```

 ejemplo.visualizar();
 }
 return 0;
}

```

La función operador `++ ()` está definida fuera de la clase, y debido a que es una función *amiga*, el sistema no pasa el puntero `this` implícitamente. Por consiguiente, tiene un argumento que es el objeto de la clase `Unitaria` sobre el que se aplica el operador.

Los operadores de incremento y decremento `++` y `--` tienen características especiales ya que presentan dos modalidades, una modalidad prefija (`++` o `--` se sitúan delante del nombre de la variable) y una modalidad postfija (`++` o `--` se sitúan después del nombre de la variable). Ambas modalidades incrementan o decrementan su operando pero la diferencia entre ellas es que la modalidad prefija proporciona el nuevo valor del operando como resultado, mientras que la modalidad postfija proporciona como resultado el valor más antiguo del operando.

## 17.9 Sobrecarga de operadores binarios

Los operadores binarios pueden ser sobrecargados tan fácilmente como los operadores unitarios. Se pueden sobrecargar pasando a la función dos argumentos si no es función miembro. El primer argumento es el operando izquierdo del operador sobrecargado y el segundo argumento es el operando derecho si la función no es miembro de la clase. Consideremos un tipo de clase `T` y dos objetos `x1` y `x2` de tipo `T`.

Definamos un operador binario `+` sobrecargado. Entonces:

```

x1 + x2
se interpreta como
operator+(x1, x2)
o como
x1.operator+(x2)

```

Un operador binario puede, por consiguiente, ser definido:

- bien como una función de dos argumentos (función *amiga* de la clase);
- bien como una función miembro de un argumento, y es el caso más frecuente;
- pero nunca las dos a la vez.

### Sobrecarga de un operador binario como función miembro

El siguiente ejemplo muestra cómo sobrecargar un operador binario como una función miembro de la clase:

```

#include <iostream>
using namespace std;
class Binario
{
 int x;
public:
 Binario() {x = 0;}
 Binario(int a) {x = a;}
 Binario operator + (const Binario&) const;
 void visualizar() {cout << x << "\n";}
};

```

La definición de la función miembro `operator +`:

```

Binario Binario::operator+(const Binario& a) const
{
 Binario aux;
 aux.x = x + a.x;
 return aux;
}

```

A continuación un programa crea tres objetos de tipo `Binario` y se realiza la operación suma (+) con dos objetos.

```
int main()
{
 Binario primero(2), segundo(5), tercero;
 tercero = primero + segundo;
 tercero.visualizar();
 return 0;
}
```

La salida del programa es

7

### Sobrecarga de un operador binario como una función amiga

Ahora la función operador + se define como una función amiga de la clase `Binario`:

```
#include <iostream>
using namespace std;
class Binario
{
 int x;
public:
 Binario() { x = 0; }
 Binario(int a) { x = a; }
 friend Binario operator+(const Binario& a, const Binario& b)
 void visualizar () { cout << x << "\n"; }
};
```

La definición de la función operator + (*amiga* de la clase):

```
Binario operator+(const Binario& a, const Binario& b)
{
 Binario aux;
 aux.x = a.x + b.x;
 return aux;
}
```

A continuación el programa crea tres objetos de tipo `Binario` y se realiza la operación suma (+) con dos objetos.

```
int main()
{
 Binario primero(8), segundo(4), tercero;
 tercero = primero + segundo;
 tercero.visualizar();
 return 0;
}
```

La salida del programa es

12

La función operador binario `operator+` está declarada amiga de la clase; debido a que es una función amiga el sistema no pasa el puntero `this` implícitamente, y por consiguiente se deben pasar los dos objetos `Binario` explícitamente con ambos argumentos. Como consecuencia, el primer argumento de la función miembro se convierte en el operando izquierdo de este operador y el segundo argumento se convierte en operando derecho.

## 17.10 Conversión de datos y operadores de conversión de tipos

Cuando en C++ una sentencia de asignación asigna un valor de un tipo estándar a una variable de otro tipo estándar, C++ convierte automáticamente el valor al mismo tipo que la variable receptora, haciendo que los dos tipos sean compatibles. Por ejemplo, las siguientes sentencias generan conversión de tipos numéricos:

```
long cuenta = 8; // el valor 8 se convierte a un tipo long
double hora = 15; // el valor 15 se convierte en un tipo double
int lado = 4.44; // el valor double 4.44 se convierte en un int, 4
```

### Conversión entre tipos básicos

Cuando se escribe una sentencia como

```
varentera = varfloat;
```

donde `varentera` es una variedad de tipo `int` y `varfloat` es una variable de tipo `float`, se supone que el compilador llama una rutina especial que convierte el valor de `varfloat`, en coma flotante, a un formato entero que se puede asignar a `varentera`. Existen muchas conversiones: `char` a `float`, `float` a `double`, etc. Cada conversión tiene su propia rutina, a la que se llama cuando los tipos de datos son diferentes a ambos lados. A esta conversión se la denomina *implícita*.

Con frecuencia, se necesita forzar al compilador para convertir un tipo a otro. Para realizar esta operación se utiliza el operador de *molde (cast)* o conversión forzada de tipos. Por ejemplo, para convertir `float` a `int` puede expresarse

```
varentera = int(varfloat);
```

### Conversión entre objetos y tipos básicos

En la conversión entre tipos definidos por el usuario y tipos básicos (incorporados) no se pueden utilizar las rutinas de conversión implícitas. En su lugar se deben escribir funciones de conversión. Así, para convertir un tipo estándar o básico, por ejemplo, `float`, a un tipo definido por el usuario, por ejemplo `Distancia`, se utilizará un *constructor con un argumento*. Por el contrario, para convertir un tipo definido por el usuario a un tipo básico se ha de emplear un *operador de conversión*.

Y se deben tener presentes los siguientes puntos:

- La función de conversión debe ser un método de la clase a convertir.
- La función de conversión no debe especificar un tipo de retorno.
- La función de conversión no debe tener argumentos.
- El nombre de la función debe ser el del tipo al que se quiere convertir el objeto.

Por ejemplo, una función para convertir a un tipo `double` tendrá este prototipo:

```
operator double();
```

Veamos una función de conversión diseñada para convertir clases diferentes: `Rect`, que representa la posición de un punto en el sistema de coordenadas rectangulares, y `Polar`, que representa la posición del mismo punto en el sistema de coordenadas polares.

Recordemos que las coordenadas de un punto en el plano se pueden representar por las coordenadas rectangulares `x` y `y`, o bien por las coordenadas polares `Radio` y `Angulo`.

```
class Rect
{
 private:
 double xco; // coordenada x
 double yco; // coordenada y
 public:
 Rect() {xco = 0.0; yco = 0.0;}// constructor sin argumentos
 Rect(double x, double y) // constructor con dos argumentos
 {xco = x; yco = y;}
 void visualizar()
 {
 cout << " (" << xco << " , " << yco << ") " ;}
}
```

```

};

class Polar
{
 private:
 double radio;
 double angulo;
 public:
 Polar() // constructor sin argumentos
 {radio = 0.0; ángulo = 0.0}
 Polar(double r, double a) // constructor dos argumentos
 {radio = r; ángulo = a ;}
 void visualizar()
 {
 cout << " (" << radio << " , " << ángulo << ") ";
 }
 operator Rect() // función de conversión
 {
 double x = radio * cos(ángulo); // calcular x
 double y = radio * sin(ángulo); // calcular y
 return Rect(x, y); // invoca al constructor de la clase Rect
 }
};

```

En el siguiente programa se crean objetos tipo Rect y Polar.

```

int main()
{
 Rect rec; // Rect utiliza constructor sin argumentos
 Polar pol(100.0,0.785398); // Polar utiliza segundo constructor
 rec = pol; // convierte Polar a Rect
 // utilizando función de conversión
 // o bien rec = Rect(pol);
 cout << "\n polar= ";
 pol.visualizar(); // polar original
 cout << "\n rectang= ";
 rec.visualizar(); // rectangular equivalente
 return 0;
}

```

La función `operator Rect ()` transforma el objeto del cual es miembro en un objeto `Rect` y devuelve este objeto, que `main()` asigna a `Rec`.

La salida del programa para unos datos iniciales de 100,0 de radio y un ángulo de 45° (0,785398 radianes).

```

polar = (100,00785398)
rectang = (70.710690,70.7106767)

```

## 17.11 Errores de programación frecuentes

Las facilidades en el uso de la estructura `clase` aumenta las posibilidades de errores.

1. *Mal uso de palabras reservadas.* Es un error declarar una clase sin utilizar fielmente una de las palabras reservadas `class`, `struct` o `union` en la declaración.
2. La palabra reservada `class` no se necesita para crear objetos de clases. Por ejemplo, se puede escribir

```

class C {
 ...
};

```

```
C c1, c2 // definiciones de objetos
en lugar de
class C c1, c2 // no se necesita class
```

3. Si un miembro de una clase es privado (`private`), se puede acceder solo por funciones amigas de la clase. Por ejemplo, este código no es correcto.

```
class C {
 int x;
 ...
public :
 C() {x = -9999;}
};

void f()
{
 C c;
 cout << c.x // ERROR
}
```

ya que `x` es privado en la clase `C` y la función `f` no es ni un método ni una amiga de `C`.

4. Uso de constructores y destructores.

- No se puede especificar un tipo de retorno en un constructor o destructor, ni en su declaración ni en su definición. Si `C` es una clase, entonces no será legal:

```
void C::C(int n) // ERROR
{
 ...
}
```

- De igual modo, no se puede especificar ningún argumento en la definición o declaración de un constructor por omisión de la clase:

```
C::C(void) // ERROR
{
 ...
}
```

- Tampoco se pueden especificar argumentos, incluso `void`, en la declaración o definición del destructor de una clase.
- No se pueden tener dos constructores de la misma clase con los mismos tipos de argumentos.
- Un constructor de una clase `C` no puede tener un argumento de un tipo `C`, pero sí puede tener un argumento de tipo referencia, `C&`. Este constructor es el constructor de copia.

5. Inicializadores de miembros.

```
class C {
 // ERROR
 int x = 5;
};

class C {
 int x;
 ...
public:
 C() {x = 5;} // CORRECTO
};
```

6. No se puede acceder a un miembro dato protegido (`protected`) fuera de su jerarquía de clases, excepto a través de una función amiga. Por ejemplo,

```
class C { // clase base
protected:
```

```

 int x;
 };
 class D:public C { // clase derivada
 ...
};

int main()
{
 C c1;
 c1.x = 10 // ERROR, x es protegido
 ...
}

```

contiene un error, ya que `x` solo es accesible por métodos de `C` y métodos y amigas de las clases derivadas de `C` (véase capítulo 18), como `D`.

7. No se pueden sobrecargar ninguno de estos operadores:

`.`    `.*`    `::`    `?:`    `sizeof`

8. No se puede utilizar `this` como parámetro, ya que es una palabra reservada. Tampoco se puede utilizar en una sentencia de asignación como

`this = ...; // this es constante; ERROR`

ya que `this` es una constante.

9. Un constructor de una clase `C` no puede esperar un argumento de tipo `C`:

```

class C {
 ...
};

C::C(C c) // ERROR
{
 ...
}

C::C(C& c) // CORRECTO
{
 ...
}

```

10. Un miembro dato `static` no se puede definir dentro de una clase, aunque sí puede ser declarado dentro de la declaración de la clase.

```

class C {
 static int x = 7; // ERROR
 ...
};

class D {
 static int x;
};

// definición con inicialización
int D::x = 7;

```

11. Los constructores o destructores no pueden declarar `static`.

12. *Olvido de puntos y coma en definición de clases.* Las llaves `{ }` son frecuentes en código C++, y, normalmente, no se sitúa un punto y coma después de la llave de cierre. Sin embargo, la definición de `class` siempre termina en `};`. Un error típico es olvidar ese punto y coma.

```

class Producto
{
 public:
 // ...
private:

```

```
//...
} // olvido del punto y coma
```

13. *Olvido de inicialización de todos los campos de un constructor.* Todo constructor necesita asegurar que todos los campos (miembros) de datos se fijan a los valores apropiados.

```
class Empleado {
public:
 Empleado();
 Empleado(string n); // se olvida el parámetro salario
 //...
private:
 string nombre;
 float salario;
};
```

14. *Referencia a un atributo privado de una clase.* Los identificadores declarados como atributos o funciones privados de una clase no pueden ser referenciados desde el exterior de la clase. Cualquier intento de referencia producirá mensaje de error similar a

"undefined symbol"

15. *Fallo por no incluir un archivo de cabecera requerido.* Se generan numerosos mensajes de error por este fallo de programación ya que un archivo de cabecera contiene la definición de un número de los identificadores utilizados en su programa y el mensaje más frecuente de estos errores será:

"undefined symbol"

16. *Fallo al definir una función como miembro de una clase.* Este error se puede producir de varias formas:

1. Fallo al prefijar la definición de la función por el nombre de su clase y el operador de resolución de ámbito (::).
2. Fallo al escribir el nombre de la función correctamente.
3. Omisión completa de la definición de la función de la clase.

En cualquiera de los casos, el resultado es el mismo: un mensaje de error del compilador que indica que la función llamada no existe en la clase indicada.

"is not a member"

Si se invoca a una función `imprimir` del objeto `c1` de la clase `contador` en la sentencia de C++:

`c1.imprimir( );`  
aparece en el programa `cliente`; se visualizará el mensaje

" 'imprimir' if not a member of 'contador'"

### Ejemplo 17.23

17. *Olvido de puntos y coma en prototipos y cabeceras de funciones.* La omisión de un punto y coma al final del prototipo de una función puede producir el mensaje de error

"Statement missing"

o bien

"Declaration terminated incorrectly".



## Resumen

- Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real.
- Un objeto de una clase tiene dos componentes: un conjunto de atributos y un conjunto de comportamientos (operaciones). Los atributos se llaman miembros dato y los comportamientos se llaman funciones miembro.

```
class circulo {
 double x_centro, y_centro;
 double superficie(void);
};
```

- Cuando se crea un nuevo tipo de clase, se deben realizar dos etapas fundamentales: determinar los atributos y el comportamiento de los objetos.
- Un objeto es una instancia de una clase.

```
circulo un_circulo, tipo_circulo[100];
```

- Una declaración de una clase se divide en tres secciones: *pública*, *privada* y *protegida*. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Se recomienda la declaración de los constructores en la sección pública. La sección privada contiene las funciones miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estas funciones miembro y atributos dato son accesibles solo por la función miembro del objeto.
- El acceso a los miembros de una clase se puede declarar como *privado* (*private*, por defecto), *público* (*public*) o *protegido* (*protected*).

```
class circulo {
public:
 double superficie(void);
 void fijar_centro(double x, double y);
 void leer_radio(double radio)
 { r = radio; }
 double dar_radio(void);
private:
 double centro_x, centro_y;
 double r;
};
```

- Los miembros dato *centro\_x*, *centro\_y* y *r* son ejemplos de ocultación de datos.
- Una función definida dentro de una clase, como *leer\_radio()*, se declara implícitamente en línea (*inline*). El operador de resolución de ámbito *::* se utiliza para definir una función miembro externa a la definición de la clase.

```
double circulo::dar_radio(void)
{ return r; }
```

Las funciones definidas así no son implícitamente *inline*.

- Existen dos métodos fundamentales de especificar un objeto

```
circulo c; // un objeto
circulo *pt = new circulo(); // un puntero
a un objeto
```

Se pueden utilizar dos operadores diferentes para acceder a miembros de la clase:

1. El operador de acceso a miembro (el operador punto).

```
c.radio = 10.0;
```

2. Un operador puntero a miembro

```
pt_objeto->radio = 10;
```

- Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class complejo {
// ...
complejo (double x, double y);
complejo(const complejo &c);
```

Para una clase, *x*, un constructor con el prototípo, *x::x(const x&)*, se conoce como *constructor de copia*.

- Un **constructor** es una función miembro especial que se invoca cuando se crea un objeto. Se utiliza normalmente para inicializar los atributos de un objeto. Los argumentos por defecto hacen al constructor más flexible y útil.
- El proceso de crear un objeto se llama *instanciación* (creación de instancia).
- Una función miembro constante es una que no puede cambiar los atributos de sus objetos. Si se desea utilizar objetos de clase tipo *const*, se deben definir funciones miembro *const*.

```
inline double complejo::real(void) const
{
 return re;
}
```

- Un **destructor** es una función miembro especial que se llama automáticamente siempre que se destruye un objeto de la clase.

```
~complejo();
```

- La sobrecarga de operadores representa nuevos significados. Por ejemplo, el significado de la expresión *a +*

`b` depende de los tipos de las variables `a` y `b`. La expresión puede significar concatenación de cadenas, suma de números complejos o suma de enteros, dependiendo de que las variables sean de tipo cadena, complejo o entero.

- Todos los operadores C++ se pueden sobrecargar con la excepción de:

`.`   `*`   `::`   `:`

- La palabra `operator` se utiliza también para sobrecargar los operadores integrados C++. Al igual que un nombre de una función `printf ( )` puede tener una variedad de significados que depende de sus argumentos, así un operador, como `+`, puede tener significados adicionales.
- La sobrecarga de operadores utiliza normalmente funciones miembro o funciones amiga ya que ambas tienen privilegios de acceso. Cuando un operador unario se sobrecarga utilizando una función miembro, tiene una lista de argumentos vacíos, ya que el único argumento del operador es el argumento implícito. Cuando un operador binario se sobrecarga utilizando una función miembro, tiene como primer argumento la variable de clase pasada implícitamente y como segundo argumento el parámetro de la lista de argumentos.
- Un operador sobrecargado puede ser una función no miembro (normalmente una amiga, `friend`).

```
class x{
// ...
 friend x operator + (const x &u, const x
 &v);
};
```

o bien una función miembro

```
class complejo{
//...
 complejo &operator += (const complejo
 &n);
};
```

- La sobrecarga de operadores se implementa por una función operador. Las funciones miembro tienen un argumento implícito, que es el puntero `this` oculto. La relación entre un operador y las correspondientes llamadas a funciones se resumen en

| Operador                    | Llamada a función                      |                                         |
|-----------------------------|----------------------------------------|-----------------------------------------|
|                             | Miembro                                | No miembro                              |
| <code>A &lt;op&gt; B</code> | <code>A.operator &lt;op&gt; B</code>   | <code>operator &lt;op&gt; (A, B)</code> |
| <code>&lt;op&gt; A</code>   | <code>A.operator &lt;op&gt; ( )</code> | <code>operator &lt;op&gt; (A)</code>    |
| <code>A &lt;op&gt;</code>   | <code>A.operator &lt;op&gt;</code>     | <code>operator &lt;op&gt; (A)</code>    |

## Ejercicios

17.1 ¿Cuál es el error de la siguiente declaración de clase?

```
union float_bytes {
private :
 char mantisa[3], char exponente ;
public:
 float num;
 char exp(void);
 char mank(void);
};
```

17.2 ¿Qué está mal en la siguiente definición de la clase?

```
#include <string.h>
struct buffer {
 char datos[255];
 int cursor ;
 void iniciar(char *s);
 inline int Long(void);
 char *contenido(void);
};
void buffer::iniciar(char *s)
```

```
{
 strcpy(datos,s) ; cursor = 0;
}
char *buffer::contenido(void)
{
 if(Long()) return datos; else
 return 0;
}
inline int buffer:: Long(void)
{return cursor;}
```

17.3 Consideremos una pila como un tipo abstracto de datos. Se trata de definir una clase que implementa una pila de 100 caracteres mediante un arreglo. Las funciones miembro de la clase deben ser:

`meter`, `sacar`, `pilavacia` y `pilallena`.

17.4 Reescribir la clase `pila` que utilice una lista enlazada en lugar de un arreglo. (Sugerencia: utilice otra clase para representar los modos de la lista.)

- 17.5 Examine la siguiente declaración de clase y vea si existen errores.

```
class punto {
 int x, y;
 void punto(int x1, int y1)
 {x = x1; y = y1;}
};
```

- 17.6 Dado el siguiente programa, ¿es legal la sentencia de `main()`?

```
class punto {
public:
 int x, int y;
 punto(int x1, int y1) {x = x1; y = y1;}
};
main()
{
 punto(25, 15); // ¿es legal esta senten-
 cia?
}
```

- 17.7 Suponiendo contestado el ejercicio anterior, ¿cuál será la salida del siguiente programa?

```
#include <stdio.h>
class punto{
public:
 int x, int y;
 punto(int x1, int y1) {x = x1; y = y1}
 punto(int z) { punto (z, z);}
};
main()
{
 punto p(25);
 printf("x = %d, y = %d\n", p.x, p.y);
}
```

- 17.8 Clasificar los siguientes constructores, cuyos prototipos son:

```
rect::rect(int a, int h);
nodo::nodo(void);
calculadora::calculadora(float *vali = 0.0);
cadena::cadena(cadena &c);
abc::abc(float f);
```

- 17.9 Dadas las siguientes declaraciones de clase y *array (arreglo)*, ¿por qué no se puede construir el array?

```
class punto {
public:
 int x, y;
 punto(int a, int b) {x = a; y = b;}
};
punto poligono[5]; // ¿es válida esta
 sentencia?
```

- 17.10 Se cambia el constructor del ejercicio anterior por un constructor por defecto. ¿Se puede construir, en este caso, el arreglo?

```
class punto {
public:
 int x, y;
 punto(int a = 0, int b = 0)
 { x = a; y = b;}
};
punto poligono[5]; // ¿es válida esta
 sentencia?
```

- 17.11 ¿Cuál es el constructor que se llama en las siguientes declaraciones?

```
class prueba_total {
private:
 int num;
public:
 prueba_total(void) {num = 0;}
 prueba_total(int n = 0) {num = n;}
 int valor(void) {return num;}
};
prueba_total prueba;
```

- 17.12 Dado el siguiente programa C++, escribir un programa C equivalente.

```
#include <stdio.h>
class operador {
public:
 float memoria;
 operador(void);
 ~operador(void);
 float sumar(float f);
};
operador::operador(void)
{
 printf("Activar maquina operador
 \n");
 memoria = 0.0
}
operador::~operador(void)
{
 printf("Desactivar maquina operador
 \n");
}
float operador::sumar(float f)
{
 memoria += f;
 return memoria;
}
main()
{
 operador o
 o.sumar(10.0);
```

```

 o.sumar(30.0);
 o.sumar(3.0);
 printf("la solucion es %f\n"
 e.memoria);
}

```

- 17.13 Crear una clase llamada hora que tenga miembros datos separados de tipo int para horas, minutos y segundos. Un constructor inicializará estos datos a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal main( ) crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.
- 17.14 Crear una clase llamada Empleado que contenga como miembro dato el nombre y el número de empleado, y como funciones miembro leerdatos( ) y verdatos( ) que lean los datos del teclado y los visualice en pantalla, respectivamente.
- Escribir un programa que utilice la clase, creando un arreglo de tipo Empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez llenado el arreglo, visualizar los datos de todos los empleados.
- 17.15 Realizar un programa que calcule la distancia media correspondiente a 100 distancias entre ciudades dadas cada una de ellas en kilómetros y metros (utilizar clases).
- 17.16 Se desea realizar una clase vector3d que permita manipular vectores de tres componentes (coordenadas x, y, z) de acuerdo con las siguientes normas:
- Solo posee una función constructor y es en línea.
  - Tiene una función miembro igual que permite saber si dos vectores tienen sus componentes o coordenadas iguales (la declaración de igual se realizará utilizando: a) transmisión por valor; b) transmisión por dirección; c) transmisión por referencia).
- 17.17 Incluir en la clase vector3d del ejercicio anterior una función miembro denominada normamax que permita obtener la norma mayor de dos vectores (Nota: La norma de un vector  $v = x, y, z$  es  $x^2 + y^2 + z^2$ ).
- 17.18 Diseñar una clase vector3d que permita manipular vectores de 3 componentes (de tipo float) y que contenga una función constructor con valores por defecto (0) y las funciones miembros suma (suma de dos vectores), producto escalar (producto escalar de dos vectores:  $v1 = x1, y1, z1$ ;  $v2 = x2, y2, z2$ ;  $v1 \cdot v2 = x1 * x2 + y1 * y2 + z1 * z2$ ).

- 17.19 Dada la siguiente clase punto:

```

class punto
{
 int x, y;
public :
 punto(int abs = 0, int ord = 0)
 { x = abs; y = ord; }
}

```

Escribir una función independiente visualizar, amiga de la clase punto, que permite visualizar las coordenadas de un punto. Se proporcionará por separado un archivo fuente que contenga la nueva declaración (definición) de punto y un archivo fuente que contiene la definición de la función visualizar. Escribir un programa que cree un punto de la clase de forma estática y un punto de clase dinámica y que visualice las coordenadas.

- 17.20 ¿Cuál es la diferencia de significado entre la estructura?

```

struct a {
 int i, j, k;
};

y la clase

class a {
 int i, j, k
};

```

Explique la razón por la que la declaración de la clase no es útil. ¿Cómo se puede utilizar la palabra reservada public para cambiar la declaración de la clase en una declaración equivalente a struct?

- 17.21 Realizar una clase Complejo que permita la gestión de números complejos (un número complejo = dos números reales double: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- Una función establecer( ) permite inicializar un objeto de tipo Complejo a partir de dos componentes double.
- Una función imprimir( ) realiza la visualización formateada de un Complejo.
- Dos funciones agregar( ) (sobrecargadas) permiten añadir, respectivamente, un Complejo a otro y añadir dos componentes double a un Complejo.

- 17.22 Crear una clase lista que realice las siguientes tareas:

- Una lista simple que contenga cero o más elementos de algún tipo específico.
- Crear una lista vacía.
- Añadir elementos a la lista.
- Determinar si la lista está vacía
- Determinar si la lista está llena.
- Acceder a cada elemento de la lista y realizar alguna acción sobre ella.

- 17.23 Escribir una clase `Racional` para números racionales. Un número racional es un número que puede ser representado como el cociente de dos enteros. Por ejemplo,  $1/2$ ,  $3/4$ ,  $4/2$ , etc. Sobrecargar los siguientes operadores de modo que se apliquen al tipo `Racional`: `==`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*` y `/`.
- 17.24 Definir una clase `Complejo` para describir números complejos. Un número complejo es un número cuyo formato es:

`a + b * i`

donde `a` y `b` son números de tipo `double` e `i` es un número que representa la cantidad  $\sqrt{-1}$ . Las variables `a` y `b` se denominan reales e imaginarias. Sobrecargar los siguientes operadores de modo que se apliquen correctamente al tipo `Complejo`: `=`, `+`, `-`, `*`,

`>> y <<`. Para añadir o restar dos números complejos, se suman o restan las dos variables miembro de tipo `double`. El producto de dos números complejos viene dado por la fórmula siguiente:

$$(a+b*i)*(c+d*i) = (a*c-b*d)+(a*d+b*c)*i$$

- 17.25 Escribir una clase `Hora_reloj` que permita expresar la hora del día en horas y minutos utilizando un reloj de 24 horas. La clase debe sobrecargar a los operadores `++` y `---` que incrementen o decremenen las horas y minutos del reloj.
- 17.26 Describir una clase `Punto` que defina la posición de un punto en un plano cartesiano de dos dimensiones. Diseñar e implementar una clase `Punto` que incluya la función operador `*` para encontrar la distancia entre dos puntos.

## Problemas

- 17.1 Implementar una clase `Random` (aleatoria) para generar números pseudoaleatorios.
- 17.2 Implementar una clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, un constructor de copia, funciones de acceso, una función `reiniciar` (`int d, int m, int a`) para reiniciar la fecha de un objeto existente, una función `adelantar` (`int d, int m, int a`) para avanzar una fecha existente (día, d, mes, m, y año a) y una función `imprimir` (). Utilizar una función de utilidad `normalizar` () que asegure que los miembros dato están en el rango correcto  $1 \leq \text{año} \leq 9999$ ,  $1 \leq \text{mes} \leq 12$ ,  $1 \leq \text{día} \leq \text{días}(\text{Mes})$ , donde `días(Mes)` es otra función que devuelve el número de días de cada mes.
- 17.3 Ampliar el programa anterior de modo que pueda aceptar años bisiestos. **Nota:** Un año es bisiesto si es divisible entre 400, o si es divisible entre 4 pero no por 100. Por ejemplo, el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.

- 17.4 Definir una clase `Racional` que represente a números racionales (fracciones). Los miembros privados serán el numerador y el denominador de la fracción, y en la parte pública se debe disponer al menos de las siguientes funciones miembros: `asignar`, `convertir`, `invertir`, `imprimir`, que realizarán las funciones de: asignar los valores de los parámetros a numerador y denominador respectivamente (por ejemplo,  $22/7$ );

convertir a decimal el número racional (por ejemplo,  $3.14286$ ); calcular el inverso de la fracción (por ejemplo,  $7/22$ ) y por último, visualizar la fracción (por ejemplo, se ha de ver  $22/7$ ,  $6/15$ , etc.).

- 17.5 Implementar una clase `Punto` que represente a puntos en tres dimensiones ( $x, y, z$ ). Incluir un constructor por defecto, un constructor de copia, una función `negar` () que transforme el punto en su opuesto (negativo), una función `norma` () que devuelva la distancia al punto desde el origen  $(0,0,0)$  y una función `visualizar` ().
- 17.6 Implementar una clase `Hora`. Cada objeto de esta clase representa una hora específica de un día, almacenando las horas, minutos y segundos como enteros. Incluir un constructor, funciones de acceso, una función `avanzar` () para avanzar (adelantar) la hora actual de un objeto existente, una función `poner_a_cero` para poner a cero la hora actual de un objeto existente y una función `visualizar` ().
- 17.7 Implementar una clase `Persona` que represente datos personales de una persona. Los miembros datos deben incluir el nombre de la persona, fecha de nacimiento y año de graduación en la universidad. Se debe incluir un constructor por defecto, un destrutor, funciones de acceso y función de visualización.
- 17.8 Implementar una clase `Cadena`. Cada objeto de la clase representa una cadena de caracteres. Los miembros datos son la longitud de la cadena y la cadena de caracteres actual. Además, se deben añadir constructo-

res, destructor, funciones de acceso y función de visualizar, así como incluir una función `caracter(int i)` que devuelva el carácter de la cadena representado por el parámetro `i` que representa el índice o lugar del carácter en la cadena.

**17.9** Implementar una clase `Matriz 2 × 2` que incluya un constructor por defecto, un constructor de copia, una

función `inversa()` que devuelva el inverso de la matriz, una función `det()` que devuelva el determinante de la matriz, una función lógica (boolean) `EsCero` que devuelva `cierto` o `falso` de acuerdo con que el determinante sea cero y una función `visualizar()`.



# Clases derivadas: herencia y polimorfismo

## Contenido

- 18.1 Clases derivadas
- 18.2 Tipos de herencia
- 18.3 Destructores
- 18.4 Herencia múltiple
- 18.5 Ligadura
- 18.6 Funciones virtuales

## 18.7 Polimorfismo

- 18.8 Uso del polimorfismo
- 18.9 Ligadura dinámica frente a ligadura estática
- 18.10 Ventajas del polimorfismo
  - › Resumen
  - › Ejercicios

## Introducción

### Conceptos clave

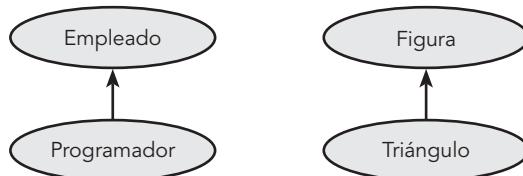
- › Clase abstracta
- › Clase base
- › Clase derivada
- › Constructor
- › Declaración de acceso
- › Destructor
- › Especificador de acceso
- › Función virtual
- › Herencia
- › Herencia múltiple
- › Herencia pública
- › Herencia privada
- › Herencia simple
- › Ligadura dinámica
- › Polimorfismo
- › Relación es-un
- › Relación tiene-un

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes orientados a objetos de los *lenguajes basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, de bajo costo, adaptable y reutilizable.

### 18.1 Clases derivadas

La *herencia* o *relación es-un* es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una clase que representa un concepto más general; en este caso la nueva clase puede *heredar* de la clase ya existente. Así por ejemplo, si existe una clase `Figura` y se desea crear una clase `Triángulo`, esta clase `Triángulo` puede derivarse de `Figura` ya que tendrá en común con



**Figura 18.1** Clases derivadas.

ella un estado y un comportamiento, aunque luego tendrá sus características propias. Triángulo *es-un* tipo de Figura. Otro ejemplo puede ser Programador, que *es-un* tipo de Empleado.

Evidentemente, la clase base y la *clase derivada* tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, muchos de los miembros de la clase base se repetirán en la clase derivada. C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase hereda todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el *especificador de acceso* que indica el tipo de herencia (*pública*, *privada* y *protegida*). La primera línea de cada declaración debe incluir el formato siguiente:

class nombre clase : tipo herencia nombre clase base

## Regla

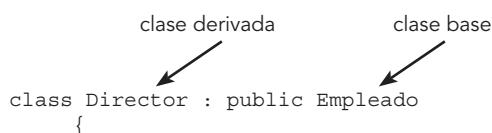
En general, se debe incluir la palabra reservada `public` en la primera línea de la declaración de la clase derivada; representa *herencia pública*. Esta palabra reservada consigue que todos los miembros que son públicos en la clase base permanezcan públicos en la clase derivada.

## Declaracin de las clases Programador y Triangulo.

### Ejemplo 18.1

```
1. class Programador : public Empleado {
 public:
 // miembros públicos
 private:
 // miembros privados
 };
2. class Triangulo : public Figura
{
 public:
 // sección pública
 ...
 private:
 // sección privada
 ...
};
```

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.



```

public:
 nuevas funciones miembro
private:
 nuevos miembros dato
};

```

En la definición de la clase `Director` solo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase `Empleado` son heredados automáticamente por la clase `Director`. Por ejemplo, la función `calcular_salario` de `Empleado` se aplica automáticamente a los directores:

```

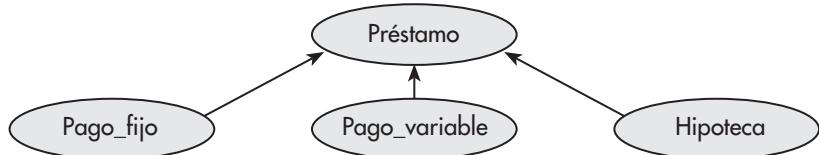
Director d;
d.calcular _salario(325000);

```



### Ejemplo 18.2

Considérese una clase `Prestamo` y tres clases derivadas de ella: `Pago_fijo`, `Pago_variable` e `Hipoteca`.



```

class Prestamo {
protected:
 float capital;
 float tasa_interes;
public:
 Préstamo(float, float);
 virtual int CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS])=0;
};

Las variables capital y tasa_interes no se repiten en la clase derivada

```

```

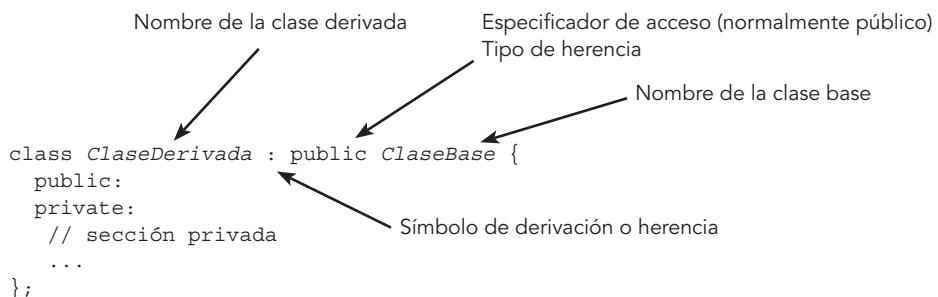
class Pago_fijo : public Préstamo {
private:
 float pago; // cantidad mensual a pagar por cliente
public:
 Pago_Fijo (float, float, float);
 int CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS]);
};

class Hipoteca : public Prestamo {
private:
 int num_recibos;
 int recibos_por_anyo;
 float pago;
public:
 Hipoteca(int, int, float, float);
 int CrearTablaPagos(float [MAX_TERM] [NUM_COLUMNAS]);
};

```

## Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:



- *Especificador de acceso* `public` significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.
- *Herencia pública* es aquella en que el especificador de acceso es `public` (*público*).
- *Herencia privada* es aquella en que el especificador de acceso es `private` (*privado*).
- *Herencia protegida* es aquella en que el especificador de acceso es `protected` (*protector*).

El especificador de acceso que declara el tipo de herencia es opcional (`public`, `private` o `protected`); si se omite el especificador de acceso, se considera por defecto `private`. La *clase base* (`ClaseBase`) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```

class nombre_clase : especificador_acceso opcional ClaseBase {
lista_de_miembros;
};

```

### Ejercicio 18.1

Representar la jerarquía de clases de publicaciones que se distribuyen en una librería: revistas, libros, etcétera.

Todas las publicaciones tienen en común una editorial y una fecha de publicación. Las revistas tienen una determinada periodicidad, lo que implica el número de ejemplares que se publican al año, y por ejemplo, el número de ejemplares que se ponen en circulación controlados oficialmente (por ejemplo, en España la OJD). Los libros, por el contrario tienen un código de ISBN y el nombre del autor:



Las clases en C++ se especifican así:

```

class Publicacion {
public:
 void NombrarEditor(const char *s);
 void PonerFecha(unsigned long fe);

private:
 String editor;
 unsigned long fecha;
};

class Revista : public Publicacion {
public:
 void NumerosPorAnyo(unsigned n);
 void FijarCirculacion(unsigned long n);
}

```

```

private:
 unsigned numerosPorAnyo;
 unsigned long circulacion;
};

class Libro : public Publicacion {
public:
 void PonerISBN(const char *s);
 void PonerAutor (const char *s);

private:
 String ISBN;
 String autor;
};

```

Así, en el caso de un objeto `Libro`, este contiene miembros dato y funciones heredadas del objeto `Publicacion`, así como `ISBN` y nombre del autor. En consecuencia serán posibles las siguientes operaciones:

```

Libro L;
L.NombrarEditor("McGraw-Hill");
L.PonerFecha(990606);
L.PonerISBN("84-481-2015-9");
L.PonerAutor("Mackoy, José Luis");

```

Por el contrario, las siguientes operaciones solo se pueden ejecutar sobre objetos `Revista`:

```

Revista R;
L.NumerosPorAnyo(12);
L.FijarCirculacion(300000L);

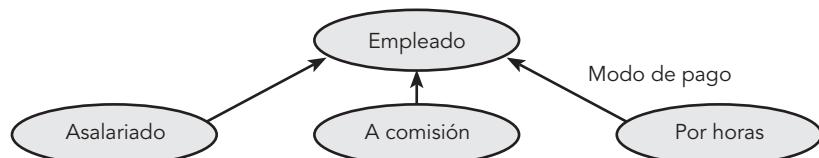
```

Si no existiese la posibilidad de utilizar la herencia, sería necesario hacer una copia del código fuente de una clase, darle un nuevo nombre y añadirle nuevas operaciones y/o miembros dato. Esta situación provocaría una dificultad para el mantenimiento, ya que siempre que se hacen cambios en la clase original, los correspondientes cambios tendrán que hacerse también en cualquier "clase copiada".

## Consideraciones de diseño

A veces, es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas.



Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fija o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es, ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿Esta relación debe ser el fundamento del diseño de clases? Evidentemente, la respuesta adecuada solo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

## 18.2 Tipos de herencia

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los elementos privados son accesibles solo a los miembros de la clase en que están definidos; se puede acceder a los elementos protegidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior, existen tres tipos de herencia: *pública*, *privada* y *protegida*. Normalmente, el tipo de herencia más utilizada es la herencia pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

### Norma

De manera predeterminada, la herencia es privada. Si accidentalmente se olvida la palabra reservada `public`, los elementos de la clase base serán inaccesibles. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar, si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

### Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y protegidos de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos.

Tabla 18.1 Acceso a variables y funciones según tipo de herencia.

| Tipo de herencia | Tipo de elemento       | ¿Accesible a clase derivada? |
|------------------|------------------------|------------------------------|
| public           | <code>public</code>    | sí                           |
|                  | <code>protected</code> | sí                           |
|                  | <code>private</code>   | no                           |
| private          | <code>public</code>    | sí                           |
|                  | <code>protected</code> | sí                           |
|                  | <code>private</code>   | no                           |

La herencia pública se representa con el especificador `public` en la derivación de clases.

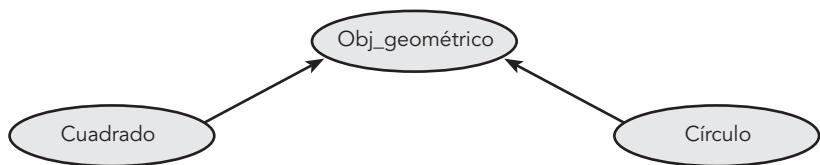
### Formato

```
class ClaseDerivada : public ClaseBase {
public:
 //sección pública
private:
 //sección privada
};
```



### Ejemplo 18.3

Considérese la jerarquía `Obj_geométrico`, `Cuadrado` y `Círculo`.



La clase `Obj_geom` de objetos geométricos se declara como sigue:

```
#include <iostream>
using namespace std;
class obj_geom {
public:
 obj_geom(float x=0, float y=0) : xC(x), yC(y) { }
 void imprimircentro() const
 {
 cout << xC << " " << yC << endl;
 }
protected:
 float xC, yC;
};
```

Un círculo se caracteriza por su centro y su radio. Un cuadrado se puede representar también por su centro y uno de sus cuatro vértices. Declaremos las dos figuras geométricas como clases derivadas.

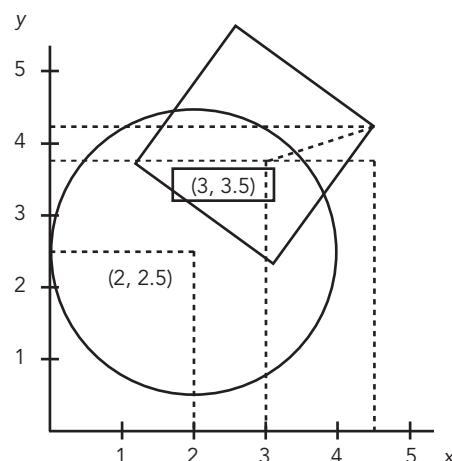


Figura 18.2 Círculo (centro: 2, 2.5), cuadrado (centro: 3, 3.5).

```

const float PI = 3.14159265;

class circulo : public obj_geom {
public:
 circulo(float x_C, float y_C, float r) : obj_geom (x_C, y_C)
 {
 radio = r;
 }
 float area() const {return PI * radio * radio; }
private:
 float radio;
};

class cuadrado : public obj_geom {
public :
 cuadrado(float x_C, float y_C, float x, float y)
 : obj_geom(x_C, y_C)
 {
 x1 = x;
 y1 = y;
 }
 float area() const
 {
 float a, b
 a = x1 - xC;
 b = y1 - yC;
 return 2 * (a * a + b * b);
 }
private:
 float x1, y1;
};

```

Todos los miembros públicos de la clase base `obj_geom` se consideran también como miembros públicos de la clase derivada `cuadrado`. La clase `cuadrado` deriva públicamente de `obj_geom`. Se puede escribir

```

cuadrado C(3, 3.5, 4.37, 3.85);
C.imprimircentro();

```

Aunque `imprimircentro` no sucede directamente en la declaración de la clase `cuadrado`, es, no obstante, una de sus funciones miembro públicas ya que es un miembro público de la clase `obj_geom` de la que se deriva públicamente `cuadrado`. Otro punto observado es el uso de `xC` y `yC` en la función miembro `area` de la clase `cuadrado`. Estos son miembros protegidos de la clase base `Obj_geom`, por lo que tienen acceso a ellos desde la clase derivada.

Una función `main` que utiliza las clases `cuadrado` y `círculo` y la salida que se produce tras su ejecución:

```

int main()
{
 círculo C(2, 2.5, 2);
 cuadrado Cuad(3, 3.5, 4.37, 3.85);
 cout << "centro del circulo : " << C.imprimircentro();
 cout << "centro del cuadrado : " << Cuad.imprimircentro();
 cout << "Area del circulo : " << C.area() << endl;
 cout << "Area del cuadrado : " << Cuad.area() << endl;
 return 0;
}

Centro del circulo : 2 2.5
Centro del cuadrado : 3 3.5

```

```
Area del circulo : 12.5664
Area del cuadrado : 3.9988
```

### Regla

Con herencia pública, los miembros de la clase derivada heredados de la clase base tienen la misma protección que en la clase base. La herencia pública se utiliza casi siempre ya que modela directamente la relación **es-un**.

## Herencia privada

La herencia privada significa que un usuario de la clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```
class ClaseDerivada : private ClaseBase {
 public:
 // sección pública
 protected:
 // sección protegida
 private:
 // sección privada
};
```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada.

La herencia privada se utiliza con menos frecuencia que la herencia pública. Este tipo de herencia oculta la clase base al usuario y así es posible cambiar la implementación de la clase base o eliminarla toda ella sin requerir ningún cambio al usuario de la interfaz. Cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

## Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que el usuario de la clase utiliza. La herencia protegida es todavía menos frecuente que la herencia privada.

La tabla 18.2 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.

**Tabla 18.2** Tipos de herencia y accesos que permiten.

| Tipo de herencia | Acceso a miembro clase base    | Acceso a miembro en clase derivada           |
|------------------|--------------------------------|----------------------------------------------|
| public           | public<br>protected<br>private | public<br>protected<br><i>inaccesible</i>    |
| protected        | public<br>protected<br>private | protected<br>protected<br><i>inaccesible</i> |
| private          | public<br>protected<br>private | private<br>private<br><i>inaccesible</i>     |

Declarar una clase base (`Base`) y tres clases derivadas de ella, `D1`, `D2` y `D3`.

### Ejemplo 18.4

```
class Base {
public:
 int i1;
protected:
 int i2;
private:
 int i3;
};

class D1 : private Base {
 void f();
};

class D2 : protected Base {
 void g();
};

class D3 : public Base {
 void h();
};
```

Ninguna de las subclases tienen acceso al miembro `i3` de la clase `Base`. Las tres clases pueden acceder a los miembros `i1` e `i2`. En la definición de la función miembro `f( )` se tiene:

```
void D1::f() {
 i1 = 0; //Correcto
 i2 = 0; //Correcto
 i3 = 0; //Error
};
```

El acceso a `i1`, `i2` e `i3` desde el exterior de las tres clases se muestra en las siguientes sentencias:

```
Base b; b.i1 = 0; //Correcto
b.i2 = 0; //Error
b.i3 = 0; //Error
D1 d1;
d1.i1 =0; //Error
d1.i2 =0; //Error
d1.i3 =0; //Error
D2 d2;
d2.i1 =0; //Error
d2.i2 =0; //Error
d2.i3 =0; //Error
D3 d3;
d3.i1 =0; //Correcto
d3.i2 =0; //Error
d3.i3 =0; //Error
};
```

## Operador de resolución de ámbito

Si se utiliza herencia privada o protegida, existe una forma de hacer a los miembros de la clase base accesibles en una clase derivada: utilizar una *declaración de acceso*. Esto se consigue nombrando uno de los miembros de la clase base en un lugar apropiado de la clase derivada.

```
class D4 : protected Base {
public:
```

```
Base::i1; // declaración de acceso
};
```

Al declarar `i1` en la sección pública de `D4` también hace a `i1` público en `D4`. Se puede, entonces, escribir

```
D4 d4;
d4.i1 = 0; // Correcto
```

De modo similar se puede hacer a `i2` protegido en `D5` nombrando `i2` en la sección protegida de `D5`.

```
class D5 : private Base {
protected:
 Base::i2; // declaración de acceso
};
```

## Constructores-inicializadores en herencia

Una clase derivada es una especialización o extensión de una clase base. En consecuencia, al crear un objeto de la clase derivada se crea la parte que tiene de la clase base, para lo cual se llama al constructor de la clase base. Una vez ejecutado el *constructor* de la clase base se activa el constructor propio de la clase derivada.

### Regla

1. Los constructores de las clases base se invocan antes del constructor de la clase derivada; los constructores de la clase base se invocan en la secuencia en que están especificados.
2. Si una clase base es, a su vez, una clase derivada, sus constructores se invocan también en secuencia: constructor base, constructor derivada.
3. Los constructores no se heredan, aunque los constructores por defecto y de copia, se generan si se requiere.



### Ejemplo 18.5

```
class B1 {
public:
 B1() { cout << "C-B1" << endl; }
};

class B2 {
public:
 B2() { cout << "C-B2" << endl; }
};

class D : public B1, B2 {
public:
 D() { cout << "C-D" << endl; }
};

D d1;
```

La construcción del objeto `d1` visualiza: C-B1, C-B2, C-D //un resultado por línea



### Ejemplo 18.6

```
class B1 {
public:
 B1(){cout <<"C-B1" <<endl;}
};

class B2 {
public:
```

```

B2() {cout <<"C-B2" <<endl;}
};

class D1 :public B1{
public:
 D1() {cout <<"C-D1" <<endl;}
};

class D2 : public D1, public B2{
public:
 D2() {cout <<"C-D2" <<endl;}
};

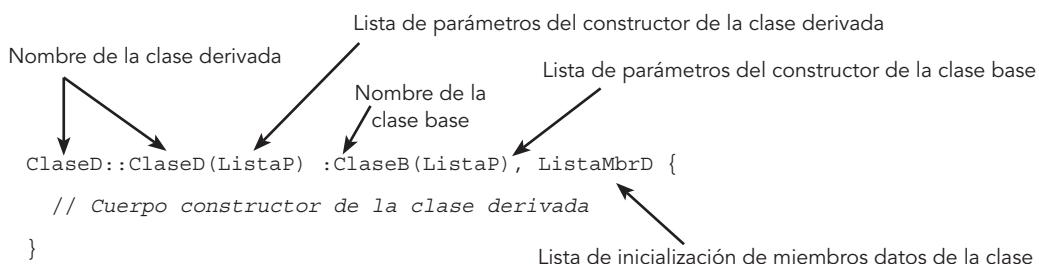
D2 d2;

```

La construcción del objeto d2 visualiza: C-B1, C-D1, C-B2 y C-D2 //un resultado por línea

## Sintaxis del constructor

La sintaxis de un constructor de una clase derivada es:



Obsérvese que la primera línea incluye una llamada al constructor de la clase base. El constructor de la clase base se llama antes de que se ejecute el cuerpo del constructor de la clase derivada. Esta secuencia tiene sentido ya que el objeto base constituye el fundamento del objeto derivado (se necesita el objeto base antes de convertirse en objeto derivado). El constructor de una clase derivada tiene que realizar dos tareas:

- Inicializar el objeto base.
- Inicializar todos sus miembros dato.

La clase derivada tiene un *constructor-inicializador*, que llama a uno o más constructores de la clase base. El inicializador aparece inmediatamente después de los parámetros del constructor de la clase derivada y está precedido por dos puntos (:).

La clase Punto3D es una clase derivada de la clase Punto.

### Ejemplo 18.7

Formato general: Punto3D::Punto3D(listaP) : inicializador-constructor

En la implementación del constructor de Punto3D se pasan los parámetros xv y yv al constructor de Punto.

```

class Punto {
public:

```

```

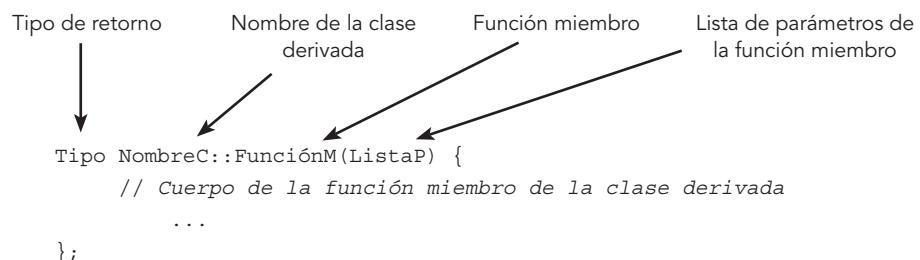
 Punto(int xv, int yv);
 // ...
 };
 class Punto3D: public Punto {
 public:
 Punto3D(int xv, int yv, int zv);
 void FijarZ(int);
 private:
 int z;
 };
 Punto3D ::Punto3D(int xv, int yv, int zv): Punto(xv, yv) {
 FijarZ(zv);
 }
}

```

Se ha implementado `Punto3D` fuera de la definición de la clase para mostrar cómo el constructor inicializador no aparece en el prototipo de la función. Si `Punto3D` se implementara en el interior de la definición de la clase, el inicializador aparecería allí.

### Sintaxis de la implementación de una función miembro

La sintaxis para la definición de la implementación de las funciones miembro de una clase derivada es idéntica a la sintaxis de la definición de la implementación de una clase base.



## 18.3 Destructores

Afortunadamente los destructores son mucho más fáciles de tratar que los constructores. Los destructores no se heredan, aunque se genera un *destructor* en forma predeterminada si se requiere. Un destructor normalmente solo se utiliza cuando un constructor de la clase ha asignado espacio de memoria que debe ser liberado. Los destructores se manejan como los constructores excepto que todo se hace en orden inverso. Esto significa que el destructor de la última clase derivada se ejecuta primero.



### Ejemplo 18.8

```

class C1 {
public:
 C1(int n);
 ~C1();
private:
 int *pi, l;
};
C1::C1(int n) :l(n)
{
 cout << l << " enteros se asignan " << endl;
 pi = new int[l];
}
C1::~C1()
{
}

```

```

cout << l << " enteros son liberados " << endl;
delete[] pi;
}
class C2 : public C1 {
public :
 C2(int n);
 ~C2();
private :
 char *pc;
 int l;
} ;
C2::C2(int n) :C1(n), l(n)
{
 cout << l << " caracteres son asignados " << endl;
 pc = new char[l];
}
C2::~C2()
{
 cout << l << " caracteres son liberados " << endl;
 delete[] pc;
}
void main()
{
 C2 a(50), b(100);
}

```

Cuando de ejecuta el programa, se imprimirá:

```

50 enteros se asignan
50 caracteres son asignados
100 enteros se asignan
100 caracteres son asignados
100 caracteres son liberados
100 enteros son liberados
50 caracteres son liberados
50 enteros son liberados

```

## 18.4 Herencia múltiple

*Herencia múltiple* es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras, hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes o padres*) para la clase derivada (*descendiente o hija*).

La herencia múltiple entraña un concepto más complicado que la *herencia simple*, no solo con respecto a la sintaxis sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la figura 18.3 se muestran diferentes ejemplos de herencia múltiple.

### Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene solo un parente). Herencia múltiple implica múltiples clases bases (tiene varios padres una clase derivada).

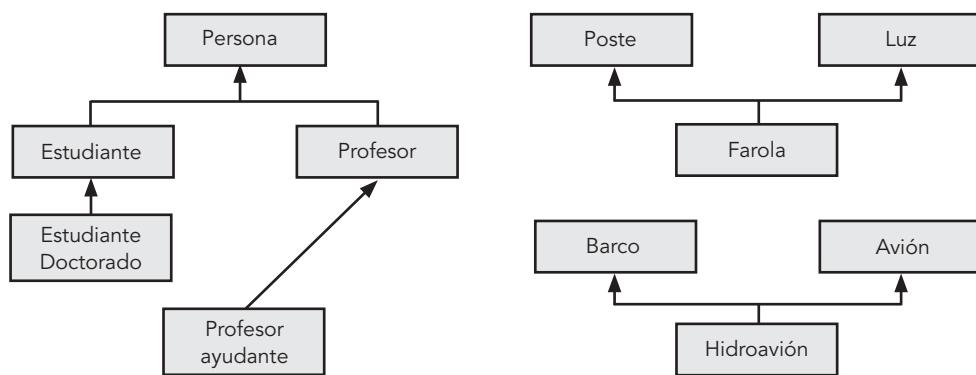


Figura 18.3 Ejemplos de herencia múltiple.

En herencia simple, el escenario es bastante sencillo, en términos de concepto y de implementación. En herencia múltiple los escenarios varían ya que las clases base pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Por qué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en el momento del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tienen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```

class Cderivada : [virtual] [tipo_acceso] Base1,
 [virtual] [tipo_acceso] Base2,
 [virtual] [tipo_acceso] Basen {
public:
 // sección pública
private:
 // sección privada
...
};

CDerivada Nombre de la clase derivada
tipo_acceso public, private o protected, con las mismas reglas que la herencia simple
Base 1, Base2,... Clases base con nombres diferentes
virtual... La palabra reservada virtual es opcional y especifica una clase base compatible.

```

Funciones o datos miembro que tengan el mismo nombre en Base1, Base2, Basen, serán motivo de ambigüedad.

### Ejemplos

```

class A : public B, public C { . . . };
class D : public E, private F, public G { . . . };
class X : Y, Z { . . . };
class M : virtual public N, virtual public P { . . . };

```

La palabra reservada `public`, ya se ha comentado anteriormente, define la relación **es-un** y crea un subtipo para herencia simple. Así, en los ejemplos anteriores, la clase `A` **es-un** tipo de `B` y **es-un** tipo de `C`. La clase `D` se deriva públicamente de `E` y `G` y privadamente de `F`. Esta derivación hace a `D` un subtipo de `E` y `G` pero no un subtipo de `F`. La clase `X` se deriva privadamente de `Y` y `Z`. Las clases `N` y `P` son clases base virtuales de `M` (se examinará más tarde el concepto). *El tipo de acceso solo se aplica a una clase base.*

### Ejemplo

```

class Derivada : public Base1, Base2 { . . . };

```

Derivada especifica derivación pública de `Base1` y derivación privada (por defecto u omisión) de `Base2`.

### Regla

Asegúrese de especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente `private` cuando lo necesite para manejar la legibilidad.

```
class Derivada : public Base1, private Base2 { . . . }
```

### Ejemplo 18.9

```
class Estudiante {
 ...
};

class Trabajador {
 ...
};

class Estudiante_Trabajador : public Estudiante, public Trabajador {
 ...
};
```

## Características de la herencia múltiple

### Ambigüedades

### Ejemplo 18.10

```
class Ventana {
 private:
 ...
 public :
 void dimensionar(); // dimensiona una ventana
 ...
};

class Fuente {
 private:
 ...
 public:
 void dimensionar(); // dimensiona un tipo fuente
 ...
};
```

Una clase `Ventana` tiene una función `dimensionar()` que cambia el tamaño de la ventana; de modo similar, una clase `Fuente` modifica los objetos `Fuente` con `dimensionar()`. Si se crea una clase `Ventana Fuente` (`VFuente`) con herencia múltiple, se puede producir ambigüedad en el uso de `dimensionar()`.

```
class VFuente : public Ventana, public Fuente {...};
VFuente v;
v.dimensionar(); // se produce un error, ¿cuál?
v.Fuente::dimensionar(); // llamada a dimensionar de Fuente
v.Ventana::dimensionar(); // llamada a dimensionar de Ventana
```

### Precaución

No es un error definir un objeto derivado con multiplicidad que tenga ambigüedades. Estas se consideran ambigüedades potenciales y solo producen errores en tiempo de compilación cuando se llaman de modo ambiguo. Esta ambigüedad se resuelve con el operador `::`

### Regla

Incluso es mejor solución, que la citada anteriormente, resolver la ambigüedad en las propias definiciones de la función `dimensionar()`.

```
class VFuente : public Ventana, public Fuente
{
 ...
 void v_dimensionar() { Ventana::dimensionar(); }
 void f_dimensionar() { Fuente::dimensionar(); }
};
```



### Ejemplo 18.11

```
class Trabajador {
public:
 const int no_ss;
 const char* nombre;
 ...
};

class Estudiante {
public:
 const char* nombre;
 ...
};

class Estu_Trabajador: public Estudiante, public Trabajador {
public:
 void imprimir() { cout << "número ss " << no_ss << endl;
 cout << nombre; ... } // error
 ...
};
```

Para evitar error en la invocación a `nombre` se debe hacer uso del operador de resolución de ámbito.

```
Estudiante :: nombre
Trabajador:: nombre
```

### Dominación (prioridad)

Un problema que se plantea cuando se manejan clases derivadas, es la *dominación* o *prioridad* que se produce cuando existen funciones en clases derivadas con el *mismo nombre* que otras funciones de las clases base. Consideremos el caso siguiente en herencia simple y luego extenderemos los conceptos a la herencia múltiple:

```
class Base {
public:
 void f(int);
 void f(char);
 void f(const String &);
};

class Derivada : public Base {
public:
 void f(const String &); // oculta todas las funciones Base::f()
};
```

```
Derivada d;
d.f(5); // error, Base::f(int) está oculta
d.f('x'); // error, Base::f(char) está oculta
d.f("abc"); // correcto, definida Derivada::f(const String &);
```

En herencia simple, las funciones en las clases derivadas con el *mismo nombre* que funciones de la clase base, *anulan* (reemplazan) a todas las versiones de la clase base. En el caso anterior, la función `Derivada::f()` oculta a todas las versiones de `Base::f()`. No se puede llamar a `f(int)` o a `f(char)` con un objeto de `Derivada` ya que `f()` de `Derivada` domina (tiene prioridad) a todas las `f()` de la clase `Base` incluso, aunque sus signaturas sean distintas.

Con herencia múltiple, una clase derivada por multiplicidad puede heredar nombres de funciones con igual nombre dos clases base diferentes. Ninguna clase base *domina* (tiene prioridad) sobre la otra. Esta *ausencia de dominio* crea una ambigüedad, incluso con signaturas (prototipos) distintas.

```
class A {
public:
 ...
 void f (int);
};

class B {
public:
 ...
 void f (const String &);
};

class C: public A, public B { . . . };
C c;
c.f (15); // error, ambiguo
```

### Ejemplo 18.12

Cuando se llama a `f(int)` con un objeto de la clase `C`, el compilador informa de una ambigüedad, incluso aunque la signatura en `A` sea diferente de la de `B::f(const String &)`. El nombre de `f` es ambiguo, debido a que ninguna clase base domina sobre la otra. La resolución de la sobrecarga no se aplica a través del ámbito de la clase.

#### Regla

Cuando la dominación crea ambigüedades, se realizan llamadas directas en la clase `Derivada` a la respectiva clase `Base`.

```
class C : public A, public B {
public:
 ...
 void f(int i) {A::f(i); }
 void f(const String &s) {B::f(s); }
};
```

## Inicialización de la clase base

¿Cómo llama el compilador a los constructores y destructores de las clases con herencia múltiple? Un ejemplo aclarará este problema.

```
class A { // clase base
public:
 A(int); // constructor con argumentos
```

```

 ~A();
 // destructor
 };
 class B {
 // clase base
 public:
 B();
 // constructor sin argumentos
 ~B();
 // destructor
 };
 class C {
 // clase base
 public:
 C(double);
 // constructor con argumentos
 ~C();
 // destructor
 };
 class D : public A, public B, public C { // herencia múltiple
 public:
 D(int i, double m) :A(i), C(m) { } // constructores de A,B,C y D
 ~D();
 // destruye D,C,B,A
 };

```

La clase `D` deriva públicamente de las clases base `A`, `B` y `C`. Los constructores de `A` y `C` tienen un argumento cada una, mientras que `B` tiene un constructor `void`. Para construir objetos `D`, el constructor `D` pasa su argumento entero al constructor `A` y su argumento `double` al constructor de `C`. Obsérvese que la lista de inicialización de miembros no necesita pasar argumentos al constructor de `B`.

La *definición de la clase* determina el orden de las llamadas de constructores y destructores, no la lista de inicialización de miembros. Esto implica que las listas de inicialización de miembros puede inicializar clases base en *cualquier orden*. En otras palabras, el siguiente constructor modificado de `D` llama a todos los constructores base en el mismo orden que el constructor de `D` anterior.

```
D(int i, double m) : C(m), A(i) { } // construye A,B,C,D
```

Los cambios a la definición de la clase, por otra parte, afectan al orden de llamadas de constructores.

```

class D : public C, public B, public A {
public:
 D(int i, double m) : A(i), C(m) { } // construye C,B,A,D
 ~D();
 // destruye D,A,B,C
};

```

### Precaución

Si el orden de llamadas de los constructores base es importante en una declaración de herencia múltiple, asegúrese de que la lista de clases base es correcta.

### Regla

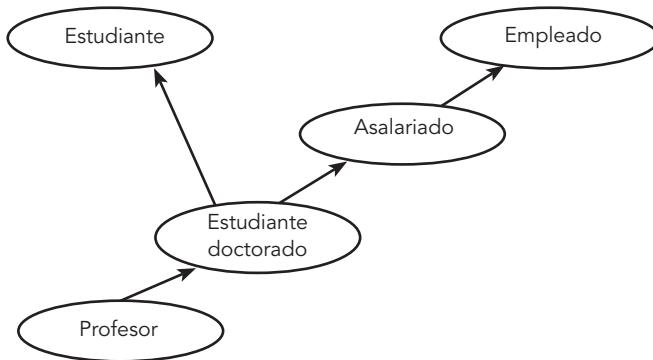
Orden de ejecución de constructores

1. Clases base inicializadas en orden de declaración.
2. Miembros inicializados en orden de declaración.
3. El cuerpo del constructor.



### Ejemplo 18.13

Diseñar e implementar una jerarquía de clases que represente las relaciones entre las clases siguientes: estudiante, empleado, empleado asalariado y un estudiante de doctorado que es, a su vez, profesor de prácticas de laboratorio.



Nota: se deja la resolución como ejercicio al lector.

## 18.5 Ligadura

*Ligadura* representa, generalmente, una conexión entre una entidad y sus propiedades. Si la propiedad se limita a funciones, ligadura es la conexión entre la llamada a función y el código que se ejecuta tras la llamada. Desde el punto de vista de atributos, la *ligadura* es el proceso de asociar un atributo a un nombre.

El momento en que un atributo o función se asocia con sus valores o funciones se denomina *tiempo de ligadura*. La ligadura se clasifica según sea el tiempo o momento de la ligadura: *estática* y *dinámica*. Ligadura estática se produce antes de la ejecución (durante la compilación), mientras que la *ligadura dinámica* ocurre durante la ejecución. Un atributo que se liga dinámicamente es un atributo dinámico. En un lenguaje de programación con ligadura estática, todas las referencias se determinan en tiempo de compilación. La mayoría de los lenguajes procedimentales son de ligadura estática; el compilador y el enlazador definen directamente la posición fija del código que se ha de ejecutar en cada llamada a la función.

La ligadura dinámica supone que el código a ejecutar en respuesta a un mensaje no se determinará hasta el momento de la ejecución. Solo la ejecución del programa (normalmente el valor de un apuntador a una clase base) determinará la ligadura efectiva entre las diversas que son posibles (una para cada clase derivada).

La principal ventaja de la ligadura dinámica frente a la ligadura estática es que la ligadura dinámica ofrece un alto grado de flexibilidad y manejar jerarquías de clases de un modo muy simple. Entre las desventajas está que la ligadura dinámica es menos eficiente que la ligadura estática.

Los lenguajes orientados a objetos que siguen estrictamente el paradigma orientado a objetos ofrecen solo ligadura dinámica. Los lenguajes que representan un compromiso entre el paradigma orientado a objetos y los lenguajes imperativos (como Simula y C++) ofrecen la posibilidad de elección con un tipo en forma predeterminada. En Simula y C++ la ligadura es estática en forma predeterminada y cuando se utiliza la declaración *virtual* se utiliza ligadura dinámica. En C++, siempre que se omite el especificador *virtual*, se supone que las referencias se resuelven en tiempo de compilación.

La ligadura en C++ es, *de manera predeterminada, estática*. La ligadura *dinámica* se produce cuando se hace preceder a la declaración de la función con la palabra reservada *virtual*. Sin embargo, puede darse el caso de ligadura estática, pese a utilizar *virtual*, a menos que el receptor se utilice como un apuntador o como una referencia.

## 18.6 Funciones virtuales

De manera predeterminada, las funciones C++ tienen ligadura estática; si la palabra reservada *virtual* precede a la declaración de una función, esta función se llama virtual, y le indica al compilador que puede ser definida (implementado su cuerpo) en una clase derivada y que en este caso la función se invocará directamente a través de un apuntador. Se debe calificar una función miembro de una clase con la palabra reservada *virtual* solo cuando exista una posibilidad de que otras clases puedan ser derivadas de aquella.

Un uso común de las funciones virtuales es la declaración de clases abstractas y la implementación del *polimorfismo*.

Consideremos la clase `Figura` como la clase base de la que se derivan otras clases, como `Rectangulo`, `Circulo` y `Triangulo`. Cada figura debe tener la posibilidad de calcular su área y poder dibujarla. En este caso, la clase `Figura` declara las siguientes funciones virtuales:

```
class Figura {
public:
 virtual double calcular_area(void) const;
 virtual void dibujar(void) const;

 // otras funciones miembro que definen una interfaz a
 // todos los tipos de figuras geométricas
};
```

Cada clase derivada específica debe definir sus propias versiones concretas de las funciones que han sido declaradas virtuales en la clase base. Por consiguiente, si se derivan las clases `Circulo` y `Rectangulo` de la clase `Figura`, se deben definir las funciones miembro `calcular_area` y `dibujar` en cada clase. Por ejemplo, las definiciones de la clase `Circulo` pueden ser similares a estas:

```
class Circulo : public Figura
{
public:
 virtual double calcular_area(void) const;
 virtual void dibujar(void) const;
 // ...
private:
 double xc, yc; // coordenada del centro
 double radio; // radio del círculo
};

#define PI 3.14159 // valor de "pi"
// Implementación de calcular_area
double Circulo::calcular_area(void) const
{
 return(PI * radio * radio);
}
// Implementación de la función "dibujar"
void Circulo::dibujar(void) const
{
 // ...
}
```

Cuando se declaran las funciones `dibujar` y `calcular_area` en la clase derivada, se puede añadir opcionalmente la palabra reservada `virtual` para destacar que estas funciones son verdaderamente virtuales. Las definiciones de las funciones no necesitan la palabra reservada `virtual`.

## Ligadura dinámica mediante funciones virtuales

```
Circulo c1;
Rectangulo r1;
double área = c1.calcular_area(); // calcular área del círculo
r1.dibujar(); // dibujar un rectángulo
```

El uso anterior es similar al de cualquier función miembro. En este caso, el compilador C++ puede determinar que se desea llamar a la función `calcular_area` de la clase `Circulo` y a la función `dibujar` de la clase `Rectangulo`. De hecho, el compilador hace llamadas directas a estas funciones, y las llamadas a funciones se enlazan a un código específico en tiempo de enlace. Esta es la ligadura que hemos denominado anteriormente *estática*.

Sin embargo, el caso más interesante es la llamada a las funciones a través de un apuntador a `Figura`, como:

```
Figura *s[10]
// crea tipos de figuras y almacena punteros en array "s"
```

```
// dibujar las figuras
for (i = 0; i < numfiguras; i++)
 figura[i] -> dibujar();
```

En este caso se produce ligadura dinámica.

### **Un apuntador (puntero) a una clase derivada es también un puntero a la clase base**

En C++ se puede utilizar una referencia o un apuntador a cualquier clase base, en lugar de una referencia o apuntador a la clase derivada, sin una conversión explícita de tipos. De modo que si `Circulo` y `Rectangulo` se derivan de la clase `Figura`, se puede llamar a una función que requiera un apuntador `Circulo` o `Rectangulo` con un apuntador a `Figura`.

Lo opuesto no es cierto; no se puede utilizar una referencia o un apuntador a la clase derivada, en lugar de una referencia o un apuntador a una instancia de una clase base.

El siguiente ejemplo muestra el uso de funciones virtuales y ligadura dinámica y las diferencias entre ligadura estática y dinámica

```
// archivo LIGADURA.CPP
#include <iostream>
using namespace std;
// define una clase A con dos funciones que imprimen sus nombres
class A {
public:
 A() { }
 virtual void Dinamica()
 {
 cout << "Función dinámica de la clase A" << endl;
 }
 void Estatica()
 {
 cout << "Función estática de la clase A" << endl;
 }
};
// define una clase B, derivada de A, redefiniendo ambas funciones
class B : public A {
public:
 B() { }
 void Dinamica()
 {
 cout << "Función dinámica de clase B" << endl;
 }
 void Estatica()
 {
 cout << "Función estática de clase B" << endl;
 }
};
void main()
{
 // definiciones
 A *a;
 B *b;
 // inicialización
 a = new A();
 b = new B();
 cout << "Funciones del objeto a de la clase A" << endl;
 a -> Dinamica();
```

```

a -> Estatica();
cout << endl;
cout << "Funciones del objeto b de la clase B" << endl;
b -> Dinamica();
b -> Estatica();
cout << endl;
// propiedad de polimorfismo
// a toma como valor un puntero a un objeto de la clase B
a = b;
cout << "Funciones del objeto a de la clase A" << endl;
<< "al que se ha asignado un valor de la clase b" << endl;
// llamada a la función virtual de la clase B
a -> Dinamica();
// llamada a la función no virtual de la clase B
a -> Estatica();
}

```

### A recordar

En C++, las funciones virtuales siguen una regla concreta: la función se debe declarar como `virtual` en la primera clase en que está presente. Esta regla significa que, normalmente, las funciones virtuales se declaran en la clase de nivel más alto de una jerarquía.

## 18.7 Polimorfismo

En POO, el polimorfismo permite que diferentes objetos respondan de modo distinto al mismo mensaje; adquiere su máxima potencia cuando se utiliza en unión de herencia.

Por ejemplo, si `figura` es una clase base de la que cada figura geométrica hereda características comunes, C++ permite que cada clase utilice una función (método) `Copiar` como nombre de una función miembro.

```

class figura {
 tipoenum tenum; // tipoenum es un tipo enumerado
public:
 virtual void Copiar();
 virtual void Dibujar();
 virtual double Area();
};

class circulo : public figura {
...
public:
 void Copiar();
 void Dibujar();
 double Area();
};

class rectangulo : public figura {
...
public:
 void Copiar(); // el polimorfismo permite que objetos diferentes
 // tengan idénticos nombres de funciones miembro
 void Dibujar();
 double Area();
};

```

Otro ejemplo se puede apreciar en la jerarquía de clases:

```
class Poligono { // superclase
```

```

public:
 float Perimetro();
 virtual float Area();
 virtual boolean PuntoInterior();
protected:
 void Visualizar();
};

class Rectangulo : public Poligono{
public:
 virtual float Area();
 virtual boolean PuntoInterior();
 void fijarRectangulo();
private:
 float Alto;
 float Bajo;
 float Izquierdo;
 float Derecho;
};

```

## El polimorfismo sin ligadura dinámica

Como ya se ha comentado anteriormente, el polimorfismo permite que diferentes objetos respondan de modo distinto al mismo mensaje; por esta razón, en los programas se puede pasar el mismo mensaje a objetos diferentes, como:

```

switch (...) {
 case circulo:
 miCirculo.Dibujar();
 break;
 case rectángulo:
 miRectangulo.Dibujar();
 d = MiRectangulo.Area();
 break;
 ...
}

```

En el ejemplo anterior, cada figura recibe el mismo mensaje (por ejemplo, `miCirculo.Dibujar`, `miRectangulo.Area()`, etc.). Esta solución, sin embargo, aunque utiliza polimorfismo, no es aceptable, ya que impone una selección del objeto que envía el mensaje mediante un selector. Este código de discriminación se puede eliminar utilizando ligadura dinámica.

## El polimorfismo con ligadura dinámica

Con ligadura dinámica, el tipo de objeto no es preciso decidirlo hasta el momento de la ejecución. El ejemplo de la sección anterior, la sentencia `switch` transfiere control a la etiqueta correspondiente al objeto que llama la etiqueta correspondiente al objeto que llama a la función, su mantenimiento será difícil, ya que añadir objetos requerirá modificaciones a cada sentencia `switch` que haga uso del registro discriminante.

Una solución que hace uso de la ligadura dinámica es la siguiente:

```

Figura * figura [] = {new Circulo (), new Rectangulo ()...};
// crea e inicializa un array de figuras
figuras[i]→Dibujar();

```

Este segmento de código pasará el mensaje `Dibujar` a la figura apuntada por `figura [i]`. La palabra clave `virtual` que se puso en la función `Dibujar` al declarar la clase base `Figura` ha indicado al compilador que esta función se puede llamar por un apuntador. Mediante la ligadura dinámica, el programa determina el tipo de objeto en tiempo de ejecución, eliminando la necesidad del registro discriminante y la sentencia `switch` asociada.

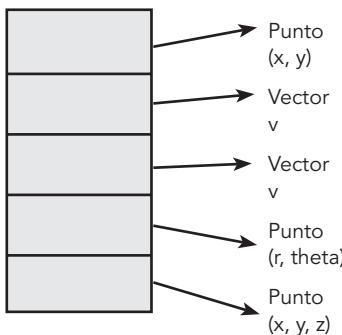


Figura 18.4 Referencias polimórficas.

El polimorfismo se puede representar con un array de elementos que se refieren a objetos de diferentes tipos (clases), como sugiere Meyer.<sup>1</sup> Así, en la figura 18.4 se muestra un array (arreglo) que incluye apuntadores que apuntan a diferentes tipos, que son todos derivados de una superclase.

## 18.8 Uso del polimorfismo

El polimorfismo permite utilizar la misma interfaz, como métodos (funciones miembro) denominados `dibujar` y `calcular_area`, para trabajar con toda clase de figuras.

Para poder utilizar polimorfismo en C++ se deben seguir estas reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por las funciones miembro declaradas como virtuales en la clase base.
2. Las implementaciones específicas de las funciones virtuales se deben hacer en las clases derivadas. Cada clase derivada puede tener su propia versión de las funciones. Por ejemplo, la implementación de la función `dibujar` varía de una figura a otra.
3. Las instancias de estas clases se manipulan a través de una referencia o un apuntador (puntero). Este mecanismo es la ligadura dinámica base.

Se obtiene ligadura dinámica solo cuando las funciones miembro virtuales se invocan a través de un apuntador de la clase base que dirige objetos de las clases derivadas.

## 18.9 Ligadura dinámica frente a ligadura estática

*Ligadura dinámica o postergada (tardía)* se produce cuando una función polimórfica se define para clases diferentes de una familia pero el código real de la función no se conecta o enlaza hasta el tiempo de ejecución. Una función polimórfica que se enlaza dinámicamente se llama *función virtual*.

La ligadura dinámica se implementa en C++ mediante *funciones virtuales*. Con ligadura dinámica, la selección del código a ejecutar cuando se llama a una función virtual se retrasa hasta el tiempo de ejecución. Esto significa que cuando se llama a una función virtual, el código ejecutable determina en tiempo de ejecución cuál es la versión de la función que se llama. Recordemos que las funciones virtuales son polimórficas y, por consiguiente, tienen diferentes implementaciones para clases distintas de la familia.

La ligadura estática se produce cuando el código de la función “se enlaza” en tiempo de compilación. Esto significa que cuando se llama una función no virtual, el compilador determina en tiempo de compilación cuál es la versión de la función a llamar. Las funciones sobrecargadas se enlazan estáticamente, mientras que las funciones virtuales se enlazan dinámicamente. Con funciones sobrecargadas, el compilador puede determinar cuál es la función a llamar basada en el número y tipos de datos de los parámetros de función. Sin embargo, las funciones virtuales tienen la misma interfaz dentro de una familia de clases dada.

<sup>1</sup> Meyer, B., *Object-Oriented Software Construction*, Nueva York, Prentice-Hall, 1998. (Esta obra fue traducida al español por un equipo de profesores universitarios coordinados y dirigidos por el autor de esta obra (profesor Luis Joyanes), que escribió también el prólogo a dicha edición española.)

Las funciones virtuales se declaran en una clase base en C++ utilizando la palabra reservada `virtual`. Cuando se declara una función como una función virtual de una clase base, el compilador conoce cuál es la definición de la clase base que se puede anular en una clase derivada. La definición de la clase base se anula (reemplaza) definiendo una implementación diferente para la misma función de la clase derivada. Si la definición de la clase base no se anula en una clase derivada, entonces la definición de la clase base está disponible a la clase derivada.

## 18.10 Ventajas del polimorfismo

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación; este es el caso de C++.

Sin embargo, otros lenguajes de programación orientada a objetos soportan un polimorfismo ilimitado que choca con la idea de la comprobación de tipos. Aunque ofrecen más flexibilidad y libertad, tienden a evitar casi cualquier comprobación del código fuente, postergándolo al momento de la ejecución. Se tiene más libertad para realizar la programación, pero se tiene más probabilidad de errores potenciales. Como resultado, el código producido con otros lenguajes orientados a objetos es menos seguro, debido a que las llamadas a funciones que se encuentran no se pueden definir para algunos objetos, por razones que van desde un simple error de sintaxis a un error conceptual o lógico (el programa resultante tiende a ser más lento, ya que muchas comprobaciones que se necesitan se hacen en tiempo de ejecución). En el lenguaje Smalltalk (que ignora la comprobación de tipos estática), un mensaje (llamada a función) puede no tener un método correspondiente, pero el entorno atrapará el error solo cuando se intenta ejecutar la instrucción defectuosa.

Aunque existe menos libertad, el polimorfismo en C++ es una herramienta muy potente y que puede ser utilizada en muchas situaciones diferentes. Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas.** El uso más común del polimorfismo es derivar clases especializadas de clases que han sido definidas. Así por ejemplo, una clase cuadrado es una especialización de la clase `rectangulo` (cualquier cuadrado es un tipo de rectángulo). Esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.
- **Estructuras de datos heterogéneos.** A veces es muy útil poder manipular conjuntos similares de objetos. Con polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos, que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.
- **Gestión de una jerarquía de clases.** Las jerarquías de clases son colecciones de clases altamente estructuradas que se pueden entender fácilmente.



## Resumen

- La relación ***es-un*** indica herencia. Por ejemplo, una rosa es un tipo de flor; un pastor alemán es un tipo de perro, etc. La relación ***es-un*** es transitiva. Un pastor alemán es un tipo de perro y un perro es un mamífero; por consiguiente, un pastor alemán es un mamífero. La ***relación tiene-un*** indica contenido. Por ejemplo, una radio tiene sintonizador, un coche tiene un motor.
- Una clase nueva que se crea a partir de una clase ya existente, utilizando herencia, se denomina *clase derivada* o *subclase*. La clase padre se denomina *clase base* o *superclase*.
- ***Herencia simple*** es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. La nueva clase se denomina *clase derivada*. Las relaciones de heren-

cia reducen código redundante en programas. Uno de los requisitos para que un lenguaje sea considerado orientado a objetos o basado en objetos es que soporte herencia.

- ***Herencia privada*** es el término que se utiliza cuando una clase derivada restringe el acceso a los miembros heredados haciéndolos privados. Esto impide a los usuarios de la clase derivada acceder a los miembros de la clase base.
- ***Herencia múltiple***, se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflicto de nombres, cosa que se produce cuando nombres idénticos aparecen en más de una clase base.

- *Poliformismo* es la propiedad de que “algo” tome diferentes formas. En un lenguaje orientado a objetos el poliformismo es la propiedad por la que un mensaje puede significar cosas diferentes dependiendo del objeto que lo recibe.
- Para implementar el poliformismo, un lenguaje debe soportar ligadura dinámica. La razón por la que el poli-

formismo es útil, es que proporciona la capacidad de manipular instancias de clases derivadas a través de un conjunto de operaciones definidas en su clase base. Cada clase derivada puede implementar las operaciones definidas en la clase base.

## Ejercicios

18.1 Definir una clase base *persona* que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: *estudiante*, *empleado*, *estudiante\_empleado*.

Escribir un programa que lea un archivo de información y cree una lista de personas: *a*) general; *b*) estudiantes; *c*) empleados; *d*) estudiantes empleados. El programa debe permitir ordenar alfabéticamente por el primer apellido.

18.2 Implementar una jerarquía *Libreria* que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etcétera.

18.3 Diseñar una jerarquía de clases que utilice como clase base o raíz una clase *LAN* (red de área local).

Las subclases derivadas deben representar diferentes topologías, como *estrella*, *anillo*, *bus* y *hub*. Los miembros datos deben representar propiedades como *soporte de transmisión*, *control de acceso*, *formato del marco de datos*, *estándares*, *velocidad de transmisión*, etc. **Se desea simular la actividad de los nodos de tal LAN.**

La red consta de nodos, que pueden ser dispositivos como computadoras personales, estaciones de trabajo, máquinas FAX, etc. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo, poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.

• Realizar estadísticas de la LAN, como tiempo medio que emplea un paquete.

18.4 Implementar una jerarquía *Empleado* de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para *Empleados* de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a *Empleados*. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o al Ministerio o Secretaría de Hacienda. Utilice la herencia para distinguir entre al menos cuatro tipos diferentes de acceso a la información de *Empleado*.

18.5 Implementar una clase *Automóvil* (Carro) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *Vehículo*, un automóvil es también *una comodidad*, *un símbolo de estado social*, *un modo de transporte*, etcétera. *Automóvil* debe tener al menos dos clases base y al menos tres clases derivadas.

18.6 Escribir una clase *FigGeométrica* que represente figuras geométricas como *punto*, *línea*, *rectángulo*, *triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.

18.7 El archivo *VIRTUAL.CPP* muestra las diferencias entre llamadas a una función virtual y a una función normal.

```
// archivo VIRTUAL.CPP
#include <iostream>
using namespace std;
class Base {
public:
 virtual void f() { cout << 'f()' : cla-
```

```
se base-!" << endl;
void g() {cout <<"g():clase base-!" <<
endl;}
};

class Derivada1:public Base {
public:
 virtual void f() {cout <<"f(): clase
Derivada-!" << endl;}
 void g() {cout <<"g() :clase Deriva-
da-!" << endl;}
};

class Derivada2 : public Derivada1 {
public:
 virtual void f() {cout << "f() : clase
Derivada2-! "endl; }
 void g() {cout << "g() :clase Deriva-
da2-!" << endl; }
};
```

```
void main()
{
 Base b;
 Derivada1 d1;
 Derivada2 d2;
 Base *p = &b;
 p -> f();
 p -> g();
 p = &d1;
 p -> f()
 p -> g();
 p = &d2;
 p -> f();
}
```

¿Cuál es el resultado de ejecutar este programa? ¿Por qué?



# Genericidad: plantillas (templates)

## Contenido

- 19.1 Genericidad
- 19.2 Conceptos fundamentales de plantillas en C++
- 19.3 Plantillas de funciones
- 19.4 Plantillas de clases

## 19.5 Una plantilla para manejo de pilas de datos

- 19.6 Modelos de compilación de plantillas
- 19.7 Plantillas frente a polimorfismo
  - › Resumen
  - › Ejercicios

## Introducción

Una de las ideas clave en el mundo de la programación es la posibilidad de diseñar clases y funciones que actúen sobre tipos arbitrarios o genéricos. Para definir clases y funciones que operan sobre tipos arbitrarios se definen los *tipos parametrizados* o *tipos genéricos*. La mayoría de los lenguajes de programación orientados a objetos proporcionan soporte para la genericidad: las unidades genéricas (paquetes o funciones) en Ada, las *plantillas* (templates) en C++. C++ proporciona la característica **plantilla** (*template*), que permite a los tipos ser parámetros de clases y funciones. C++ soporta esta propiedad a partir de la versión 2.1 de AT&T.

Las *plantillas* o *tipos parametrizados* pueden utilizarse para implementar estructuras y algoritmos que son en su mayoría independientes del tipo de objetos sobre los que operan. Por ejemplo, una plantilla **Pila** puede describir cómo implementar una pila de objetos arbitrarios; una vez que la plantilla se ha definido, los usuarios pueden escribir el código que utiliza pilas de tipos de datos reales, cadenas, enteros, apuntadores, etcétera.



## Conceptos clave

- › Funciones de plantilla
- › Genericidad
- › Plantillas de funciones
- › Polimorfismo
- › **template**
- › Tipo genérico
- › Tipo parametrizado

### 19.1 Genericidad

La *genericidad* es una construcción importante en un lenguaje de programación orientada a objetos, que si bien no es exclusivo de este tipo de lenguajes, es en ellos en los que ha adquirido verdadera carta de naturaleza, dado que ha servido principalmente para aumentar la reutilización.

La genericidad es una propiedad que permite definir una clase (o una función) sin especificar el tipo de datos de uno o más de sus miembros (parámetros). De esta forma, se puede cambiar la clase para adaptarla a los diferentes usos sin tener que reescribirla.

La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de datos que procesa y, sin embargo, cuando se implementan en un lenguaje de programación, los programas que resuelven cada algoritmo serán diferentes para cada tipo de dato que procesan. Por ejemplo, un algoritmo que implementa una pila de caracteres es esencialmente el mismo que el algoritmo necesario para implementar una pila de enteros o de cualquier otro tipo. Así, por ejemplo, en Pascal, C, COBOL, etc., se requiere un programa distinto para manejar una pila de enteros, de reales, de cadenas, etc. Sin embargo, en C++ 3.0 y en Java existen las plantillas (*templates*); en Ada, los paquetes genéricos, etc., que permiten definir unas *clases genéricas* o *paramétricas* que pueden implementar esas estructuras o clases con independencia del tipo de elemento que procesan. Es decir, las plantillas, paquetes, etc., pueden diseñar pilas de elementos con independencia del tipo de elemento que procesan. Las plantillas o unidades genéricas, tras ser implementadas, han de ser instanciadas para producir subprogramas, paquetes o clases reales que ya utilizan tipos de datos concretos. Las clases genéricas se denominan también **clases contenedoras** (*container class*), clases que contienen objetos de algún otro tipo. Ejemplos típicos de clases contenedoras son *pilas*, *colas*, *listas*, *conjuntos*, *diccionarios*, *arrays*, etc. Estas clases se definen con independencia del tipo de los objetos contenidos, y es el usuario de la clase quien deberá especificar el tipo de argumento de la clase en el momento que se instancia.

En el caso más común, los parámetros representan tipos. Los módulos reales, denominados instancias del módulo genérico, se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos. Booch señala cuatro métodos para construir clases contenedoras:

- Usar macros, en algunos lenguajes, para compilar una clase varias veces, cada vez que se proporciona una definición del tipo que realmente se utiliza.
- Otros lenguajes, como Smalltalk, evitan comprobación de tipos. En Smalltalk, el polimorfismo no se limita a jerarquías, como en C++, y cualquier clase puede ser sustituida por otra. Por consiguiente, cualquier contenedor es totalmente heterogéneo.
- Este enfoque puede ser ligeramente modificado añadiendo pruebas a los contenedores polimórficos.
- El último enfoque es el uso de módulos genéricos. Algunos lenguajes proporcionan un mecanismo para clases parametrizadas, como en los casos de Ada, Eiffel y C++.

La genericidad se implementa en Ada mediante unidades genéricas y en C++ con plantilla de clases y *plantillas de funciones*.

## 19.2 Conceptos fundamentales de plantillas en C++

Las **plantillas** (*templates*) fueron propuestas por Stroustrup<sup>1</sup> en la conferencia USENIX C++ de Denver, en 1988. En esa ocasión planteó las siguientes preguntas:

- ¿Puede ser fácil de utilizar la parametrización de tipos?
- ¿Pueden objetos de tipos parametrizados ser integrados en C++?
- ¿Pueden los tipos parametrizados ser implementados de modo que la velocidad de compilación y enlace sea similar al realizado por un sistema de compilación que no soporta parametrización de tipos?
- ¿Puede tal sistema de compilación ser simple y transportable?

Naturalmente, el mismo Stroustrup comentaba en la citada conferencia que su respuesta a todas esas preguntas era *sí*. El mecanismo de plantillas presentado en el ARM (*Annotated Reference Manual*) en julio de 1990 fue aceptado por el comité ANSI C++. Por fin, la versión 3.0 del C++ de AT&T introdujo la noción de plantilla (*template*), una construcción que permite escribir funciones y clases muy generales que pueden aplicarse a objetos de tipos diferentes. Existen dos tipos de plantillas: *plantillas de clases* y *plantillas de funciones*.

El interés de las plantillas proviene del hecho de que esta generalidad no arrastra pérdida de rendimiento y no obliga a sacrificar las ventajas de C++ en materia de control estricto de los tipos de datos.

<sup>1</sup> Stroustrup, Bjarne, *Parametrized Types for C++*. Proceeding Acts, Usenix C++ Conference, Denver, octubre de 1988.

La traducción de **template** en esta obra ha sido **plantilla**,<sup>2</sup> aunque también puede utilizarse *patrones*, *modelos*, o también *tipos genéricos* o *tipos parametrizados*. Asimismo, utilizaremos indistintamente los términos *plantilla de funciones* y *funciones plantillas*, o también *plantilla de clases* o *clases plantillas*.

Una *plantilla* es un patrón para creación de clases o funciones como *instancias* o *especializaciones* de la plantilla en tiempo de compilación, de igual forma que una clase es un patrón para crear objetos como instancias de la clase en tiempo de ejecución. Por esta razón, se denominan a veces *función patrón* y *clase patrón* en lugar de *plantilla de funciones* y *plantilla de clases*, como se verá más tarde.

En los lenguajes de programación es muy frecuente que las funciones que realizan una determinada tarea, deban ser definidas repetidamente a medida que los tipos de los parámetros son diferentes. Ejemplos típicos son las funciones *maximo/minimo* que devuelven en el *máximo* o *minimo* de dos valores enteros; *intercambiar* función que intercambia entre sí los valores enteros de dos variables.

Estas funciones se deben volver a implementar para cada pareja de tipos para las cuales se necesite el *máximo*, *mínimo* o el *intercambio* de valores.



### Ejemplo 19.1

Esta función se aplica a tipos enteros

```
void Intercambio (int& m, int& n)
{
 int aux,
 aux = m;
 m = n;
 n = aux;
}
```

Si alguna vez se desea intercambiar valores de tipo *char*, se puede sobrecargar la función *Intercambio* con la siguiente definición:

```
void Intercambio (char& var, char& var2)
{
 char aux;
 aux = var1;
 var1 = var2;
 var2 = aux;
}
```

Si ahora se desea utilizar la función *Intercambio* para aplicarla a pares de variables tipo *double* o *float*, se tendrán que escribir de nuevo una tercera y una cuarta definiciones de la función casi idéntica a las anteriores.

Igual sucedería si consideramos la función *maximo* de dos valores, o cualquier otra función. Se requiere mucho esfuerzo de implementación repetida de funciones. Por esta razón, sería importante disponer de una característica especial del lenguaje con la cual, la definición de función se pudiera aplicar a variables de cualquier tipo, con una sintaxis similar a

```
void Intercambio (tipo_variable& var1, tipo_variable& var2)
{
 tipo_variable aux;
 aux = var1;
 var1 = var2;
 var2 = aux;
}
```

<sup>2</sup> Este término es el que hemos utilizado en nuestras clases, cursos y conferencias impartidas en España y Latinoamérica en los últimos años. No obstante, hemos comprobado personalmente que también se utiliza en Latinoamérica el término *patrón* (este término es el empleado en la traducción del libro *El lenguaje de programación C++*, de Stroustrup, realizada en México).

Esta característica se dispone en C++ y permite definir funciones que se apliquen a todos los tipos de variables, aunque la sintaxis a utilizar será un poco más complicada. La entidad que permite estas características se denomina **plantillas** (*templates*).

Las plantillas o patrones son funciones y clases que no están implementadas para un tipo determinado, sino para un tipo que se debe definir en cada momento. Para utilizar estas funciones o clases el programador solo debe especificar el tipo para el cual debe realizarse la plantilla. Así la función `maximo` o `Intercambio` solo deben definirse e implementarse una vez. Igual sucede con clases contenedoras como pila, listas enlazadas, etc., que solo deben ser implementadas una vez.

### Terminología

Los términos utilizados para describir plantillas no están definidos con rigurosidad. Así, una función cuyo tipo está parametrizado se le conoce a veces como *plantilla de funciones*<sup>3</sup> (*function template*) y a veces como función plantilla. En general, nosotros la definiremos como *plantilla de funciones*. De modo similar para las clases con tipos parametrizados, utilizaremos *plantillas de clases* en lugar de *clases plantilla* o *clases patrón*.

El proceso de creación de una clase o función de miembro de una plantilla mediante la sustitución real de los valores de sus argumentos se llama *instanciación* de la plantilla. Recuerde que la palabra *instanciar* se utiliza en programación orientada a objetos para la creación de los objetos de una clase. Por consiguiente, el término *instanciar* en C++ dependerá del contexto en que se utiliza. Una *instancia*, *ejemplar* o *especialización* de una plantilla función crea una función; una instancia de una plantilla de clases crea una clase y todos sus miembros.

## 19.3 Plantillas de funciones

Una *plantilla de funciones* especifica un conjunto infinito de funciones sobrecargadas. Cada función de este conjunto es una *función plantilla* y una instancia de la plantilla de función. Una función plantilla apropiada se produce automáticamente por el compilador cuando es necesario.

Una plantilla de funciones se utiliza para definir un grupo de funciones que se pueden utilizar para tipos diferentes. Una plantilla de funciones es un conjunto indeterminado de funciones sobrecargadas y describe las propiedades específicas de una función. Se puede sobrecargar una plantilla de funciones con una función no plantilla o con otras plantillas de funciones. Se puede, incluso, disponer de una plantilla de función y una función no plantilla con el mismo nombre y parámetros.

### Fundamentos teóricos

Supongamos que se desea escribir una función `min(a, b)` que devuelve el valor más pequeño de sus argumentos.

C++ impone una declaración precisa de los tipos de argumentos necesarios que recibe `min()`, así como el tipo de valor devuelto por `min()`. Es necesario utilizar diferentes funciones sobrecargadas `min()`, cada una de las cuales se aplica a un tipo de argumento específico. Un programa que hace uso de funciones `min()` es:

```
#include <iostream>
using namespace std;

// datos enteros (int)
int min(int a, int b)
{
 if(a <= b)
 return a;
 return b;
}
// datos largos(long int)
```

<sup>3</sup> También se utilizan los términos *plantilla patrón*, o lo que es igual *función patrón* o *clase patrón*.

```

long min(long a, long b)
{
 if(a <= b)
 return a;
 return b;
}
// datos char
char min(char a, char b)
{
 if(a <= b)
 return a;
 return b;
}
// datos double
double min(double a, double b)
{
 if(a <= b)
 return a;
 return b;
}

void main()
{
 int ea = 1, eb = 5;
 cout << "(int) :" << min(ea, eb) << endl;

 long la = 10000, lb = 4000;
 cout << "(long) :" << min(la, lb) << endl;

 char ca = 'a', cb = 'z';
 cout << "(char) :" << min(ca, cb) << endl;

 double da = 423.654, db = 789.10;
 cout << "(double) :" << min(da, db) << endl;
}

```

Al ejecutar el programa se visualiza:

```

(int) : 1
(long) : 4000
(char) : a
(double) : 423.654

```

Obsérvese que las diferentes funciones `min()` tienen un cuerpo idéntico, pero como los argumentos son distintos, se diferencian entre sí en los prototipos. En este caso sencillo se podría evitar esta multiplicidad de funciones definiendo una macro con `#define`:

```
#define min(a, b) ((a) <= (b) ? (a) : (b))
```

Sin embargo, con la macro se perderán los beneficios de las verificaciones de tipos que efectúa C++ para evitar errores. Con el fin de seguir disponiendo de las ventajas de las verificaciones de tipos se requieren las plantillas de funciones.

Las funciones `min` anteriores se escriben igual en todos los casos, pero son funciones totalmente diferentes ya que ellas manejan argumentos y valores de retorno de tipos distintos. Es cierto que en C++ estas funciones están sobrecargadas con el mismo nombre, pero se deben escribir definiciones independientes de cada una de ellas; en el lenguaje C, que no soporta la sobrecarga de funciones para tipos diferentes, no pueden tener el mismo nombre, de modo que en C++ una función `abs` puede estar sobrecargada mientras que la biblioteca C soporta funciones como `abs()`, `fabs()`, `labs()` y `cabs()` para representar diferentes funciones con tipos distintos.

La reescritura del código del cuerpo de la función consume tiempo y malgasta espacio en el código y luego en memoria. También, en caso de encontrar un error en alguna función, se necesitará corregir ese

error en el cuerpo de todas las funciones. La plantilla de funciones permite escribir una función patrón una sola vez y utilizarla en muchos tipos diferentes. La idea se muestra en la figura 19.1.

## Definición de plantilla de funciones

La innovación clave en las plantillas de funciones es representar el tipo de dato utilizado por la función no como un tipo específico como `int`, sino por un nombre que representa a cualquier tipo. Normalmente, este tipo se representa por `T` (aunque puede ser cualquier nombre que decida el programador, como `Tipo`, `UnTipo`, `Complejo` o similar; es decir, cualquier identificador distinto de una palabra reservada).

La sintaxis de una plantilla de funciones tiene dos formatos, según se utilice la palabra reservada `class` o `typename`. Ambos formatos se pueden utilizar: `class` es el formato clásico y que incorpora todos los compiladores y `typename` es el formato introducido por el estándar ANSI/ISO C++ para utilizar en lugar de `class`.

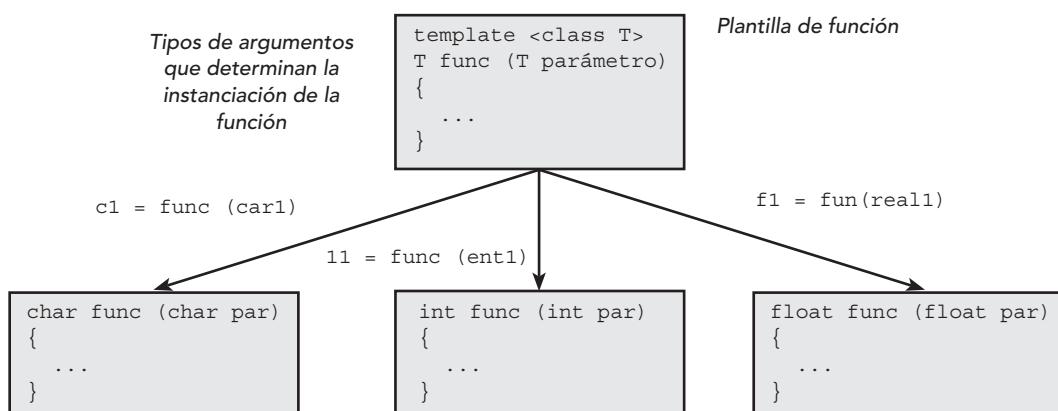


Figura 19.1 Plantilla de función.

### Sintaxis:

1. `template <class T>`
2. `template <typename T>`

`T` es un parámetro tipo, que puede ser remplazado por cualquier tipo, y que también se conoce como un argumento de la plantilla.

Sintácticamente, no hay ninguna diferencia entre las dos palabras reservadas, `class` y `typename`, y se pueden usar, una u otra, indistintamente.

Una definición de plantilla comienza con la palabra reservada `template` seguida por una *lista de parámetros de la plantilla*, que es una lista de uno o más parámetros de plantilla, separados por comas, encerrados entre corchetes tipo ángulo (`< y >`). *La lista de parámetros no puede estar vacía*.

### Reglas prácticas

- La palabra reservada `class` se puede utilizar en lugar de `typename` y tienen el mismo significado en este contexto (una recomendación puede ser utilizar `class` cuando el argumento deba ser una clase y `typename` cuando el argumento pueda ser cualquier tipo).
- La palabra reservada `template` indica al compilador que el código que sigue es una plantilla o patrón de funciones, no la cabecera o definición real de una función.
- Los parámetros de tipo (y los argumentos para la plantillas de clase) aparecen entre corchetes de desigualdad (`< >`).
- Las plantillas de funciones, al contrario que las funciones ordinarias, no se pueden separar en un archivo de cabecera que tenga su cabecera y un archivo compilado por separado que contenga sus definiciones. Las definiciones se deben compilar con las cabeceras que se deben incluir en un programa mediante `#include`.

Las cabeceras de las plantillas de funciones no se pueden guardar en un archivo de cabecera `nombre.h` y las definiciones en un archivo de cabecera `nombre.cpp` que se compilen de modo separado. Una práctica habitual es poner todo en el mismo archivo.

Las plantillas de funciones pueden tener más de un parámetro de tipo.



### Ejemplo 19.2

```
1. //Devolver el valor mayor de dos valores
template <typename T>
const T& max(const T& a, const T& b)
{
 return a > b ? a : b;
}
2. //Devolver el valor mínimo de a y b
template <typename T>
T min(T a, T b)
{return a < b ? a : b;}
```

Para utilizar una plantilla de funciones se debe especificar una instancia de una plantilla, proporcionando argumentos para cada parámetro de la plantilla. Se puede hacer esta tarea, explícitamente, listando argumentos dentro de los corchetes tipo ángulo. Unos ejemplos de una llamada a la función `max` definida antes:

- a. `long x = max <long> (40, 50);`
- b. `int x = max (40, 50);`



### Ejemplo 19.3

Mostrar el funcionamiento completo de una plantilla de función que realiza el intercambio de los valores de dos variables.

```
#include <iostream>
using namespace std;

//definir una plantilla de función, intercambio
template <class T>
void intercambio (T& v1, T& v2)
{
 T aux;

 aux = v1;
 v1 = v2;
 v2 = aux;
}
int main()
{
 int numero1 = 5, numero2 = 8;
 cout << "valores originales: "
 << numero1 << " " << numero2 << endl;
 intercambio (numero1, numero2);
 cout << "valores intercambiados: "
 << numero1 << " " << numero2 << endl;

 //intercambio de caracteres
 char car1 = 'L' , car2 = 'J' ;
 cout << "valores originales: "
 << car1 << " " << car2 << endl;
 intercambio (car1, car2)
 cout << "valores intercambiados: "
 << car1 << " " << car2 << endl;
 return 0;
}
```

Escribir una función para comparar dos valores e indicar si el primer valor es menor, igual o mayor que el segundo.

### Ejemplo 19.4

El sistema de sobrecarga de funciones puede definir varias funciones con el mismo nombre:

```
//devolver 0 si los valores son iguales, -1 si v1 es el más
//pequeño y 1 si v2 es el menor.
int comparar(const string &v1, const string &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
}
//otra función comparar puede ser
int comparar (const double &v1, const double &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
}
```

Estas dos funciones son casi idénticas excepto los tipos de sus parámetros que son diferentes: están *sobrecargadas*. Una versión de plantilla de función de `comparar` es:

```
template <typename T>
int comparar (const T &v1, const T &v2)
{
 if (v1 < v2) return -1;
 if (v2 < v1) return 1;
 return 0;
```

#### Plantilla de función `class (typename)`

Cada parámetro formal consta de la palabra reservada `class` o `typename`, seguida por un identificador. Esta palabra reservada indica al compilador que el parámetro representa un posible tipo incorporado definido por el usuario. Se necesita, al menos, un parámetro `T` que proporcione datos sobre los que pueda operar la función. También se puede especificar un apuntador (`T *` parámetro) o una referencia (`T &` parámetro).

La función puede declararse con un parámetro formal o con múltiples parámetros formales y devolver, inclusive, un valor de tipo `T`. Algunas posibles declaraciones pueden ser:

1. `template <class T> T & f(T parámetro)`  
`{`  
 `// cuerpo de la función`  
`}`
2. `template <class T> T f(int a, T b)`  
`{`  
 `// cuerpo de la función`  
`}`
3. `template <typename T> T f(T a, T b)`  
`{`  
 `// cuerpo de la función`  
`}`

Se pueden declarar también dos parámetros tipo `T`, pero con la condición de que sean distintos.

4. `template <class T1, class T2> T1 f(T1 a, T2 b)`

```

{
 // cuerpo de la función
}

```

Una función plantilla se puede declarar externa (`extern`), en línea (`inline`) o estática (`static`), de igual forma que una función no plantilla. La palabra reservada correspondiente (`extern`, ...) se sitúa a continuación de la línea de parámetros formales ( *nunca delante de la palabra reservada template*).

```

// declaración correcta
template <class T> inline T f(T a, T b)
{
 // cuerpo de la función
}
// declaración incorrecta
extern template <class T> T f(T a, T b)
{
 // cuerpo de la función
}

```



### Ejemplo 19.5

El archivo `minmax.h` declara dos plantillas de funciones: `min()` y `max()`.

```

//Archivo minmax.h
//evitar múltiples #include

#ifndef _INMAX_H
#define _INMAX_H

//plantilla de función max
template <class T> T max(T a, T b)
{
 if(a > b)
 return a;
 else
 return b;
}
// plantilla de función min
template <class T> T min(T a, T b)
{
 if(a < b)
 return a;
 else
 return b;
}
#endif // _INMAX_H

```

Un programa que utiliza funciones plantilla se muestra en el listado `PLANTI.CPP`, donde se incluyen los correspondientes prototipos de funciones.

```

#include <iostream>
using namespace std;

#include "minmax.h"

int max(int a, int b);
double max(double a, double b);
char max(char a, char b);

int main()
{
 int e1 = 100, e2 = 200;
 double d1 = 3.141592, d2 = 2.718283;

```

```

cout << "max(e1, e2) es igual a: " << max(e1, e2) << '\n';
cout << "max(d1, d2) es igual a: " << max(d1, d2) << '\n';
cout << "max(c1, c2) es igual a: " << max(c1, c2) << '\n';
return 0;
}

```

## Un ejemplo de función plantilla

Se puede utilizar la plantilla `intercambio` definida en el ejemplo 19.3 para construir una función plantilla `ordenar`:

```

template <class T> void ordenar(T* v, int n)
{
 for (int intervalo = n/2; intervalo > 0; intervalo /=2)
 for(int i = intervalo; i < n; i++)
 for(int j = i - intervalo; j >= 0 && v[j+intervalo] < v[j]; j -= intervalo)
 intercambio(v[j], v[j+intervalo]);
}
extern int rango_ent[30];
extern string lista_cadena[10];

ordenar(rango_ent, 30); //llama a ordenar(int *, int);
ordenar(lista_cadena, 10); //llama a ordenar (string *, int);

```

## Plantillas de función ordenar y buscar

Las plantillas de funciones se utilizan con mucha frecuencia para ordenar listas o buscar elementos en listas. Por ejemplo, se puede definir una función parametrizada y ordenar cualquier tipo de array o lista, como:

```

template <class T> void ordenar (Array <T>)
{
 // ...
}

```

Este ejemplo declara esencialmente un conjunto de funciones `ordenar` sobrecargadas, una por cada tipo de `Array`. Se puede invocar la función `ordenar` como si fuera cualquier función ordinaria. El compilador C++ analiza los argumentos de la función y llama a la versión adecuada de la función. Por ejemplo, dados dos arreglos, `eArray` y `fArray` de tipos `int` y `float`, respectivamente, se puede aplicar la función `ordenar` para cada uno de los tipos de array:

```

ordenar(eArray); //ordenar el array de enteros
ordenar(fArray); //ordenar el array de float

```

Otra función plantilla que implementa una búsqueda genérica en listas puede tener el siguiente código:

```

template <class T>
unsigned BusquedaBin(T& DatosABuscar, T lista[], unsigned num)
{
 in primero = 0, último = num -1;
 unsigned m;
 do {
 m = (primero + último)/2;
 if (DatosABuscar < lista[m])
 ultimo = m-1;
 else
 primero = m + 1;
 }
}

```

```

 } while (!DatosABuscar == array[m] || primero > ultimo);
 return (DatosABuscar == array[m] ? m: 0xfffff;
}

```

## Una aplicación práctica

Diseñar una plantilla de función para calcular el máximo de dos datos, otra para calcular el máximo de tres y un programa que haga uso de esa plantilla.

```

template <class Tipo>
Tipo max2(Tipo primero, Tipo segundo)
{
 return primero > segundo? primero: segundo;
}
template <class Tipo>
Tipo max3(Tipo primero, Tipo segundo, Tipo tercero)
{
 return max2(max2 (primero, segundo), tercero);
}

```

Un código que utiliza esta función plantilla es:

```

void main()
{
 cout << max2(6, 4) << "\n";
 cout << max3(10, 40, 20) << "\n";
 cout << max3(9.99, 4.45, 3.1416) << "\n";
 cout << max3('M', 'A', 'S',) << "\n";
}

```

Al ejecutarse el programa anterior se visualizará:

```

6
40
9.99
S

```

### Precaución

Suponga que se realizan dos invocaciones:

```

cout << max2 (4, 5.9) << "\n";
cout << max2 ('A', 66) << "\n";

```

Al compilar este segmento de programa se producirán dos errores en tiempo de compilación. Una solución posible para evitar estos errores de compilación es definir una nueva función plantilla, como la siguiente:

```

template <class Tip01, class Tip02>
Tip01 maxd(Tip01 primero, Tip02 segundo)
{
 return primero > (Tip01) segundo ? primero : (Tip01) segundo;
}

```

El resultado será del tipo del primer parámetro, y el siguiente código:

```

void main()
{
 cout << maxd(4, 5.9) << "\n";
 cout << maxd('A', 66) << "\n";
}

```

producirá los resultados siguientes:

```

5
B

```

## Problemas en las funciones plantilla

Cuando se declaran plantillas de funciones con más de un parámetro, para evitar errores es preciso tener mucha precaución.

Así, por ejemplo, si la función se declara con múltiples parámetros y devuelve un valor de tipo *T*:

```
template <class T> T f(int a, T b)
{
 //cuerpo de la función
}
```

Esta versión de la función plantilla devuelve un tipo *T* y tiene dos parámetros: un parámetro *int* llamado *a* y un tipo objeto no especificado *T* llamado *b*. El usuario de la función plantilla ha de proporcionar el tipo de dato para *T*. Por ejemplo, un programa puede especificar el prototipo:

```
double f(int a, double b);
```

Consideremos otro ejemplo, la función *min* definida anteriormente:

```
// archivo MINAUX.CPP
template <class T> min (T a, T b)
{
 if (a < b)
 return a;
 else
 return b;
}
int main()
{
 char c1 = 'J', c2 = '1';
 int n1 = 25, n2 = 65;
 long n3 = 50000;
 float n4 = 84.25, n5 = 9.999;
 min(c1, n1); // error
 min(n3, n4); // error
 min(n2, n3); // error
 min(c1, c2); // correcto
 min(n1, n2); // correcto
 min(n4, n5); // correcto
}
```

La compilación de este programa genera errores debido a la discordancia de tipos en las siguientes líneas:

```
min(c1, n1); // c1 y n1 tipos distintos
min(n3, n4); // n3 y n4 tipos distintos
min(n2, n3); // n2 y n3 tipos distintos
```

## 19.4 Plantillas de clases

Las *plantillas de clase* permiten definir clases genéricas que pueden manipular diferentes tipos de datos. Una aplicación importante es la implementación de *contenedores*, clases que contienen objetos de un tipo dato, como vectores (*arrays*), listas, secuencias ordenadas, tablas de dispersión (*hash*); en esencia, los contenedores manejan estructuras de datos.

Así, es posible utilizar una clase plantilla para crear una pila genérica que se puede instanciar para diversos tipos de datos predefinidos y definidos por el usuario. Puede tener también clases plantillas para colas, vectores (*arrays*), matrices, listas, árboles, tablas (*hash*), grafos y cualquier otra estructura de datos de propósito general. En terminología orientada a objetos, las plantillas de clases se denominan también *clases parametrizables*.

## Definición de una plantilla de clase

La sintaxis de la plantilla de clase es:

```
template <class nombretipo> class tipop {
 //...
};
```

donde *nombretipo* es el nombre del tipo definido por el usuario utilizado por la plantilla, tipo genérico *T*, y *tipop* es el nombre del *tipo parametrizado* para la plantilla (es decir, *tipop* es su *clase genérica*). *T* no está limitado a clases o tipos de datos definidos por el usuario y puede tomar incluso el valor de tipos de datos aritméticos (*int*, *char*, *float*, etc.).

Al igual que en las plantillas de funciones, el código de la plantilla siempre está precedido por una sentencia en la cual se declara *T* como parámetro de tipo (pueden definirse también varios parámetros tipo en las plantillas de clases).

La sintaxis básica de una plantilla de clase es una cabecera de la declaración de la clase seguida por una declaración o definición de clase.

### Sintaxis:

```
template <typename T>
class Pila {
 ...
};
```

Alternativamente

```
template <class T>
class Pila {
 ...
};
```



### Ejemplo 19.6

```
template <typename T>
class Punto {
 T x, y;
};
```

Para utilizar la plantilla de clase, proporcione un argumento para cada parámetro de la plantilla:

```
Punto<int> pt {45,15};
```

De acuerdo con la sintaxis propuesta, la plantilla para una clase genérica *Pila* se puede escribir así:

```
// archivo PILAGEN.H
// interfaz de una plantilla de clases para definir pilas
```

```
template <class T>
class Pila
{
 T datos[50]
 int elementos;
 public:
```

```

Pila(): elementos(0) { }
// añadir un elemento a la pila
void Meter(T elem);
// obtener un elemento de la pila
T sacar();
// número de elementos reales en la pila
int Numero();
// ¿está la pila vacía?
bool vacia();
};

```

El prefijo `template <class T>` en la declaración de clases indica que se declara una plantilla de clase y que se utilizará `T` como el tipo genérico. Por consiguiente, `Pila` es una clase parametrizada con el tipo `T` como parámetro.

Con esta definición de la plantilla de clases `Pila` se pueden crear pilas de diferentes tipos de datos, como:

```

Pila <int> pila_ent; // Una pila para datos int
Pila <float> pila_real; // Una pila para datos float

```

De igual modo, se define una clase genérica `Array` como sigue:

```

template <class T> class Array
{
public :
 Array(int n = 16) {pa = new T [longitud = n];}
 ~Array() {delete[] pa;}
 T& operator[] (int i);
 // ...
private :
 f* pa;
 int longitud;
};

```

Se pueden crear, a continuación, diferentes tipos de arreglos de la siguiente forma:

```

Array <int> intArray(128); // Array de 128 elementos int
Array <float> fArray(32); // Array de 32 elementos float

```

Consideremos la siguiente especificación de la clase plantilla `cola`, que contiene dos parámetros:

```

template <class elem, int tam > class cola {
 int tamaño;
 ...
public;
 cola(int n);
 int vacia();
 ...
};

```

La clase plantilla `cola` tiene dos parámetros plantilla: una variable de tipo `elem`, que especifica el tipo de los elementos de la cola, y `tamaño`, que especifica el tamaño de la cola.

Algunas definiciones de variables que ilustran el uso de la clase plantilla `cola`:

```

cola <int, 2048> a;
cola <char, 512> b;
cola <char, 1024> c;
cola <char, 512*2> d;

```

Dos nombres de clase plantillas se refieren a las mismas clases, solo si los nombres de plantillas son idénticos y sus argumentos tienen valores idénticos. En consecuencia, solo las variables `c` y `d` son del mismo tipo.

La implementación de una clase plantilla requerirá unas funciones constructor, destructor y miembros. Así, una definición de un constructor de plantilla tiene el formato:

```
template <declaraciones-parámetro-plantilla>
 nombre-clase <parámetros-plantilla>::nombre_clase (argumentos)
{
 // ...
}
```

El cuerpo del constructor de la plantilla `cola`:

```
template <class elem, int tam> cola <elem, tam>::cola (argumentos)
{
 ...
}
```

Las definiciones de destructores son similares a las definiciones de los constructores. Una definición de la función miembro vacía de una plantilla de la clase `cola` tiene el formato siguiente:

```
template <declaraciones-parámetros-plantilla> tipo-resultado
 nombre-clase <parámetros-plantilla>::
 nombre-func-miembro (declaraciones-parámetros)
{
 // ...
}
```

Como ejemplo de la sintaxis anterior se puede definir la función vacía de la clase plantilla `cola`:

```
template <class elem, int tam> int cola <elem, tam>::vacía()
{
 // ...
}
```

## Instanciación de una plantilla de clases

Al igual que con las plantillas de funciones, se pueden instanciar las plantillas de clases. Una *clase plantilla* es una clase construida a partir de una plantilla de clases. La plantilla de clases ha de ser instanciada para manipular los objetos del tipo adecuado. Es decir, cuando el compilador encuentra especificado un tipo plantilla, como:

```
cola <int, 2048> a;
```

la primera vez toma los argumentos dados para la plantilla y construye una definición de clase automáticamente (figura 19.2).

## Utilización de una plantilla de clase

La manipulación de una plantilla de clase requiere tres etapas:

- Declaración del tipo parametrizado (por ejemplo `Pila`).
- Implementación de la pila.
- Creación de una instancia concreta de pila (por ejemplo, datos de tipo `entero -int-` o carácter `char`).

Así, por ejemplo, supongamos que se desea crear un tipo parametrizado `Pila` con las funciones miembro `poner` y `quitar`.



Figura 19.2 Instanciación de una plantilla.

### Declaración de la plantilla Pila

```
template <class Tipo>
class Pila {
public:
 Pila();
 bool poner(const Tipo); // meter elemento en la pila
 bool quitar(Tipo&); // sacar elemento de la pila
private :
 Tipo elementos[MaxElementos]; // elementos de pila
 int cima; // cima de la pila
};
```

### Implementación de la pila

```
template <class Tipo>
Pila <Tipo>:: Pila()
{
 cima = -1;
}

template <class Tipo> // función miembro poner
bool Pila <Tipo>:: poner(const Tipo item)
{
 if(cima < MaxElementos -1) {
 elementos[++cima] = item;
 return true;
 }
 else {
 return false;
 }
}

template <class Tipo>
bool Pila <Tipo> :: quitar(Tipo& item)
{
 if (cima < 0) {
 return false;
 }
 else {
 item = elementos[cima--];
 return true;
 }
}
```

### Instanciación de la plantilla de clases

A continuación se instancia una pila de enteros y otra de char:

```
Pila <int> pila_ent; // pila de enteros
Pila <char> pila_car; // pila de caracteres
```

#### Nota

Mediante un tipo parametrizado no se puede utilizar una pila que se componga de tipos diferentes. Posteriormente se verá que objetos polimórficos podrían realizar el efecto de tener tipos diferentes de objetos procesados a la vez, aunque no siempre esto es lo que se requiere.

## Argumentos de plantillas

Los argumentos de plantilla no se restringen a tipos de datos, aunque este sea el uso predominante. Los parámetros de una plantilla pueden ser cadenas de caracteres, nombres de funciones y expresiones de constantes.

Un caso interesante es el uso de una constante entera para definir el “tamaño” de una estructura de datos de tipo genérico. Por ejemplo, el siguiente código declara un vector genérico de  $n$  elementos:

```
template <class T, int n>
class vector
{
 T datos[n];
 // ...
};
```

Este argumento constante puede incluso tener un valor predeterminado, como argumentos normales de funciones. La regla de compatibilidad de tipos entre instancias de argumentos de plantilla permanece igual: dos instancias son compatibles si sus argumentos tipo son iguales y sus argumentos expresión constante tiene el mismo valor. Esta regla significa que las declaraciones siguientes definen dos objetos compatibles:

```
Vector <float, 100> V1,
Vector <float, 25*4> V2;
```

### 19.5 Una plantilla para manejo de pilas de datos

Se trata de diseñar una clase *Pila* que permita manipular pilas de diferentes tipos de datos. Una *pila* es una estructura de datos que permite almacenar datos de modo que el último dato en *entrar* en la pila es el primero en *salir*.

Las operaciones que se consideran son *poner*, *quitar* y *visualizar*.

```
// archivo PILA1.CPP
enum estado_pila {OK, LLENA, VACIA};
template <class T> class Pila
{
public:
 Pila(int _longitud = 10);
 ~Pila() {delete[] tabla;}
 void poner(T);
 T quitar();
 void visualizar();
 int num_elementos();
 int leer_long() {return longitud;}
private:
 int longitud;
 T* tabla;
 int cima;
 estado_pila estado;
};
```

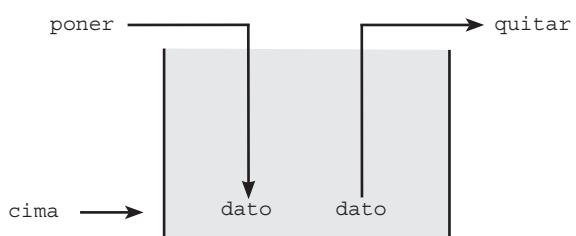


Figura 19.3 Estructura de datos *pila*.

En la declaración anterior se considera el tipo genérico T. Pila es una clase parametrizada por un tipo T y se representa con la notación Pila <T>.

## Definición de las funciones miembro

La declaración de cada función se precede por template <class T> que indica al compilador que la función está parametrizada por el tipo T.

```
template <class T> Pila <T>::Pila(int _longitud)
{
 longitud = _longitud;
 tabla = new T[longitud];
 cima = 0;
 estado = VACIA;
}
template <class T> void Pila <T>::poner(T _elemento)
{
 if (estado!= LLENA)
 tabla [++cima] = _elemento;
 else
 cout << "*** Pila llena ***" << endl;
 if (cima >= longitud)
 estado = LLENA;
 else
 estado = OK;
}
template <class T> T Pila <T> :: guitar()
{
 T elemento = 0;
 if (estado!= VACIA)
 elemento = tabla[--cima];
 else
 cout << "*** Pila vacía ***" << endl;
 if (cima <= 0)
 estado = VACIA;
 else
 estado = OK;
 return elemento;
}
template <class T> void Pila <T>::visualizar()
{
 for (int i = cima; i > 0, i--)
 cout << "[" << tabla[i] << "]" << endl;
}
template <class T> int Pila<T>:: num_elementos()
{
 return cima;
}
```

## Utilización de una clase plantilla

La manipulación de la pila con diversos tipos de datos se puede comprobar con el siguiente programa:

```
void main()
{
 // Pila de enteros
 Pila <int> p1(6);
```

```

p1.poner(6);
p1.poner(12);
p1.poner(18);

cout << "Número de elementos:" << p1.num_elementos() << endl;
p1.visualizar();

cout <<"Quitar 1 :" << p1.quitar() << endl;
cout <<"Quitar 2 :" << p1.quitar() << endl;
cout <<"Quitar 3 :" << p1.quitar() << endl;
cout <<"Número de elementos :" << p1.num_elementos() << endl;
cout <<"Quitar 4 :" << p1.quitar() << endl;

cout << endl ;

// Pila de enteros largos
Pila <long> p2(6);
p2.poner(60000L);
p2.poner(1000000L);
p2.poner(2000000L);

cout <<"Número de elementos:" << p2.num_elementos() << endl;
p2.visualizar();

cout <<"Quitar 1 : " << p2.quitar() << endl;
cout <<"Quitar 2 : " << p2.quitar() << endl;
cout <<"Quitar 3 : " << p2.quitar() << endl;
cout <<"Número de elementos :" << p2.num_elementos() << endl;
cout <<"Quitar 4 : " << p2.quitar() << endl;

Pila <double> p3(6);

p3.poner(6.6);
p3.poner(12.12);
p3.poner(18.18);

cout <<"Número de elementos : " << p3.num_elementos() << endl;
p3.visualizar();

cout << "Quitar 1 : " << p3.quitar() << endl;
cout << "Quitar 2 : " << p3.quitar() << endl;
cout << "Quitar 3 : " << p3.quitar() << endl;
cout << "Número de elementos :" << p3.num_elementos() << endl;
cout << "Quitar 4 : " << p3.quitar() << endl;
}

```

La ejecución de este programa proporciona la siguiente salida:

```

Número de elementos : 3
[18]
[12]
[6]
Quitar 1 : 18
Quitar 2 : 12
Quitar 3 : 6
Número de elementos : 0
*** Pila vacía ***
Quitar 4 : 0

```

```
Número de elementos : 3
[2000000]
[1000000]
[60000]
Guitar 1 : 2000000
Guitar 2 : 1000000
Guitar 3 : 60000
Número de elementos : 0
*** Pila vacía ***
Guitar 4 : 0

Número de elementos : 3
[18.18]
[12.12]
[6.6]
Guitar 1 : 18.18
Guitar 2 : 12.12
Guitar 3 : 6.6
Número de elementos : 0
*** Pila vacía ***
Guitar 4 : 0
```

Otra forma de implementar una pila genérica:

```
// FICHERO. PilaGen.h

#ifndef PILAGEN4_H
#define PILAGEN4_H

template <class Tipo>
class Pila
{
public :
 Pila();
 Pila(const unsigned n);
 ~Pila();
 void Vaciar();
 void Poner(const Tipo & x);
 Tipouitar();
 Tipo Cima() const;
 bool Vacía() const;
 bool Llena() const;
private:
 unsigned max ;
 unsigned cima;
 Tipo * valor;
};

template <class Tipo>
Pila<Tipo>::Pila():
 max(100), cima(0), valor(new Tipo[max]) { }

template <class Tipo>
Pila<Tipo>::Pila(const unsigned n):
 max(n), cima(0), valor(new Tipo[max]) { }

template <class Tipo>
Pila<Tipo>::~Pila()
{
 delete [] valor;
}
```

```

 }
 template <class Tipo>
 void Pila<Tipo>::Vaciar()
 {
 cima = 0;
 }

 template <class Tipo>
 void Pila<Tipo>::Poner(const Tipo & x)
 {
 valor[cima++] = x;
 }

 template <class Tipo>
 Tipo Pila<Tipo>::Quitar()
 {
 return valor[--cima];
 }
 template <class Tipo>
 Tipo Pila<Tipo>::Cima() const
 {
 return valor [cima - 1];
 }
 template <class Tipo>
 bool Pila<Tipo>::Llena() const
 {
 return (cima >= max) ;
 }
 template <class Tipo> bool Pila<Tipo>::Vacia() const
 {
 return cima = 0;
 }
#endif

```

## Instanciación de una clase plantilla con clases

Una clase plantilla se puede instanciar con cualquier tipo de dato; por ejemplo, en la aplicación anterior *Pila* con objetos de *tipo Complejo* o *tipo Cadena*. Así, por ejemplo, suponiendo que se dispone de dos clases, *Complejo* y *Cadena*, que permiten realizar operaciones sobre números complejos y cadenas de caracteres (*strings*). Se instancia fácilmente la plantilla *Pila* para manipular los tipos de datos citados.

```

void main()
{
 // Pila de complejos
 Pila <Complejo> p1(5);

 p1.poner(Complejo(5, -4));
 p1.poner(Complejo(10, -2));
 p1.poner(Complejo(15, -3));

 cout << "Número de elementos : " << p1.num_elementos() << endl;
 p1.visualizar();

 // Pila de Cadenas
 Pila <Cadena> p2(5);

 p2.poner("Prueba primera");
 p2.poner("Prueba segunda");
 p2.poner("Prueba tercera");

```

```

cout << " Número de elementos : " << ps.num_elementos() << endl;
p2.visualizar();
}

```

## Uso de las plantillas de funciones con clases

Las funciones plantilla se pueden utilizar con clases. Como es normal, la clase definirá la operación realizada sobre el objeto en la función plantilla. En el programa siguiente, la función plantilla `min` opera sobre los elementos de una clase numérica.

```

// archivo PLANTI.CPP
// ejemplo de función plantilla utilizada con una clase

template <class T> T min(Ta, Tb)
{
 if (a < b)
 return a;
 else
 return b;
}
class Numero
{
 long num;
public:
 Numero(long n):num(n) { }
 long Valor()
 {
 return num;
 }
 int operator < (Numero n2) // redefinición del operador "menor que"
 {
 return num < n2.Valor();
 }
};

void main()
{
 Numero nn1 = 15;
 Numero nn2 = 25;
 Numero nn3 = min(nn1, nn2);
}

```

## 19.6 Modelos de compilación de plantillas<sup>4</sup>

Cuando el compilador ve una definición de plantilla, no se genera código inmediatamente. El compilador produce instancias específicas de tipos de la plantilla solo cuando ve una llamada de la plantilla: se invoca a una plantilla de función o se define un objeto de una plantilla de clase.

Normalmente, cuando se invoca a una función, el compilador necesita ver solo una declaración de esa función. De modo similar, cuando se define un objeto de un tipo clase, la definición de la clase debe estar disponible, pero las definiciones de las funciones miembro no necesitan estar presentes. Como resultado se ponen las declaraciones de la función y las definiciones de la clase en archivos cabecera y las definiciones de funciones ordinarias y miembros de la clase en archivos fuente.

Las plantillas son diferentes [Lippman, 2005]. Para generar una *instanciación* el compilador debe tener que acceder al código fuente que define la plantilla. Cuando se llama a una plantilla de función o

<sup>4</sup> Lippman, Lajoie y Moo realizan una excelente explicación de programación genérica y plantillas en [Lippman, 2005] que recomendamos al lector. [Lippman, 2005]. C++ Primer, 4a. ed. Addison-Wesley, 2005.

una función miembro de una plantilla de clase, el compilador necesita la definición de la función; se necesita el código fuente que está en los archivos fuente.

C++ estándar define dos modelos para compilación del código de las plantillas [Lippman, 2005]. Ambos modelos estructuran los programas de un modo similar: las definiciones de las clases y las declaraciones de las funciones van en archivos de cabeza y las definiciones de miembros y funciones van en archivos fuente. Los dos modelos difieren en el modo que las definiciones de los archivos fuente se ponen disponibles al compilador. Todos los compiladores soportan el modelo denominado “inclusión” y solo algunos compiladores soportan el modo de “compilación separada”.

### Modelo de compilación de inclusión

En el modelo de inclusión, el compilador debe ver la definición de cualquier plantilla que se utilice. La solución que se adopta es incluir en el archivo de cabecera no solo las declaraciones, sino también las definiciones. Esta estrategia permite mantener la separación de los archivos de cabecera y los archivos de implementación, aunque se incluye una directiva `#include` en el archivo de cabecera para que inserte las definiciones del archivo `.ccp`.



#### Ejemplo 19.7

Escribir los archivos necesarios para implementar una clase. Normalmente, las definiciones se hacen disponibles añadiendo una directiva `#include` a las cabeceras que declaran las plantillas de clases o de función. Esta `#include` lleva los archivos fuente que contienen las definiciones variables.

```
//archivo de cabecera demo.h
#ifndef DEMO_H
#define DEMO_H
template<class T>
int comparar(const T&, const T&);
//otras declaraciones

#include "demo.cc" //definiciones de comparar
#endif

//implementación del archivo demo.cc
template<class T> int comparar(const T &a, const T &b)
{
 if(a < b) return -1;
 if(b < a) return 1;
 return 0;
}
//otras definiciones
```

Este método permite mantener la separación de los archivos de cabecera y los archivos de implementación pero asegura que el compilador verá ambos archivos cuando se compila el código que utiliza la plantilla.

Algunos compiladores que utilizan el modelo de inclusión pueden generar instancias múltiples. Si dos o más archivos fuente compilados por separado utilizan la misma plantilla, estos compiladores generan una instancia para la plantilla en cada archivo. Esto supondrá que una plantilla se puede instanciar más de una vez.

### Modelo de compilación separada

Este modelo de compilación es muy parecido al modelo típico de C++ y permite escribir las declaraciones y funciones en dos archivos (extensiones `.h` y `.cpp`). La única condición es que se debe utilizar la palabra reservada `export` para conseguir la compilación separada de definiciones de plantillas y decla-

raciones de *funciones de plantillas*. Sin embargo, este modelo de compilación no está disponible en la mayoría de los compiladores.

La palabra reservada `export` indica que una definición dada puede ser necesaria para generar instancias en otros archivos. Una plantilla puede ser definida como exportada solamente una vez en un programa. El compilador decide cómo localizar la definición de la plantilla cuando se necesiten generar estas implantaciones. La palabra reservada `export` no necesita aparecer en la declaración de la plantilla. Normalmente, se indica que una plantilla de funciones sea *exportada* como parte de su definición. Se debe incluir la palabra reservada `export` antes de la palabra reservada `template`.

La declaración de la plantilla de función se pone en un archivo de cabecera, pero la declaración no debe especificar `export`.

```
//definición de la plantilla en un archivo compilado por separado
export template<typename T> T suma(T t1, T t2)
```

El uso de `export` en una plantilla de clase es un poco más complicado. Como es normal, la declaración de la clase debe ir en un archivo de inclusión. La declaración de la clase en el archivo de inclusión no utiliza `export`. Se utiliza `export` en el archivo que define las funciones miembro, como se observa a continuación.

```
//cabecera de la plantilla de clase está en el archivo
//de cabecera compartido: Pila.h
template <class T> class Pila {...};
//Archivo pila.cpp declara Pila como exportada
export template <class T> class Pila;
#include "Pila.h"
//definiciones de funciones miembro de Pila
```

### Ejemplo 19.8

Los miembros de una clase exportada se declaran automáticamente como exportados; también se pueden declarar miembros individuales de una plantilla de clase como exportados. En este caso, la palabra `export` no se especifica en la plantilla de clase; solo se especifica en las definiciones de los miembros específicos que se exportan. La definición se debe situar dentro del archivo de cabecera que define la plantilla de clase.

#### Nota

La compilación separada es muy interesante pero no es fácil su implementación. Por otra parte, esta característica solo está implementada en las últimas generaciones de compiladores y pudiera suceder que su propio compilador no la incorpore.

## 19.7 Plantillas frente a polimorfismo

Una pregunta usual tras estudiar las plantillas y el *polimorfismo* es conocer cuáles son las diferencias entre estos conceptos. ¿Se puede sustituir el polimorfismo por una plantilla? ¿Cuál de las dos características es mejor? Para dar una contestación repasemos ambos conceptos.

Una función es polimórfica si uno de sus parámetros puede suponer tipos de datos diferentes, que deben derivarse de los parámetros formales. Cualquier función que tenga un parámetro como apuntador a una clase puede ser una función polimórfica y se puede utilizar con tipos de datos diferentes.

Una función es una función plantilla solo si está precedida por una cláusula `template` apropiada. Esta definición significa que estas funciones se diseñan, definen e implementan pensando en genericidad. Por consiguiente, escribir una función plantilla implica pensar en genericidad, evitando cualquier dependencia en tipos de datos, constantes numéricas, etcétera.

Una función plantilla es solo una plantilla (un patrón) y no una verdadera función. Aunque no necesita ser instanciada, este proceso se hace automáticamente por el compilador para cualquier llamada diferente. Como resultado se dispone de una familia de funciones sobrecargadas, que tienen todos el mismo nombre y parámetros distintos. En otras palabras, la cláusula plantilla es un generador automático de funciones sobrecargadas.

Las funciones polimórficas son funciones que se pueden ejecutar dinámicamente con parámetros de tipos diferentes. Por otra parte, las funciones plantilla son funciones que se pueden compilar estáticamente en versiones distintas para acomodarse con parámetros de tipos de datos diferentes.

Desde el punto de vista de usos prácticos de estas dos construcciones, se puede observar que:

- Las funciones plantillas trabajan también con tipos aritméticos.
- Las funciones polimórficas deben utilizar apuntadores.
- La genericidad polimórfica se limita a jerarquías.
- Las plantillas tienden a generar un código ejecutable grande, dado que se duplican las funciones.

## Resumen

En el mundo real, cuando se define una clase o función, se puede desear poder utilizarla con objetos de tipos diferentes, sin tener que reescribir el código varias veces. Las últimas versiones de C++ incorporan las plantillas (*templates*) que permiten declarar una clase sin especificar el tipo de uno o más miembros datos (esta operación se puede retardar hasta que un objeto de esa clase se define realmente). De modo similar, se puede definir una función sin especificar el tipo de uno o más parámetros hasta que la función se llama.

Para declarar una familia completa de clases o funciones se puede utilizar la cláusula `template`. Con plantillas solo se necesita seleccionar la clase característica de la familia.

Las plantillas proporcionan la implementación de tipos parametrizados o genéricos. La *genericidad* es una construcción muy importante en lenguajes de programación orientados a objetos. Una definición muy acertada se debe a Meyer:

Genericidad es la capacidad de definir módulos parametrizados. Tal módulo se denomina módulo genérico, no es útil directamente; en su lugar, es un patrón de módulos. En la mayoría de los casos, los parámetros representan tipos. Los módulos reales denominados instancias del módulo genérico se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos [Meyer, 88].

El propósito de la genericidad es definir una clase (o una función) sin especificar el tipo de uno o más de sus miembros (parámetros).

Las plantillas permiten especificar un rango de funciones relacionadas (sobrecargadas), denominadas *funciones plantilla*, o un rango de clases relacionadas, denominadas *clases plantilla*.

Todas las definiciones de plantillas de funciones comienzan con la palabra reservada `template`, seguida por una lista de parámetros formales encerrados entre ángulos (`<>`); cada parámetro formal debe estar precedido por la palabra reservada `class` o `typename`. Algunos patrones de sintaxis son:

1. `template<class T>`
2. `template<typename TipoElemento>`
3. `template<class TipoBorde, class TipoReleno>`

Las plantillas de clases proporcionan el medio para describir una clase genéricamente y se instancian con versiones de esta clase genérica de tipo específico. Una plantilla típica tiene el siguiente formato:

```
template<class T>
class Demo
{
 T v;
 ...
public:
 Demo (const T & val):v(val) { }
 ...
};
```

Las definiciones de las clases (*instanciaciones*) se generan cuando se declara un objeto de la clase especificando un tipo específico. Por ejemplo, la declaración

```
Demo<short> ic;
```

hace que el compilador genere una declaración de la clase en el que cada ocurrencia del tipo de parámetro `T` en la plantilla se reemplaza por el tipo real `short` en la declaración de la clase. En este caso, el nombre de la clase es `Demo<short>`.

El objetivo de la genericidad es permitir reutilizar el código comprobado sin tener que copiarlo manualmente. Esta propiedad simplifica la tarea de programación y hace los programas más fiables.

- C++ suministra las plantillas (*templates*) para proporcionar *genericidad* y polimorfismo paramétrico. El mismo código se utiliza con diferentes tipos, donde el tipo es un parámetro del cuerpo del código.

- Una plantilla de función es un mecanismo para generar una nueva función.
- Una plantilla de clases es un mecanismo para la generación de una nueva clase.
- Un parámetro de una plantilla puede ser o bien un tipo o un valor.
- El tipo de los parámetros de la plantilla en una definición de una plantilla de función, se puede utilizar para

especificar el tipo de retorno y los tipos de parámetros de la función generada.

- C++ estándar describe una biblioteca estándar de plantilla que, en parte, incluye versiones de plantillas sobre tareas típicas de computación como búsqueda y ordenación.



## Ejercicios

- 19.1 Definir plantillas de funciones `min()` y `max()` que calculen el valor mínimo y máximo de dos valores.
- 19.2 Realizar un programa que utilice las funciones plantilla del ejercicio anterior para calcular los valores máximos de parejas de enteros, de doble precisión (`double`) y de carácter (`char`).
- 19.3 Escribir una clase plantilla que pueda almacenar una pequeña base de datos de registros.
- 19.4 Realizar un programa que utilice la plantilla del ejercicio anterior para crear un objeto de la clase de base de datos.
- 19.5 Escribir una plantilla de función `tintercambio` que intercambia dos variables de cualquier tipo.
- 19.6 Definir una clase plantilla para pilas.
- 19.7 ¿Cómo se implementan funciones genéricas en C++? Compárela con plantillas de funciones.
- 19.8 Declarar una plantilla para la función `gsort` para ordenar arrays de un tipo dado.
- 19.9 Definir una función plantilla que devuelva el valor absoluto de cualquier tipo de dato incorporado o predefinido pasado a ella.
- 19.10 Definir una función plantilla `max()` que trabaje con tipos de datos predefinidos.



# Excepciones

## Contenido

- 20.1 Condiciones de error en programas
- 20.2 El tratamiento de los códigos de error
- 20.3 Manejo de excepciones en C++
- 20.4 El mecanismo de manejo de excepciones
- 20.5 El modelo de manejo de excepciones

## 20.6 Especificación de excepciones

- 20.7 Excepciones imprevistas
- 20.8 Aplicaciones prácticas de manejo de excepciones
  - › Resumen
  - › Ejercicios

## Introducción

Uno de los problemas más importantes en el desarrollo de software es la gestión de condiciones de error. No importa lo bueno que sea el software y la calidad del mismo, siempre aparecerán errores por múltiples razones (errores de programación, errores imprevistos de los sistemas operativos, agotamiento de recursos, etc.). *Excepciones* son, normalmente, condiciones de errores imprevistos. Estas condiciones suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos típicos son: agotamiento de memoria, errores de rango en intervalos, división por cero, etc. El rango y definición de estos errores, así como el modo en que se manejan los errores, pueden ser definidos por el programador. El manejo de excepciones es el mecanismo previsto por C++ para el tratamiento de excepciones. Normalmente el sistema aborta la ejecución del programa cuando se produce una excepción y C++ permite al programador intentar la recuperación de estas condiciones y continuar la ejecución del programa.

## Conceptos clave

- › Captura de excepciones
- › `catch`
- › El bloque `try`
- › Especificación de excepciones
- › Excepción
- › Lanzamiento de una excepción
- › Levantar una excepción
- › Manejador de excepciones
- › Manejo de excepciones
- › `terminate ( )`
- › `throw`
- › `unexpected`

## 20.1 Condiciones de error en programas

La escritura de código fuente y el diseño correcto de clases y funciones es una tarea difícil y delicada, por ello es necesario manejar los errores que se produzcan con la mayor eficiencia. La mayoría de los diseñadores y programadores se enfrentan a dos importantes problemas en el manejo de errores:

1. ¿Qué tipo de problemas se pueden esperar cuando los clientes hacen mal uso de clases, funciones y programas en general?
2. ¿Qué acciones se deben tomar una vez que se detectan estos problemas?

El manejo de errores es una etapa importante en el diseño de programas ya que no siempre se puede asegurar que las aplicaciones utilizarán objetos o llamadas a funciones correctamente. En lugar de añadir conceptos aislados de manejo de errores al código del programa, es deseable construir un mecanismo de manejo de errores como una parte integral del proceso de diseño.

Para ayudar a la *portabilidad* y *diseño* de bibliotecas, C++ incluye un *mecanismo de manejo de excepciones* que está soportado por el lenguaje. En este capítulo se trata de explorar el *manejo de excepciones* y cómo utilizarlo en su diseño. Para ello se examinarán: detección de errores, manejo de errores, gestión de recursos y especificaciones de excepciones.

### ¿Por qué considerar las condiciones de error?

La *captura* (*catch*) o *localización* de errores inadecuados ha sido siempre un problema en el software. Parte del problema es que la captura de errores es una labor muy importante en el trabajo de un programador. También, otros factores a tener en cuenta son: ¿dónde deben ser capturados los errores? o ¿cómo pueden ser manejados? Una vez que se encuentra un error, un programa debe tener varias opciones: terminar inmediatamente; ignorar el error con la esperanza de que no suceda nada “desastroso”; o bien puede establecer un indicador o señal de error, el cual (presumiblemente) se comprobará por otras sentencias del programa. En esta última opción, cada vez que se llama una función específica, el llamador debe comprobar el valor de retorno de la función, y si se detecta un error, se debe determinar un medio de recuperar el error o de terminar el programa.

En la práctica, los programadores no son consistentes en estas tareas, debido en parte a que exige una gran cantidad de trabajo y también debido a que las sentencias de verificación de errores a veces oscurecen la comprensión del resto del código. También es difícil recordar cómo capturar (“atrapar”) cada condición posible de error cada vez que se llama una función determinada. Con frecuencia, el programador debe hacer esfuerzos excepcionales para invocar a los destructores, liberar memoria y cerrar archivos de datos antes de detener un programa.

La solución a tales problemas en C++ es llamar a mecanismos del lenguaje que soportan manejo de errores que eviten la realización de códigos de manejo de errores complejos y artificiales en cada programa. En C++ cuando se genera una *excepción*, el error no se puede ignorar o el programa terminará. Si el código de tratamiento del error está en lugar de un tipo de error específico, el programa tiene la opción de recuperación del error y continuar la ejecución. Este enfoque ayuda a asegurar que no se deslice ningún error que produzca consecuencias fatales y origine que un programa se comporte erráticamente.

Un programa “lanza” (*throws*) una excepción en el punto en que primero se detecta el error. Cuando esto sucede, un programa C++ busca automáticamente en un bloque de código llamado *manejador de excepciones*, que responde a la excepción de un modo apropiado. Esta respuesta se llama “capturar o atrapar una excepción” (*catching an exception*). Si no se puede encontrar un *manejador de excepciones*, el programa, simplemente, termina.

## 20.2 El tratamiento de los códigos de error

Los desarrolladores de software han estado utilizando, durante mucho tiempo, códigos para indicar condiciones de error. Normalmente, los procedimientos devuelven un código para indicar sus resultados. Por ejemplo, un procedimiento *AbrirArchivo* puede devolver 0 (cero) para indicar un fallo. Otros procedimientos pueden devolver (-1) para indicar un fallo y (0) para indicar éxito. Los sistemas operativos y las bibliotecas de software documentan todos los códigos de error posibles que pueden ser devueltos por un procedimiento. Los programadores que utilizan tales procedimientos deben comprobar cuidadosamente el código devuelto y las cosas a realizar en función de los resultados. Esto puede producir más serios problemas en el código posterior. Cuando una aplicación termina anormalmente puede dejar recursos inoperativos como: archivos que se abrieron no se cierran, conexiones de redes no se cierran, datos no se escriben en disco. *Una aplicación bien diseñada e implementada no debe permitir que esto suceda.*

*Los errores no se deben propagar innecesariamente.* Si los errores se detectan y se manejan tan pronto como suceden, se evitan daños mayores en código posterior. Cuando los errores se propagan a diferentes

partes del código, será mucho más difícil trazar la causa real del problema debido a que los síntomas del problema pueden no indicar la causa real.

Cuando se detecta un error en una aplicación, se debe poder fijar la causa del problema y volver a intentar (procesar) la operación que produjo el error. Por ejemplo, si se ha producido un error porque el índice de un arreglo se había fijado más alto que el final de dicho arreglo, es fácil localizar la causa de error. En otras situaciones, la aplicación puede no ser capaz de fijar la condición de error. Una salida elegante puede ser la única salida en tales situaciones. El programador debe tener la libertad de tomar la decisión correcta.

Cuando un procedimiento encuentra una condición de error y vuelve al llamador, se debe esperar que el procedimiento ejecute las operaciones de “limpieza” necesarias antes de retornar al llamador. El código que detecta y maneja la condición de error debe incorporar código extra para esta operación de limpieza. En otras palabras, no es una tarea fácil y es más difícil cuando hay múltiples puntos de salida del procedimiento. Si existen múltiples posiciones en el código en las que se deben verificar las condiciones de error, el procedimiento se volverá complicado y enrevesado.

En ciertos casos, un procedimiento puede no tener información suficiente para manejar una condición de error; en estos casos puede ser más seguro propagar las condiciones de error a un procedimiento exterior en el que se pueda manejar. Los procedimientos que devuelven códigos de error sencillos pueden no ser capaces de cumplir estos retos.

Los códigos de error devueltos de procedimientos no transmiten mucha información al procedimiento llamador. Es normalmente un número que indica la causa del fallo. Sin embargo, en muchas situaciones puede ser muy útil si existe más información disponible sobre causa del fallo en el llamador. Esto ayudará a fijar la condición de error (si es posible). Un código de error sencillo no puede cumplir este objetivo.

En resumen, las alternativas típicas para el *manejo de errores en programas* son:

1. Terminar el programa.
2. Devolver un valor que representa un error.
3. Devolver un valor legal, pero establecer un indicador (señal) de error global.
4. Llamar a una función de error proporcionada por el usuario.

Cualquiera de los métodos tiene inconvenientes y deficiencias, en ocasiones graves. Por esta causa es necesario buscar nuevos métodos o esquemas. Uno de los esquemas más populares que ha sido comprobado y está soportado en muchos lenguajes (C++, Ada, Java...) es el principio de “levantamiento” o “alzamiento” (*raising*) de una excepción. En este principio se fundamenta el mecanismo de manejo de excepciones de C++ que ya se ha citado anteriormente y que trataremos ahora.

### 20.3 Manejo de excepciones en C++

Una excepción se “levanta” (*raise*) en caso de un error e indica una condición anormal que no se debe encontrar durante la ejecución normal de código. Una excepción indica una necesidad urgente de tomar una acción reparadora (de remedio).

La palabra *excepción* indica aquí una excepción software. No se debe confundir con una excepción hardware. Una excepción software se produce por una parte de código escrita por un programador. No tiene nada que ver con las excepciones hardware. Una excepción software se inicia por alguna parte del código que encuentra una condición anormal.

Una excepción es un error de programa que ocurre durante la ejecución. Si ocurre una excepción y está activo un segmento de código denominado *manejador de excepción* para esa excepción, entonces el flujo de control se transfiere al manejador. Si en lugar de ello ocurre una excepción y no existe un manejador para la excepción, el programa termina.

Una excepción se puede levantar cuando “el contrato” entre el *llamador* y el *llamado* se violan. Por ejemplo, si se intenta acceder a un elemento que está fuera del rango válido de un arreglo se produce una violación del contrato entre la función que controla los índices (operator `[ ]` en C++) y el llamador que utiliza el arreglo. La función de índices garantiza que devuelve el elemento de la función especificada si el índice que se le ha pasado es válido. Pero si el índice no es válido, la función de índice debe indicar la condición de error. Siempre que se produzca tal violación del contrato se debe levantar (*azar*) una excepción.

Una vez que se levanta una excepción, esta no desaparece aunque el programador lo ignore (una excepción no se puede ignorar ni suprimir). Una condición de excepción se debe *reconocer y manejar*.

Una excepción no manejada se propagará dinámicamente hasta alcanzar el nivel más alto de la función (`main` en C++). Si también falla el nivel de función más alto, la aplicación termina sin opción posible.

Los lenguajes que admiten el mecanismo de excepciones tienden a mantener el código de excepciones bastante independiente del código normal. Ello se debe a que el mecanismo de manejo de errores, no importa lo bueno que sea, es inútil si se degrada el rendimiento (prestaciones) del código normal.

### Precaución

El manejo de errores usando excepciones no evita errores, solo permite la detección y posible recuperación de los mismos. Evitar errores se puede conseguir utilizando aserciones (`assert`).

En general, el mecanismo de excepciones en C++ (y en la mayoría de los lenguajes) permite:

1. Detección de errores energética y posible recuperación.
2. Limpieza y salida elegante en caso de errores no manejados.
3. Propagación sistemática de errores en una cadena de llamadas dinámicas.

### Inconvenientes

La inclusión de un mecanismo de gestión y manejo de excepciones al código normal para manejar errores no es gratuito. Se debe añadir nuevo código a los procedimientos que gestionan las excepciones. El código de gestión de excepciones entremezclado con el código normal, oscurece la lógica del procedimiento original. Por último, añadir excepciones a una aplicación incrementa también el tamaño del código ejecutable.

## 20.4 El mecanismo de manejo de excepciones

En C++, una excepción es un objeto; es decir, un valor, de un tipo fundamental incorporado en el lenguaje o de un tipo definido por el usuario (normalmente una clase). En la práctica, una excepción es un objeto que se pasa desde el área de código donde surge un problema al área de código que va a manipular el programa. Cuando ocurre una excepción se dice que se *lanza* (en inglés, *throw*) en una función donde se detecta un error o un problema, y cuando se manipula una excepción se dice que se ha capturado (*caught*, en inglés) en otra función donde se puede tratar o procesar (en inglés, *try*) el suceso. Si no se captura la excepción se tiene previsto un comportamiento bien definido.

En realidad el manejo de excepciones permite que partes de un programa desarrolladas independientemente se comuniquen para manejar problemas que surgen durante la ejecución del programa. Una parte de un programa puede detectar un problema que esa parte del programa no puede resolver. Es decir, la parte que detecta el problema puede pasar el problema junto a otra parte que se prepara para manejar aquello que está mal.

En esencia, *las excepciones nos permiten separar la detección del problema de la resolución del problema. La parte del programa que detecta un problema no necesita conocer cómo se debe tratar dicho problema.*

El manejo de errores y otros comportamientos anómalos pueden ser una de las partes más difíciles del diseño de cualquier sistema. Los sistemas interactivos más usuales, hoy día, en el mundo de las comunicaciones como los conmutadores (*switches*) y los encaminadores (*routers*) pueden dedicar hasta 90% de su código a detección y manipulación de errores. Con la proliferación de aplicaciones basadas en web que se ejecutan indefinidamente, la tarea de manipulación de errores se ha vuelto cada vez más importante para la mayoría de los programadores.

Las excepciones son anomalías en tiempo de ejecución como agotamiento de la memoria o encontrarse con una entrada no prevista. Las excepciones existen fuera del funcionamiento normal de un programa y requiere de manipulación inmediata por parte del programa. En sistemas bien diseñados, las excepciones representan un subconjunto de la manipulación de errores. *La parte del programa que detecta el problema necesita un medio para transferir el control a la parte del programa que puede manipular el problema.* La parte de detección necesita también poder indicar qué tipo de problema ha ocurrido y debe tener que proporcionar información adicional.

Las excepciones admiten el tipo de comunicación entre las partes de un programa que detectan el error y las partes que manipulan el error. Las excepciones se procesan de acuerdo con los siguientes conceptos:

- Si ocurre una situación anormal en una función, ésta se comunica con el *llamador* con una sentencia especial. Se cambia entonces del flujo normal de datos al manejo de excepciones. Esta manipulación de excepciones deja todos los bloques o funciones llamados hasta que se encuentra un bloque en el que se puede manipular una excepción.
- Cuando se ejecutan las sentencias, podemos definir lo que sucede si una situación de excepción ocurre. Un bloque individual de sentencias puede entonces determinar cómo se trata esa situación.

## Clases de excepciones

C++ adopta un enfoque orientado a objetos para manipular excepciones:

- Las excepciones se consideran como objetos. Si ocurre una excepción, se crea un objeto correspondiente que describe la excepción.
- Las clases de excepción definen las propiedades de este objeto. Las propiedades de una excepción se pueden definir como atributos (por ejemplo, un índice incorrecto o un mensaje explicatorio). Las operaciones se pueden utilizar para consultar información fuera de una excepción. Para tipos diferentes de excepciones existe también la capacidad de proporcionar clases correspondientes diferentes. Cada clase de excepción describe un tipo particular de excepción.

Cuando ocurre una excepción, se crea un objeto de la correspondiente clase de excepción. Al igual que cualquier otro objeto, estos poseen miembros que describen las excepciones y las condiciones en las que ocurren. Ellos son, por consiguiente, parámetros de la excepción.

Se pueden formar jerarquías de clases de excepción utilizando el mecanismo de la herencia. Por ejemplo, los errores matemáticos se pueden dividir en clases especiales de “división por cero”, “raíz cuadrada de números negativos”, etc.; de este modo, un programa de aplicación puede manejar errores matemáticos, en general, o divisiones por cero, en particular.

## Partes de la manipulación de excepciones

En C++, la manipulación de excepciones implica:

- *expresiones throw*. Son utilizadas por la parte de detección de errores para indicar que se ha encontrado un error que no se puede manipular. Se dice que una sentencia *throw* levanta (*raise*) una condición excepcional.
- *throw* se utiliza para “lanzar” un objeto excepción en un área de programa; de este modo se cambia desde el flujo de datos normal al manejo de excepciones.
- *bloques try*, que utilizan la parte de manejo de errores para tratar una excepción. Un bloque *try* comienza con la palabra reservada *try* y termina con una o más cláusulas *catch*. El bloque *try* indica un ámbito en el cual se interceptan las excepciones y se manipulan utilizando *catch*. Las excepciones lanzadas desde el código ejecutado en el interior de un bloque *try* se manipulan normalmente por una de las cláusulas *catch*. Puesto que ellas “manipulan” la excepción, las cláusulas *catch* se conocen como *manejadores* (*handlers*).
- *catch* se utiliza para “capturar” un objeto excepción y para reaccionar a la situación de excepción. Se utiliza para implementar lo que sucede cuando el flujo de datos no funciona.
- Un conjunto de **clases de excepciones** definidas en la biblioteca que son utilizadas para parar la información acerca de un error entre un *throw* y una *catch* asociada.

### 20.5 El modelo de manejo de excepciones

El modelo de un mecanismo de excepciones consta, fundamentalmente, de tres nuevas palabras reservada *try*, *throw* y *catch*.

- *try*, un bloque para detectar excepciones;
- *catch*, un manejador para capturar excepciones de los bloques *try*;
- *throw*, una expresión para levantar (*raise*) excepciones.

Los pasos del modelo son:

1. Primero, un programador “intentará” (*try*) una operación para anticipar errores.
2. Cuando un procedimiento encuentra un error, se “lanza” (*throw*) una excepción. El lanzamiento (*throwing*) de una excepción es el acto de *levantar una excepción*.
3. Por último, alguien interesado en una condición de error (para limpieza y/o recuperación) anticipará el error y “capturará” (*catch*) la excepción que se ha lanzado.

El mecanismo de excepciones se completa con:

- Una función *terminate* que atrapa las excepciones no capturadas.
- Especificaciones de excepciones que dictamina cuáles excepciones, si existen, puede lanzar una función.
- Una función *unexpected* que atrapa las violaciones de especificaciones de excepciones.

## El modelo de manejo de excepciones

La filosofía que subyace en el modelo de manejo de excepciones es simple. El código que trata con un problema no es el mismo código que lo detecta. Este tipo de separación construye un cortafuegos (*firewall*), similar a los establecidos en internet, entre aplicaciones y bibliotecas de clases (figura 20.1).

Cuando una excepción se encuentra en un programa C++, la parte del programa que detecta la excepción puede comunicar que la expresión ha ocurrido levantando, o lanzando (*throwing*) una excepción.

De hecho, cuando el código de usuario llama a una función incorrectamente o utiliza un objeto de una clase de manera inadecuada, la biblioteca de la clase crea un objeto excepción que contiene información sobre lo que era incorrecto. La biblioteca levanta (*raise*) una excepción, una acción que hace el objeto excepción disponible al código de usuario a través de un *manejador de excepciones*. El código de usuario que maneja la excepción puede decidir qué hacer con el objeto excepción. Este enfoque ofrece diversas ventajas. Los programas se hacen más legibles, dado que el código de manejo de errores es independiente del código que detecta los errores. Un estilo regular de manejo de errores es consistente a través de bibliotecas diferentes. Los usuarios han de tener también una regla acerca de qué hacer con las excepciones.

El cortafuegos que actúa de puente entre una biblioteca de clases y una aplicación debe hacer varias cosas para gestionar debidamente el flujo de excepciones, reservando y liberando memoria de modo dinámico.

Una de las razones más significativas para utilizar excepciones es que las aplicaciones no pueden ignorarlas. Cuando el mecanismo de excepciones levanta una excepción, alguien debe tratarla. En caso contrario, la excepción es “no capturada” (*uncaught*) y el programa termina de manera predeterminada. Este poderoso concepto es el corazón del manejo de excepciones y fuerza a las aplicaciones a manejar excepciones en lugar de ignorarlas.

El mecanismo de excepciones de C++ sigue un *modelo de terminación*. Esto implica que nunca vuelve al punto en que se levanta una excepción. Las excepciones no son como manejadores de interrupciones que bifurcan a una rutina de servicio antes de volver al *spot* de interrupción. Esta técnica (llamada *resumption*) tiene un alto tiempo suplementario y es propensa a bucles infinitos y es más complicado de

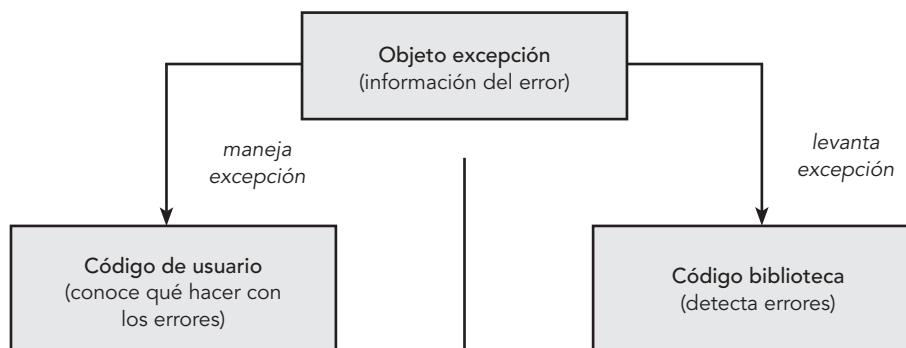


Figura 20.1 Manejo de excepciones construye un “cortafuegos”.

implementar que terminación. El modelo de C++ está diseñado para manejar solo excepciones síncronas con un solo hilo de control, el mecanismo de excepciones implementa un camino alternativo de una sola vía en diferentes sitios de su programa.



### Ejemplo 20.1

```
void f()
{
 // código que produce una excepción que se lanza
 // ...
 throw i;
 //...
}
main()
{
 try {
 f(); //Llamada a f, preparada para cualquier error
 // Código normal aquí
 }
 catch(...)
 {
 // capturar cualquier excepción lanzada por f()
 // hacer algo
 }
 // Resto código normal de main()
}
```

`f( )` es un simple procedimiento. Cuando se encuentra una condición de error se lanza una excepción. Esto se consigue mediante la sentencia

```
throw i;
```

El operando de la expresión `throw` es un objeto. Normalmente se lanzan objetos con información sobre el error.

En el programa `main( )`, la llamada a `f( )` se encierra en un bloque `try`. El código es un bloque `try`

```
try {
 // ...
}
```

En otras palabras, es un bloque de código encerrado dentro de una sentencia `try`. Un bloque `try` indica al compilador la posibilidad de una excepción.

Un bloque `catch( )` captura una excepción del tipo indicado. En el ejemplo anterior

```
catch(...)
```

indica que captura excepciones de todo tipo. Los puntos suspensivos (...) significan cualquier argumento. Una expresión `catch` es comparable a un procedimiento con un argumento.

El código C++ puede levantar una excepción en un bloque `try` utilizando la expresión `throw`. La excepción se maneja invocando un manejador apropiado seleccionado de una lista de manejadores que se encuentran al final del bloque `try`.

## Diseño de excepciones

La palabra reservada `try` designa un bloque, que es un área de su programa que detecta excepciones. En el interior del bloqueo `try`, normalmente se llama a funciones que pueden levantar o *lanzar* excepciones. La palabra reservada `catch` designa un manejador de capturas con una firma que representa un tipo de excepción. Los manejadores de captura siguen inmediatamente a bloques `try` o a otro manejador `catch` con una firma diferente.

Se debe establecer un bloque `try` y un manejador de capturas (`catch`) para capturar excepciones lanzadas. En caso contrario, la excepción no es capturada y su programa termina en forma predeterminada. Los bloques `try` son importantes ya que sus manejadores de captura asociados determinan cuál es la parte de su programa que maneja una excepción específica. El código que está dentro del manejador de capturas (`catch`) es donde se decide lo que se hace con la excepción lanzada.

## Bloques `try`

Un *bloque try* debe encerrar las sentencias que pueden lanzar excepciones y, comienza con la palabra reservada `try` seguida por una secuencia de sentencias de programa encerradas entre llaves. A continuación del bloque `try` hay una lista de manejadores llamados *cláusulas catch*. Al menos un manejador `catch` debe aparecer inmediatamente después de un bloque `try` para manejar excepciones lanzadas; en caso contrario, el compilador genera errores. Cuando un tipo de excepción lanzada coincide con la firma de un manejador `catch`, el control se reanuda dentro del bloque del manejador `catch`. Si ninguna excepción se lanza desde un bloque `try`, el control “salta” todos los manejadores `catch` y prosigue en la sentencia siguiente.

La sintaxis del bloque `try` es:

```
1 try {
 código del bloque try
}
catch (signatura) {
 código del bloque catch
}
```

```
2 try
 sentencia compuesta
 lista de manejadores
```

También se puede anidar bloques `try`.

```
void sub(int n)
{
 try {
 ...
 try {
 ...
 if (n==1)
 return ;
 }
 catch (signatura1) {...} // manejador catch interno
 }
 catch (signatura2) {...} // manejador catch externo
 ...
}
```

Una excepción lanzada en el bloque interior `try` ejecuta el manejador `catch` con *signatura1* si coincide el tipo de excepción. El manejador `catch` con *signatura2* maneja excepciones lanzadas desde el bloque `try` exterior si el tipo de la excepción coincide. El manejador externo de `catch` también captura excepciones lanzadas desde el bloque interior `try` si el tipo de excepción coincide con *signatura2* pero no con *signatura1*. Si los tipos de excepción no coinciden con ninguna firma, la excepción se propaga al llamador de `sub ( )`.

## Normas

```
try {
 sentencias
}
catch (parámetro) {
 sentencias
}
catch (parámetros) {
 sentencias
}
etc.
```

1. Cuando una excepción se produce en sentencias dentro de `try`, hay un salto al primer manejador (la parte `catch`) cuyo parámetro coincide con el tipo de excepción.
2. Cuando las sentencias en el manejador se han ejecutado, se termina el bloque `try` y la ejecución prosigue en la sentencia siguiente. No se produce nunca un salto hacia atrás al lugar en que ocurrió la interrupción.
3. Si no hay manejadores para tratar con una excepción, se aborta el bloque `try` y la excepción se propaga.

El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada. El orden en que los manejadores se definen, determina el orden de llamada. Una excepción solo puede lanzarse después de que la ejecución de un programa se ha introducido en un bloque `try`.

### Precaución

Se puede transformar el control fuera de bloques `try` con una sentencia `goto`, `return`, `break` o `continue`.

Los bloques `try` sirven para muchos propósitos útiles.



### Ejemplo 20.2

La función `calcu_media()` calcula una media de arreglos de tipo `double` con una función plantilla `avg()` (cálculo de la media aritmética). En el caso de ser llamada incorrectamente, `avg()` lanza excepciones de cadena de caracteres.

```
double calcu_media(unsigned int long) {
 const unsigned int max = 6;
 double b[max] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
 if (lon > max) {
 cerr << "calcu_media: uso de longitud por defecto de"
 << max << endl;
 lon = max;
 }
 try {
 return avg(b, lon); // cálculo media
 }
 catch (char *msg) {
 cerr << msg << endl;
 cerr << calcu_media: uso de longitud por defecto de"
 << max << endl;
 return avg(b, max);
 }
}
```

La función declara un arreglo de 6 tipos `double` y llama a `avg()` con el nombre del arreglo (`b`) y un argumento de longitud (`lon`). Un bloque `try` circunda a `avg()` para capturar excepciones de cadena de caracteres. Obsérvese que `calcu_media` asegura que `lon` no es nunca mayor que `max`.

Si `avg` lanza una excepción de cadena de caracteres, el manejador de `catch` recupera volviendo a llamar `avg()` con un valor por defecto (la longitud del arreglo `b`). Esta técnica permite a `calcu_media` llamar a `avg()` una segunda vez con un valor correcto.

## Lanzamiento de excepciones

La sentencia `throw` levanta una excepción. Cuando se encuentra una excepción en un programa C++, la parte del programa que detecta la excepción puede comunicar que la excepción ha ocurrido por levantamiento o *lanzamiento* de una excepción. El formato de `throw` es:

1. `throw expresión`
2. `throw`

El bloque más interno de `try` en el que se levanta una excepción se utiliza para seleccionar la sentencia `catch` que procesa la excepción. La sentencia `throw` sin argumento se puede utilizar dentro de un `catch` para *relanzar* la excepción actual. Normalmente se utiliza cuando se desea que un segundo manejador sea llamado desde el primer manejador para procesar posteriormente la excepción.

La función `demo` levanta una excepción.

```
void demo()
{
 int i;
 // lanzamiento de una excepción
 i = -16;
 throw i;
}
int main()
{
 try {
 demo()
 }
 catch(int n)
 { cerr << "excepción capturada \n" << n << endl; }
}
```

### Ejemplo 20.3

El valor entero lanzado por `throw` persiste hasta que sale del manejador `catch (int n)`. Este valor está disponible para usar con el manejador como su argumento.

## Captura de una excepción: `catch`

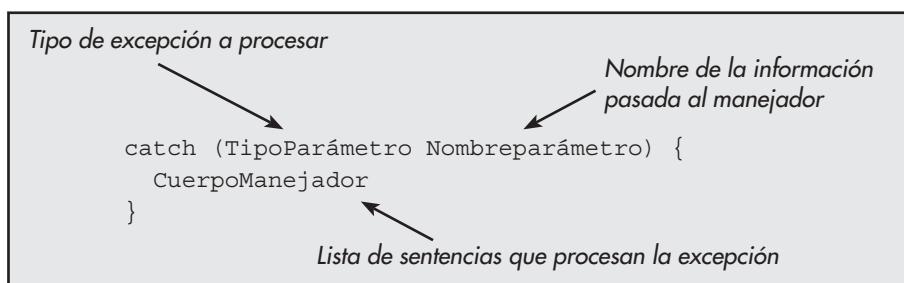
Un manejador de excepciones C++ es una cláusula de captura (`catch`). Cuando una excepción se lanza desde sentencias dentro de un bloque `try`, la lista de cláusulas `catch` que siguen al bloque `try` se buscan para encontrar una cláusula `catch` que pueda manejar la excepción.

Una cláusula `catch` consta de tres partes: la palabra reservada `catch`, la declaración de un solo tipo o un simple objeto dentro de paréntesis (referenciado como *declaración de excepciones*), y un conjunto de sentencias dentro de una sentencia compuesta. Si la cláusula `catch` se selecciona para manejar una excepción, se ejecuta la sentencia compuesta.

### Formatos

1. `catch (signatura) {`  
    *código del bloque catch*  
    `}`      captura `signatura` correspondiente excepción
2. `catch (...) {`  
    *código del bloque catch*  
    `}`      captura cualquier excepción

La especificación 1 del manejador `catch` se asemeja a una definición de función.



El manejador de excepciones consta de la palabra reservada `catch` y de las sentencias que le siguen (*código del bloque o cuerpo manejador*). Al contrario que el bloque `try`, el bloque `catch` solo se ejecuta bajo circunstancias especiales. La *signatura* es una declaración de argumentos. Puede constar de un tipo, un tipo seguido por un nombre de argumento o una sintaxis especial:

```
catch (tipo)
catch (tipo nombre_argumento)
catch (...)
```

Cada una de las dos primeras sentencias crea un manejador de excepciones que busca tipos coincidentes. Al igual que con funciones sobrecargadas, el manejador de excepciones se activa solo si el argumento que se pasa (o se *lanza*, “thrown”) se corresponde con la declaración del argumento.

La tercera sentencia es una sintaxis especial que significa “cualquier excepción”. Se puede utilizar esta sintaxis para escribir manejadores de excepciones de manera predeterminada que capturan todas las excepciones no capturadas ya.

La sintaxis completa de `try` y `catch` permite escribir cualquier número de manejadores de excepciones al mismo nivel.

```
try {
 sentencias
}
catch (parámetro1) {
 sentencias
}
catch (parámetro2) {
 sentencias
}
...
```

Los puntos suspensivos (...) significan que se puede tener cualquier número de manejadores `catch` a continuación del bloque `try`.

### Funcionamiento

Cuando ocurre (se lanza) una excepción en una sentencia durante la ejecución del bloque `try`, el programa comprueba, por orden, cada bloque `catch` hasta que encuentra un manejador (la parte `catch`) cuyo parámetro coincide con el tipo de excepción. Tan pronto como se encuentra una coincidencia, se ejecutan las sentencias del bloque `catch`; cuando se han ejecutado las sentencias del manejador, se termina el bloque `try` y prosigue la ejecución con la siguiente sentencia. Es decir, si el programa no ha terminado, la ejecución se reanuda normalmente después del final de todos los bloques `catch` de la función actual.

Si no existen manejadores para tratar con una excepción, el bloque `try` se aborta y la excepción no es capturada. Cada manejador se introduce con la palabra reservada `catch` y tiene un único parámetro. Si no ocurre ninguna excepción, las sentencias se ejecutan de modo normal y ninguno de los manejadores será invocado.

Imaginemos la siguiente función:

```
int f(int x)
{
 int a = fa(x);
 int b = fb(x);
 return a / b;
}
```

Suponemos que las funciones `fa` y `fb` son dos funciones arbitrarias. Si la función `fb` devuelve un valor de 0, se tendrá un problema. Si no se hace nada, se realizará una división entre cero, lo que proporcionará un resultado imprevisto y no definido (infinito; la mayoría de los compiladores de C++ no admiten la

división entre cero). Se trata de tomar el control de la excepción levantada (división por cero) y tratar de hacer alguna acción adecuada para permitir que continúe el programa. Una solución podía ser esta:

```
int f(int x)
{
 int a = fa(x);
 int b = fb(x);
 if (b == 0)
 return INT_MAX;
 return a / b;
}
```

La constante `INT_MAX` se define en el archivo estándar `limits.h` e indica el número mayor que se puede almacenar en una variable de tipo `int`. Si, por ejemplo, el tipo `int` se representa con 16 bits, `INT_MAX` será igual a 32.767. Este sería el número entero mayor que se devolvería en el caso de que `b` fuera igual a 0, cosa que naturalmente es absurda.

La solución pasa por escribir

```
if (b == 0)
{
 cout << "Desbordamiento en la función f";
 abort();
}
```

pero entonces el programa se detiene y puede que esto no sea necesario. El resultado de la función `f` puede no ser tan importante, después de todo. Si la función llamada `f` ha encontrado que se ha producido un error, puede proseguir su ejecución. En resumen, se puede escribir la nueva versión de la función `f` que genera una excepción cuando `b` se convierte en 0.

```
int f(int x)
{
 int a = fa(x);
 int b = fb(x);
 if(b == 0)
 throw error_desbordamiento ("Error en f");
 return a/b;
}
```

La función `g` define un bloque `try` y un manejador `catch` asociado.

```
void g()
{
 int i;
 while (i)
 try {
 cout << "?";
 if (not cin >> i)
 break;
 int r = f(i);
 cout << "El resultado era " << r << endl;
 }
 catch (error_desbordamiento) {
 cout << "No se puede dar resultado" << endl;
 }
}
```

#### Ejemplo 20.4

La función `g` realiza la lectura de enteros del teclado. La entrada termina cuando aparece la combinación de caracteres final de archivo (`Ctrl-Z` o bien `Ctrl-D`). Por cada número leído, se llama a la función `f` y

el resultado de `f` se escribe en el terminal. Si se lee un número tal que `b` en la función `f` se hace igual a cero, se ejecuta la siguiente sentencia:

```
throw error_desbordamiento ("Error en f");
```

Dado que la excepción que ahora se produce no reside en un bloque `try`, no se tratará en la función `f`. En su lugar, `f` se interrumpirá y la excepción se enviará más tarde a la función `g`, que llama a `f`. En `g` existe entonces una excepción: la del punto en que fue llamada `f`, esto es, en la declaración:

```
int r = f(i);
```

Esta línea se aborta y dado que reside en un bloque `try`, el programa saltará al primero de los manejadores de bloques `try`, que coincide con el tipo `error_desbordamiento` y que es el manejador `catch`, obteniéndose entonces:

```
No se puede dar resultado
```

Tan pronto como se alcanza un manejador que captura la excepción, esta se elimina. La ejecución prosigue entonces normalmente con la sentencia que viene *después* del bloque `try`; es decir, la sentencia que viene después del último `catch`.



### Ejemplo 20.5

Se definen dos manejadores `catch`.

```
catch(char* mensaje)
{
 cerr << mensaje << endl;
 exit(1);
}
catch (...)
{
 cerr < " esto es todo pueblo " << endl;
 abort();
}
```

Está permitido que una firma con puntos suspensivos se corresponda con cualquier tipo de argumentos. También, el argumento formal puede ser una declaración abstracta, lo cual significa que puede tener información de tipo sin un nombre de variable.

El manejador se invoca en una expresión `throw` apropiada. En ese punto se sale del bloque `try`.

## 20.6 Especificación de excepciones

La técnica de manejo de excepciones se basa, como se ha visto, en la *captura de excepciones* que han ocurrido en las funciones que han sido llamadas. Varios métodos se pueden aplicar a tratar con diferentes tipos de excepciones. Una pregunta que cabe hacerse es: “¿Cómo se conoce cuál es el tipo de excepción que puede generar una función llamada?” Un método de conocer esto es, naturalmente, leer el código de programa de la función real, pero esto no es posible en la práctica cuando se aplica a funciones que son parte de grandes programas y las cuales llaman a otras funciones por sí mismas. Otro método es leer la documentación disponible sobre la función real y esperar que contenga información sobre los diferentes tipos de excepciones que se pueden generar. Desgraciadamente, tal información no siempre se puede encontrar.

C++ ofrece una tercera posibilidad. Una declaración de función puede contener una especificación de cuáles excepciones puede generar la función. Una *especificación de excepciones* proporciona una solución para listar las excepciones que una función puede lanzar. Garantiza que la función no lance ningún otro tipo de excepciones. Los sistemas bien diseñados con manejo de excepciones necesitan definir qué funciones lanzan excepciones y cuáles no. Con especificaciones de excepciones se describen exactamente cuáles excepciones, si existen, lanzan una función. También se tienen que disponer de controles para saber qué sucede si las funciones lanzan una excepción “*no prevista*”.

Una especificación de excepciones se añade a una declaración o a una definición de una función. Los formatos son:

```
tipo nombre_función(signatura) throw (e1, e2, eN); // prototipo
tipo nombre_función(signatura) throw (e1, e2, eN) // definición
{
 cuerpo de la función
}
```

e1, e2, eN lista separada por comas de nombres de excepciones (la lista especifica que la función puede lanzar directa o indirectamente esas excepciones, incluyendo excepciones derivadas públicamente de estos nombres).

Si se lanza una excepción diferente, el mecanismo de excepciones llama a `unexpected()`. Una función no lanza excepciones si la lista de excepciones está vacía. Se puede incluir una especificación de excepciones con cualquier función en C++, incluyendo funciones miembros de clases y funciones de plantilla.

Sintácticamente, una *especificación de excepciones* es parte de una declaración o definición de funciones.

Formato: `cabecera_función throw (lista de tipos)`.

Si la lista está vacía el compilador puede suponer que no se ejecutará ningún `throw` por la función, bien directa o indirectamente.

```
void noexcep(int i) throw ();
```

### Especificación de excepciones vacía

1. `void g (parámetros) throw()` la función no puede generar ninguna excepción  
{...}
2. `void g (parámetros)` la función puede generar excepciones de cualquier tipo  
{...}

### Ejemplo

```
void f (parámetros) throw (T1, T2)
{...}
```

↑  
especificación de excepciones

### Regla

Una especificación de excepciones sigue a la lista de parámetros de funciones. Se realiza con la palabra reservada `throw` seguida por una lista de tipos de excepciones encerradas entre paréntesis.

clase con funciones miembro que especifica excepciones.

```
class PilaTest {
public:
 // ...
 voiduitar (int &valor) throw (quitarEnPilaVacía);
 voidmeter (int valor) throw (meterEnPilaLlena);
private:
 // ...
};
```

### Ejemplo 20.6





### Ejemplo 20.7

En el prototipo de la función siguiente `f( )`, se indica que `int` y `double` son los tipos de excepciones que se puede requerir manejar si se invoca `f( )`.

```
void f(char c) throw (int, double)
```

Si la función `f( )` intenta generar una excepción diferente a los tipos `int` o `double` (o en general, tipos derivados de la lista `throw`), entonces esta excepción se corresponde con una invocación de la función `unexpected`, que de manera predeterminada termina el programa. Por ejemplo, supongamos que `f( )` tenga la definición siguiente:

```
void f(char) throw (int, double) {
 if (isupper(c))
 throw(1);
 else if (islower(c))
 throw(1.0);
 else if (c == '.')
 throw(c);
}
```

Si la invocación

```
f('a')
```

se ejecuta, entonces se genera una excepción de tipo `double`. Si se invoca la función

```
f('A')
```

entonces se genera una excepción del tipo `int`. Si se ejecuta

```
f('.)
```

el programa termina. Esto se debe a que se lanza una excepción de tipo `char`, donde el tipo `char` no es ni del tipo `int` ni `double`, ni se deriva de ninguno de los tipos `int, double`.

## 20.7 Excepciones imprevistas

Las especificaciones de las excepciones representan las excepciones que una función puede lanzar bien directa o bien indirectamente. Si una función lanza una excepción que no aparece en la especificación de excepciones de la función, la excepción es imprevista (`unexpected`). El mecanismo de excepciones llama a `unexpected( )`, que llama a `terminate( )` para detener el programa.

La función del sistema `terminate( )` se llama cuando ningún manejador se ha proporcionado para tratar con una excepción. La función `abort( )` se llama por omisión. Inmediatamente termina el programa, devolviendo el control al sistema operativo. Se puede también especificar otra acción utilizando `set_terminate( )` para proporcionar un manejador. Estas declaraciones se encuentran en el archivo `except.h`.

### Normas de especificación de excepciones

- Una especificación de excepción se puede añadir al final de una declaración de función
 

```
tipo_retorno f (parámetros) throw(T1, T2, T3, ...);
```

 Entonces la función solo genera excepciones de los tipos dados en la lista de `throw`.
- Una especificación de excepción vacía significa que ninguna excepción se generará por la función:
 

```
tipo_retorno f (parámetros) throw();
```
- Si no hay ninguna especificación de excepción, se pueden generar todos los tipos de excepciones:
 

```
tipo_retorno f (parámetros);
```
- Si se genera una excepción de un tipo no dado en la especificación de excepciones, se llama a la función `unexpected` y esta acción termina el programa. La función `unexpected` puede ser reemplazada por la función no estándar:
 

```
set_unexpected (nombre_de_función_estándar);
```

Esta acción puede generar una excepción de tipo `bad_exception`

## 20.8 Aplicaciones prácticas de manejo de excepciones

Para ilustrar el manejo de excepciones en C++ examinaremos dos programas en los que se hace un tratamiento completo de excepciones.

### Calcular las raíces de una ecuación de segundo grado

#### Aplicación 20.1

Tratamiento de las excepciones para la resolución de una ecuación de segundo grado.

Una ecuación de segundo grado de la forma  $ax^2 + bx + c = 0$  tiene las siguientes raíces reales.

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Los casos en los cuales las expresiones anteriores no tienen sentido son: 1)  $a = 0$ ; 2)  $b^2 - 4ac < 0$ , que no producen raíces reales, sino imaginarias. En consecuencia, se consideran excepciones de tipo error como:

```
enum error {no_raices_reales, coeficiente_a_cero};

Una codificación completa es:

#include <cstdlib>
#include <iostream>
#include <math.h>

using namespace std;
enum error {no_raices_reales, Coeficiente_a_cero};
void raices(float a, float b, float c, float &r1, float &r2) throw(error)
{
 float discr;
 if(b*b < 4*a*c)
 throw no_raices_reales;
 if(a==0)
 throw Coeficiente_a_cero;
 discr = sqrt(b * b - 4 * a * c);
 r1 = (-b - discr) / (2 * a);
 r2 = (-b + discr) / (2 * a);
}

int main()
{
 float a, b, c, r1, r2;
 cout << " introduzca coeficientes de la ecuación de segundo grado : ";
 cin >> a >> b >> c;
 try {
 raices (a, b, c, r1, r2);
 cout << " raíces reales " << r1 << " " << r2 << endl;
 }
 catch (error e)
 {
 switch(e) {
 case no_raices_reales :
 cout << "Ninguna raíz real" << endl;
 break;
 case Coeficiente_a_cero :
```

```

 cout << "Primer coeficiente cero" << endl;
 }
}
system("PAUSE");
return EXIT_SUCCESS;
}

```

### Control de excepciones en una estructura tipo pila

La *pila* es una estructura lineal que recibe datos por un extremo y los retira por el mismo extremo, siguiendo la regla “último elemento en entrar, primero en salir”.

Declaremos una clase *Pila*, que es una pila de enteros con un máximo de diez elementos. La clase *Pila* declara dos clases anidadas, *Desbordamiento* (*overflow*) y *Subdesbordamiento* (*underflow*), que se utilizarán para manejar las condiciones de error: *desbordamiento* (la pila está llena) y *subdesbordamiento* o *desbordamiento negativo* (la pila está vacía); en ambos casos, ni se pueden introducir datos en la pila (*Desbordamiento*) ni sacar datos de la misma (*Subdesbordamiento*). Cuando la pila se desborda en alguno de los dos extremos (cima o el fondo), se lanzan las excepciones apropiadas:

```

#include <iostream.h>
const TAMAPILA = 5; // tamaño máximo de la pila
using namespace std;
class Pila
{
public:
 class Desbordamiento // una clase excepción
 {
 public:
 int valdesborde;
 Desbordamiento(int i): valdesborde(i) { }
 };
 class Subdesbordamiento // una clase excepción
 {
 public:
 Subdesbordamiento() { }
 };
 Pila () {cima = -1;}
 void meter(int item)
 {
 if (cima < (TAMAPILA-1)) lapila[++cima] = item;
 else throw Desbordamiento(item);
 }
 int sacar()
 {
 if (cima > -1) return lapila[cima--];
 else throw Subdesbordamiento();
 }
}

```

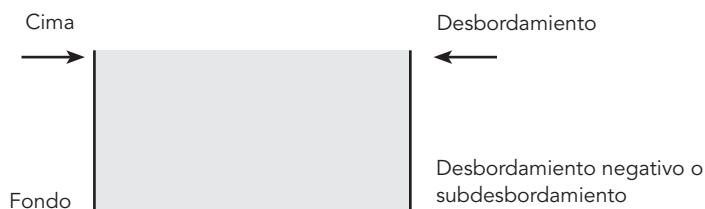


Figura 20.2 Estructura Pila.

```

private:
 int lapila[TAMAPILA];
 int cima;
};

```

El siguiente programa principal declara una pila y manejadores de excepciones para las condiciones de desbordamiento y subdesbordamiento de la pila. El programa fuerza a la pila para que se desborde, haciendo que se invoque al manejador de excepciones.

```

#include <iostream>
using namespace std;

void main()
{
 Pila mipila;
 int i = 5, j = 25;
 // Bloque try
 try
 {
 mipila.meter(i);
 mipila.meter(j);
 mipila.meter(1);
 mipila.meter(12345);
 mipila.meter(9999);
 // Pila llena con cinco números se fuerza una excepción
 mipila.meter(100); // Lanza Pila::Desbordamiento
 }
 // Manejadores de excepciones
 catch(Pila::Desbordamiento &p)
 {
 cout << "La pila se ha desbordado tratando de meter :"
 << p.valdesborde << endl;
 }
 catch(Pila::Subdesbordamiento &p)
 {
 cout << "Se ha producido un rebose negativo" << endl;
 }
}

```

Cuando se ejecuta el programa, se visualiza:

```
La pila se ha desbordado tratando de meter : 100
```



## Resumen

- Las excepciones son, por lo general, condiciones (situaciones) de error imprevistas. Normalmente estas condiciones terminan el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos son: división por cero, índices fuera de límites en un arreglo, etcétera.
- C++ posee un mecanismo para manejar excepciones, muy similar al del lenguaje Ada.
- El código C++ puede levantar (*raise*) una excepción utilizando la expresión *throw*. La excepción se maneja invocando un manejador de excepciones seleccio-

nado de una lista de manejadores que se encuentran al final del bloque *try* del manejador.

- Sintácticamente, *throw* presenta los formatos:

```

throw
throw expresión
throw expresión lanza una excepción en un bloque
try. throw sin argumentos se puede utilizar en una
sentencia catch para relanzar la excepción actual.

```

- Sintácticamente, un bloque *try* tiene el formato

```
try
sentencia compuesta
lista de manejadores
```

El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada. El orden en el que están definidos los manejadores, determina el orden en el que un manejador de una excepción levantada va a ser invocada.

- La sintaxis de un manejador es:

```
catch (argumento formal)
sentencia compuesta
```

- La especificación de excepciones es parte de una declaración de función y tiene el formato:

```
cabecera función throw (tipo lista)
```

- La función `terminate()` se llama cuando ningún otro manejador se ha previsto para tratar una excepción. El manejador del sistema `unexpected()` se llama cuando una función lanza una excepción que no está en su lista de especificaciones de excepciones. De manera predeterminada, `terminate()` llama a la función `abort()`. El comportamiento de `unexpected()` es llamar a `terminate()`.

## Ejercicios

20.1 El siguiente programa que maneja un algoritmo de ordenación básico no funciona bien. Sitúe declaraciones en el código del programa de modo que se compruebe si este código funciona correctamente. Escriba el programa correcto.

```
// Programa de ordenar
#include <iostream.h>
using namespace std;
void intercambio (int x, int y)
{
 int aux = x;
 x = y;
 y = aux;
}
void ordenar (int l[], int n)
{
 int i, j;
 for (i=0; i!=n; ++i)
 for (j=i; j!=n; ++j)
 if (l[j] < l[j+1])
 intercambio (l[j], l[j+1]);
}
main()
{
 int z[12]={14,13,8,7,6,12,11,10,9,
 -5,1,5};
 ordenar (z, 12);
 for (int i=0; i<12; ++i)
 cout << z[i] << '\t';
 cout << '\ ordenado' << endl;
}
```

20.2 Escribir el código de una clase C++ que lance excepciones para cuantas condiciones estime convenientes. Utilizar una cláusula `catch` que utilice una sentencia `switch` para seleccionar un mensaje apropiado y terminar el cálculo.

*Nota:* Utilice un tipo enumerado para listar las condiciones enum error\_pila {overflow, underflow,...};

20.3 El código siguiente sirve para definir y manejar excepciones:

```
class cadena {
char* s;
public:
 enum {minLong = 1, maxLong = 1000};
 cadena();
 cadena(int);
 ...
};
cadena::cadena(int longitud)
{
 // definir excepción fuera de límites y
 lanzarlo
 if (longitud < minLong || longitud >
 maxLong)
 throw (longitud);
 s = new char [longitud];
 // definir excepción "fuera de memoria"
 y lanzamiento
 if (s == 0)
 throw ("Fuera de memoria");
}
...
void f(int n)
{
 try{
 cadena cad (n);
 }
 catch (char* Msgerr)
 {
 cerr << Msgerr << endl;
 abort();
 }
}
```

```
 catch(int k)
 {
 cerr << 'Error de fuera de rango-' <<
 k << endl;
 }
}
```

a) ¿Qué sucede si las líneas siguientes se eliminan de la función `f` y se invoca `f` con el argumento 5000?

b) ¿Qué sucede si se cambia el constructor por el código siguiente y se invoca `f` con el argumento 5000?

```
cadena::cadena (int n) throw (char*)
{
 ...
}
```



PARTE

V

## Programar en Java





## De C/C++ a Java 6/7/8

### Contenido

- 21.1 Historia del lenguaje Java: de Java 1.0 a Java 7
- 21.2 El lenguaje de programación Java
- 21.3 Un programa sencillo en Java
- 21.4 Tipos de datos
- 21.5 Variables, constantes y asignaciones
- 21.6 Operadores y expresiones
- 21.7 Funciones matemáticas
- 21.8 Cadenas

- 21.9 Código de caracteres ASCII y Unicode
- 21.10 Entrada y salida
- 21.11 Flujo de control
- 21.12 Arreglos (arrays)
- 21.13 Applets
- 21.14 Creación de un applet
- 21.15 Ciclo de vida de un applet
- › Resumen
- › Ejercicios
- › Problemas

### Introducción

Java es conocido como un lenguaje de programación para desarrollar aplicaciones de internet. Sin embargo, en este libro, como en muchos otros, se utilizará Java como un lenguaje de programación de propósito general, adecuado para la mayoría de las aplicaciones sean o no para internet. De hecho, la primera versión de Java no fue creada para ninguna de estas tareas, aunque posteriormente evolucionó para desarrollo de programación orientada a objetos y, sobre todo, de aplicaciones para internet.

En este capítulo, tras el estudio de las técnicas de programación en C y C++, se afronta el inicio de programación en Java y se explican los conceptos básicos de dicha programación, como tipos de datos, flujo de control, operadores y expresiones, cadenas, entrada y salida, y arreglos (arrays), ya conocidos por el lector, y estudiados ahora desde la perspectiva de Java.

Java permite escribir un tipo especial de programa denominado *applet*, pensado para ejecutarse mediante un navegador en una página web.

### Conceptos clave

- › aplicación
- › *applet*
- › arreglo irregular
- › *bytecode*
- › caracteres Unicode
- › clase
- › clase *String*
- › código intermedio
- › coordenadas
- › *for each*
- › *JApplet*
- › Java 8
- › JDK
- › máquina virtual
- › Scanner

### 21.1 Historia del lenguaje Java: de Java 1.0 a Java 8

En 1991, James Gosling y otros desarrolladores de Sun Microsystems, comenzaron a trabajar en un proyecto denominado “Green”, que pretendía diseñar un sencillo lenguaje de programación para utilizar en dispositivos de consumo como aparatos de televisión

y máquina lavadoras. El lenguaje se diseñó para que fuese sencillo y neutro a la arquitectura, de modo que pudiera ser ejecutado con una gran variedad de *hardware*, es decir procesadores y dispositivos asociados.

La idea original consistía en diseñar un lenguaje de desarrollo para realizar aplicaciones que fuesen independientes del sistema operativo, sin tener que compilar de nuevo el código fuente; es decir, un lenguaje portable que generara *código intermedio* para una máquina hipotética. Se creó la denominada *máquina virtual*, luego denominada *máquina virtual Java* (JVM, Java Virtual Machine).

Este código intermedio se podría utilizar en cualquier máquina que tuviera el intérprete adecuado. El proyecto “*Green*” entregó una primera versión en 1992, denominado “7”, que consistía en un control remoto inteligente. El proyecto “*Green*” se transformó en “*First Person, Inc.*” pero no tuvo éxito comercial y en 1994 se disolvió.

Mientras sucedía todo lo anterior en Sun, la Word Wide Web crecía a gran velocidad. La clave era el navegador (**browser**) que traducía las páginas de hipertexto a la pantalla. En 1994, la mayoría de las personas estaban utilizando Mosaic, un navegador web no comercial creado en la Universidad de Illinois en 1993 (Marc Andreessen, estudiante de posgrado fue uno de sus inventores; posteriormente fue uno de los cofundadores y director de tecnología en Netscape). Gosling y su equipo se reconvirtieron y comenzaron a mirar a la Web. Su primer trabajo, ya conocido como Java, se presentó en 1995 con ocasión de la exposición SunWorld, después de que Patrick Naughton y Jonathan Payne de Sun Microsystem desarrollaran un navegador web que podía correr programas Java en Internet. El navegador web evolucionó al navegador conocido como HotJava. En el otoño de 1995, Netscape Incorporated creó un navegador web capaz de ejecutar programas Java. Otras compañías siguieron desarrollando software que se apoyaba en programas Java.

Sun lanzó su primera versión de Java a principios de 1990 (Java 1.0). Las primeras deficiencias de Java 1.0 se resolvieron con la presentación en 1997 de Java 1.1, que añadía la propiedad de reflexión y un nuevo modelo de sucesos (eventos), para programación de interfaces gráficos de usuario, IGU (Graphical User Interface, GUI). En 1998, se presentó Java 1.2 y a finales del mismo año, Sun cambió el nombre del software que pasó a denominarse *Java 2 Standard Edition Software Development Kit Version 1.2*. Posteriormente Sun presentó otras dos ediciones, *Micro Edition* para dispositivos empotrados como teléfonos celulares y *Enterprise Edition* para procesamiento de servidores. Este libro se centra en la Standard Edition. Las versiones 1.3 y 1.4 se presentaron en la primera mitad de la década de 2000 con mejoras incrementales sobre la revisión inicial Java 2.

En 2005 se lanzó la **versión 5.0** que es la primera revisión (*release*) desde la versión 1.1 que actualiza el lenguaje Java de modo significativo. Se añaden tipos genéricos (que son aproximadamente las plantillas de C++) y otras características inspiradas en C#, paradójicamente el lenguaje creado por Microsoft para competir con Java, como el bucle “*for each*” y metadatos.

La **versión 6** se lanzó a finales de 2006. En este caso no hay mejoras sustanciales en el lenguaje sino mejoras adicionales, como mejoras en la biblioteca de clases.

La **versión 7** de Java se lanzó en julio del año 2011 y su última actualización Java 7 Update 51 en enero de 2014.

La **versión 8** de Java ha sido anunciada el 25 de marzo de 2014 y desde entonces disponible para descarga.

**Tabla 21.1** Versiones de Java.\*

| Versión | Año  | Nuevas características del lenguaje                                                                             |
|---------|------|-----------------------------------------------------------------------------------------------------------------|
| 1.0     | 1996 | Definición del lenguaje                                                                                         |
| 1.1     | 1997 | Clases internas                                                                                                 |
| 1.2     | 1998 | Colecciones, Swing                                                                                              |
| 1.3     | 2000 | Mejoras en prestaciones                                                                                         |
| 1.4     | 2002 | Aserciones, XML                                                                                                 |
| 5       | 2004 | Clases genéricas, bucle “ <i>for each</i> ”, <i>autoboxing</i> , metadatos, enumeraciones, importación estática |
| 6       | 2006 | Mejoras en la biblioteca                                                                                        |
| 7       | 2011 | Mejoras en seguridad. Actualización de bibliotecas                                                              |
| 8       | 2014 | Presentada marzo 2014. Centrada en la convergencia de Java SE 8, Java ME 8 y Oracle Java Embedded               |

\*Página oficial de Java: [www.java.com/es](http://www.java.com/es)

## 21.2 El lenguaje de programación Java

Java es un lenguaje de programación con una gran cantidad de características técnicas notables. Ya que Java no fue diseñado específicamente para estudiantes, no es un lenguaje fácil de aprender, en primera instancia, ya que requiere una cierta cantidad de maquinaria técnica para escribir programas en Java, incluso los más simples. Sin embargo, las ventajas que tienen los profesionales terminan a la larga convirtiéndose también en ventajas para los estudiantes a medida que estos van superando esas dificultades y terminan convirtiendo a los estudiantes con experiencia en magníficos programadores. Esta es una de las razones esenciales de este libro, enseñar a programar de un modo evolutivo; primero aprender el lenguaje moderno por excelencia, C; a continuación el lenguaje de programación orientado a objetos, por autonomía, C++; para terminar con el lenguaje profesional y de internet del siglo xxi, **Java**.

Las características de Java han sido definidas por sus autores en un artículo educativo (“White Paper”) ya muy famoso y donde se describen las 11 propiedades sobresalientes (*buzzwords*) de Java:

- Sencillo (simple).
- Orientado a objetos.
- Capacidad de trabajo en red (Network-Savvy).
- Robusto.
- Seguro.
- Neutro (independiente) de la arquitectura.
- Portable.
- Interpretado.
- Altas prestaciones.
- Multihilo (*Multithread*)
- Dinámico.

### Sitio web de referencia

Portal oficial de Java de Oracle

[www.oracle.com/technetwork/java/index.htm](http://www.oracle.com/technetwork/java/index.htm)

Página “Java y tú”: Descargar hoy (Oracle) [www.java.com/es](http://www.java.com/es)

Sus autores pretendían de Java, desde el punto de vista educativo, que fuera más fácil de aprender que C++ y más difícil de hacer un mal uso del lenguaje como pensaban que sucedía con C y C++.

Destaquemos entre todas las propiedades de Java las que consideramos más significativas en esta fase del aprendizaje: **Java** es un *lenguaje de programación orientado a objetos, independiente de la plataforma (neutra), seguro* y de más *facilidad de aprendizaje* que C++, reiteramos que es el concepto original de los autores de Java.

- **Programación orientada a objetos (POO).** Recordemos que un programa orientado a objetos es un grupo de objetos que trabajan juntos (una técnica de programación que se centra en los datos = objetos y en las interfaces a ese objeto). Los objetos se crean utilizando modelos o plantillas denominadas *clases*, que contienen datos y los programas, sentencias, requeridas para usar los datos. Java es completamente orientado a objetos.
- **Independiente (neutralidad) de la plataforma.** Es la capacidad de un programa de correr (ejecutarse) sin modificación en diferentes entornos de computación, Windows, Unix, Linux, Macintosh (Mac OS) y para plataformas móviles como el sistema operativo Android. Los programas Java se traducen primero a un *lenguaje intermedio* que es el mismo para todas las aplicaciones o aparatos (computadoras). Este lenguaje intermedio se llama *Java bytecode* o simplemente *bytecode* (la máquina virtual Java de su computadora ejecuta el código portable) que se ejecuta en cualquier sistema operativo, software o dispositivo que tenga un intérprete (máquina virtual) Java. Así, es posible crear un programa en una máquina Windows 8 y se ejecutará también en un servidor web Linux, en un Apple Mac utilizando OS X o en plataformas Android de Google. Siempre que la máquina o plataforma disponga de un intérprete Java se podrá ejecutar el código *bytecode* (existen versiones del intérprete para todos los sistemas operativos más utilizados). En otras palabras, los archivos de extensión *.class* (archivos con el *bytecode*), se compilan en sentencias de máquina que ejecuta el procesador de la computadora.
- **Seguro.** Java está concebido para ser utilizado en entornos distribuidos (en red) y ha hecho de la seguridad, una de sus fortalezas. Desde el principio, Java se diseñó para crear sistemas libres de virus y hacer que ciertos tipos de ataques sean imposibles.

- **Facilidad de aprendizaje.** Java fue diseñado para ser más fácil que C++ principalmente en los siguientes aspectos:
  - Java automáticamente se preocupa de la *asignación y liberación de memoria*, liberando a los programadores de esta tediosa y compleja tarea.
  - Java no incluye *punteros*, una característica de programación muy potente para ser utilizada por programadores experimentados pero de alto riesgo por su facilidad de mal uso.
  - Java no posee *herencia múltiple* en programación orientada a objetos; esta propiedad ha sido reemplazada por el concepto más sencillo de interfaces y su modelo de metaclasses Java.

La *ausencia de punteros* y la *presencia de la gestión automática de la memoria* son dos elementos clave en la seguridad de Java, que antes se describió.

## Tipos de programas Java

Existen dos tipos de programas en Java: *applets* y *aplicaciones*. Un *programa de aplicación* o una *aplicación* es simplemente un programa ordinario. Desde un punto de vista de orientación a objetos, es una *clase* con un método (en otros lenguajes, recuerde el lector, se denominan *función*, *método*, *procedimiento* o *subprograma*) llamado `main` y cuando se ejecuta el programa Java, el sistema en tiempo de ejecución invoca automáticamente a `main` (es decir, se inicia automáticamente el método o función `main`).

Existe otro tipo de programa conocido por *applet* (little Java application). *Applets* y aplicaciones son casi idénticos. La diferencia es que las aplicaciones se refieren a que se ejecutan en su computadora de igual forma que otros programas, mientras que un *applet* se ejecuta desde un navegador (*browser*) de la web y así se puede enviar a otra posición o sitio de internet y ejecutarse allí en ese sitio. Los *applets* utilizan siempre una interfaz de ventana, pero no todos los programas con una interfaz ventana son *applets*.

La idea del *applet* es simple. Los usuarios descargarán *bytecodes* de Java de internet y los ejecutan sus propias máquinas.

Los programas Java que trabajan en páginas web se denominan *applets*.

Para utilizar un *applet* solo se necesita un navegador web compatible Java, que ejecutará los *bytecodes*. Si se visita una página web que contiene un *applet*, el código comienza a ejecutarse automáticamente. Cuando el usuario descarga un *applet*, se trabaja como si se incrustase una imagen en una página web. El *applet* se convierte en una parte de la página y el texto fluye alrededor del espacio utilizado por el *applet*. No se necesita instalar ningún software ya que un *applet* de Java se ejecutará en cualquier navegador compatible Java que prácticamente son todos, y al menos los más populares Explorer, Firefox, Opera, etcétera.

Aunque los *applets* fueron diseñados para ser ejecutados desde un navegador web, pueden ser ejecutados con un programa denominado *visualizador de applet* (*appletviewer*). En ocasiones, los *applets* se ejecutan como programas autónomos utilizando el *visualizador de applet* (*appletviewer*).

## 21.3 Un programa sencillo en Java

Un programa Java es realmente una definición de una clase con un método `main`. La aplicación Java se ejecuta cuando se utiliza la orden `java` que lanza la Máquina Virtual Java (JVM, Java Virtual Machine). La figura 21.1 muestra el código fuente del clásico programa "Hola mundo".

| Código                                                                                                                                                                                                                                                | Nombre de clase<br>(programa) |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| <pre>public class PrimerPrograma {     public static void main(String[ ] args)     {         // visualizar un mensaje de saludo         System.out.println("¡ Hola, Mundo !");     } // fin del método main } // fin de la clase PrimerPrograma</pre> | Método main                   |
| Salida                                                                                                                                                                                                                                                |                               |
|                                                                                                                                                                                                                                                       | ¡ Hola, Mundo !               |

Figura 21.1 Primer programa Java.

Examinemos el programa de demostración. En primer lugar, se debe dar un nombre al archivo del programa fuente: el mismo que el nombre de la clase pública con (en nuestro ejemplo, `PrimerPrograma.java`) la extensión `java`. Introduzca el código fuente con un editor; a continuación compile el programa y tendrá como resultado un archivo que contiene `bytecodes` de la clase. El compilador asigna automáticamente un nombre al archivo de `bytecode`, `PrimerProgramma.class`, y lo almacena en el mismo directorio que el archivo fuente. Por último, se ejecuta el programa con la orden siguiente:

```
java PrimerProgramma
```

Cuando se ejecuta el programa se visualiza la cadena `¡Hola, Mundo !`

### Nota

Al ejecutar un programa compilado `java NombreClase`, la máquina virtual Java arranca siempre la ejecución del método `main` de la clase indicada. Por consiguiente, debe existir un método `main` en el archivo fuente de su clase en su código de ejecución. Naturalmente, se pueden añadir métodos propios a una clase y llamarlo desde el método `main`.

A continuación se analiza la estructura del programa línea a línea: Java es sensible a las mayúsculas (`Main` es distinto de `main`) y por consiguiente una escritura incorrecta puede producir errores. Java tiene *diseño de formato libre*, se puede utilizar cualquier número de espacios y sangrados o saltos de línea para separar palabras, aunque se recomienda al lector seguir las notaciones de escritura que se han seguido hasta ahora en el libro.

En la primera línea

```
public class PrimerProgramma
```

`public` es una palabra clave y un modificador de acceso y su significado es similar al ya estudiado en C++. A continuación de la palabra clave `class` viene el nombre de la clase. Las normas para los nombres de las clases son muy flexibles. Deben comenzar con una letra, y después pueden tener una combinación de letras y dígitos. La longitud es ilimitada, pero no se puede utilizar una palabra reservada (como `class` o `public`) para un nombre de clase. La convención estándar para dar nombres es que los nombres de las clases comiencen con una letra mayúscula. Si un nombre se compone de varias palabras, utilice una letra mayúscula en la inicial de cada palabra. `PrimerProgramma`, `BienvenidoUsuario`, ...

La construcción

```
public static void main(String[] args)
{
 sentencias de programa
}
```

define un método denominado `main`. Cada aplicación Java debe tener un método `main`. La mayoría de los programas Java contienen otros métodos además de `main`.

El parámetro `String[ ] args` es una parte requerida del método `main` (con argumentos de la línea de órdenes). La palabra `static` (en el capítulo siguiente se analiza con más detalle) indica que el método `main` no opera en un objeto. La siguiente línea es un comentario

```
// visualizar un mensaje de saludo
```

El fragmento de código

```
System.out.println("¡Hola, Mundo !");
```

es el cuerpo del método. Este método tiene solo una sentencia, pero como en todos los lenguajes de programación pueden existir diferentes sentencias. En Java, cada sentencia ha de terminar con un punto y coma. En particular, el retorno de carro no señala el final de una sentencia. El cuerpo del método `main` contiene una sentencia que saca una única línea a la consola. Aquí `System.out` llama o invoca a su método `println`. Observe que el punto se utiliza para invocar un método.

### Sintaxis

```
System.out.println("¡Hola, Mundo !");
```

Objeto.metodo(parámetros)

### A recordar

C/C++ y Java utilizan dobles comilla para delimitar cadenas.

Si el método no tiene parámetros se deben utilizar parámetros vacíos. La sentencia

```
System.out.println();
```

visualiza una línea en blanco.

Los comentarios en Java, como en la mayoría de los lenguajes de programación, no se muestran en el programa ejecutable. Java tiene tres métodos de señalar comentarios. El método más común es el conocido de C++:

```
// esto es un comentario
```

y el segundo método es también similar y utiliza los separadores /\* y \*/ cuando se desean comentarios en varias líneas.

Existe un tercer tipo de comentario que se puede utilizar para generar documentos automáticamente. Este tipo de comentarios utiliza un símbolo /\*\* para inicio y \*/ para el final.

## 21.4 Tipos de datos

Java es *un lenguaje fuertemente tipado*. Esto significa que cada variable debe tener un tipo declarado. Hay ocho *tipos primitivos* en Java y cada uno de ellos es una palabra reservada. Cuatro son tipos enteros, dos son tipos coma flotante; uno es el tipo carácter, `char`, utilizado para caracteres individuales (código **ASCII** y **Unicode**) y un último dato es un tipo `boolean` para representar los tipos de dato lógicos.

## 21.5 Variables, constantes y asignaciones

En Java cada variable tiene un tipo. El nombre de una variable (u otro elemento del programa) es un identificador. Un identificador en Java no debe comenzar con un dígito y todos los caracteres deben ser letras, dígitos o carácter de subrayado (el símbolo \$ está permitido, pero está reservado para fines específicos y no se debe utilizar \$ en identificadores Java).

En Java, una variable debe estar declarada antes de ser utilizada.

Tabla 21.2 Tipos primitivos.\*

| Tipo                | Requisito de almacenamiento | Rango                                                                                  |
|---------------------|-----------------------------|----------------------------------------------------------------------------------------|
| <code>int</code>    | 4 bytes                     | -2.147.483.648 a 2.147.483.647                                                         |
| <code>short</code>  | 2 bytes                     | -32.768 a 32.767                                                                       |
| <code>long</code>   | 8 bytes                     | -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807                                 |
| <code>byte</code>   | 1 byte                      | -128 a 127                                                                             |
| <code>float</code>  | 4 bytes                     | $3.4 \times 10^{-45}$ a $3.4 \times 10^{45}$<br>(6-7 cifras decimales significativas)  |
| <code>double</code> | 8 bytes                     | $1.7 \times 10^{-308}$ a $1.7 \times 10^{308}$<br>(15 cifras decimales significativas) |
| <code>char</code>   | 2 bytes                     | carácter Unicode<br>\\u0000 a \\uFFFF<br>p.e. \\n \\u000a \\r \\u000d                  |

\* Todos los tipos de datos tienen signo, excepto `char` que no tiene signo.

**Sintaxis**

```
Tipo variable1, variable2, ...
```

**Ejemplo**

```
int numeroDeAlumnos;
double pesoNeto, pesoTotal;
double salario;
int diasVacaciones;
```

**Inicialización de variables**

Después de declarar una variable se debe inicializar explícitamente mediante una sentencia de asignación.

**Sintaxis**

```
variable = expresión;
```

**Ejemplo**

```
int diasDeVacaciones;
diasDeVacaciones = 20;
char grado = 'A';
int cuentaInicial = 50, cuentaFinal;
```

**Errores típicos**

```
int diasDeV;
System.out.println(diasDeV); // Error, variable no inicializada
```

En Java, se pueden poner declaraciones en cualquier parte de su programa, y, a veces se considera buena práctica declarar variables lo más cerca posible del punto donde se utilizan por primera vez.

```
double salario = 45000;
System.out.println(salario);
int diasSemanaSanta = 10;
System.out.println("Semana Santa: " + diasSemanaSanta);
```

**Inicialización de una variable en una declaración**

Se puede combinar la declaración de una variable como una sentencia de asignación que proporciona el valor de una variable.

**Sintaxis**

```
Tipo variable_1 = expresión_1, variable_2 = expresión_2, ... ;
```

**Ejemplo**

```
int numeroAlumnos = 50, ultimoAlumno, numeroDeGrupo = 4;
double velocidad = 29.5, distancia = velocidad*10;
char final = 'F';
```

## Asignación multiple; combinación de asignación con operadores aritméticos

Se pueden realizar asignaciones múltiples con el operador = y variables del mismo tipo. El formato es :

```
variable1 = variable2 = variable3 = ... = expresión;
```

### Ejemplo:

```
int x, y, z;
x = y = z = 25;
```

Al igual que en C/C++ se pueden realizar asignaciones con el operador = y un operador aritmético de la forma:

```
variable op = expresión;
```

es equivalente a

```
variable = variable op expresión;
```

## Ejemplo

```
numero += 5;
total -= descuento;
cambio %= 39;
```

## Equivale a

```
numero = numero + 5;
total = total - descuento;
cambio = cambio % 39;
```

## Constantes

En numerosos programas se necesitan constantes numéricas pero es una buena idea usar nombres simbólicos para todos los valores, incluso para los que parecen evidentes. En Java, las constantes se identifican con la palabra clave `final`. Una variable etiquetada como `final` nunca puede modificar su valor ya que se ha convertido en una constante.

### Regla

Utilizar constantes con nombres para hacer sus programas más fáciles de leer y mantener.

Muchos programadores utilizan nombres en letras mayúsculas para constantes (variables `final`).

### Ejemplo

```
final double VALOR_DESCUENTO = 14.50;
final double IVA = 16;
final double EURO = 166.67;
```

A veces se necesitan valores constantes en diferentes métodos de una clase. Entonces deben declararse junto con los campos instancia de una clase y etiquetarse como `static` y `final`. La palabra reservada `static` significa que la constante pertenece a la clase.

### Ejemplo

```
public class MercadoValores
{
 // métodos
 ...
 // constantes
 public static final double PESO = 0.55;
 public static final double EURO = 1.55;
}
```

## Definición de constantes en un método o en una clase

### 1. En un método

```
final Tipo nombreVariable = expresión;
```

## 2. En una clase

```
especificadorAcceso static final Tipo nombreVariable = expresión;
```

### Ejemplo

```
final double VALOR_EURO = 166.67;
public static final double EUROS_LITRO = 1.75;
```

Una clase Constantes para representar formatos de impresión.

```
public class Constantes
{
 public static void main(String [] arg)
 {
 final int NUM_LINEAS_PAGINA = 55;
 final double ANCHURA_PAPEL = 20.5;
 System.out.println("Tamaño papel: " +
 ANCHURA_PAPEL* NUM_LINEAS_PAGINA);
 }
}
```

Las clases en java permiten declarar constantes que estarán disponibles para todos los métodos de la clase, incluso para otras clases. Estas constantes se denominan **constantes de clases**. Se establece una constante de clase con las palabras reservadas `static final`. Un ejemplo de uso de una constante de clase:

```
public class ConstantesDemo
{
 public static void main(String [] arg)
 {
 double anchoPapel = 24.5;
 double altoPapel = 54.5;
 ...
 }
 public static final double IVA = 20;
}
```

Obsérvese que la definición de la constante de clase aparece fuera del método `main`. Por consiguiente la constante puede ser utilizada en otros métodos de la misma clase.

## 21.6 Operadores y expresiones

Como en la mayoría de los lenguajes, Java permite formar expresiones utilizando variables, constantes y operadores aritméticos. Estas expresiones se pueden utilizar en cualquier lugar.

```
int x = 5;
int y = x + 5;
int z = x * y;
```

### Operadores aritméticos

Cinco operadores se utilizan para operaciones básicas en Java: `+` `-` `*` `/` y `%` (módulo). El operador `/` representa la división entera si ambos argumentos son enteros y división en coma flotante, en caso contrario.

|        |                |
|--------|----------------|
| 15/2   | es igual a 7   |
| 15%2   | es igual a 1   |
| 15.0/2 | es igual a 7.5 |

Ya se ha tratado anteriormente, pero es importante recordarlo: asignación múltiple en operadores:

```
x = y = z = 15; // a todas las variables se les asigna el valor 15
x += y; equivale a x = x + y;
x /= y; equivale a x = x / y;
```

Java, al igual que C y C++, tiene operadores de incremento y de decremento.

|     |                                   |
|-----|-----------------------------------|
| n++ | añade 1 al valor de la variable n |
| n-- | resta 1 al valor de la variable n |

```
int n = 12;
n++; cambia el valor de n a 13
```

Existen, al igual que C y C++, dos formas en estos operadores: **postfija** (el operador se sitúa después del operando) y **prefija** (el operador se sitúa delante del operando).

### Ejemplo

```
int n = 7;
int m = 7;
int a = 2 * ++m; // a es igual a 16, m es 8
int b = 2 * n++; // b es igual a 14, n es 8
```

## Operadores relacionales y lógicos

Java tiene diferentes operaciones para realizar comparaciones entre variables, variables y literales, y otros tipos de información de un programa. Estos operadores se utilizan en expresiones que devuelven valores *boolean* de *true* o *false*.

| Operador           | Significado       | Ejemplo                |
|--------------------|-------------------|------------------------|
| <code>==</code>    | Igual             | <code>x == 5</code>    |
| <code>!=</code>    | No igual          | <code>x != 5</code>    |
| <code>&lt;</code>  | Menor que         | <code>x &lt; 5</code>  |
| <code>&gt;</code>  | Mayor que         | <code>x &gt; 5</code>  |
| <code>&lt;=</code> | Menor o igual que | <code>x &lt;= 5</code> |
| <code>&gt;=</code> | Mayor o igual que | <code>x &gt;= 5</code> |

Java, siguiendo a C++, también utiliza los operadores `&&` (operador lógico “*and*”) y `||` (operador lógico “*or*”). Los operadores `&&` y `||` se evalúan en “*cortocircuito*”. El segundo argumento no se evalúa si el primer argumento determina ya el valor.

1. `expresión1 && expresión2`

Si el valor de `expresión1` es falso, entonces es imposible que el resultado sea verdadero.

2. `expresión1 || expresión2`

Es automáticamente verdadero si la primera expresión es verdadera, sin evaluar la segunda expresión.

## Operadores de manipulación de bits

Cuando se trabaja con tipos enteros, existen operadores que pueden trabajar directamente con los bits que forman los enteros. Esto significa que se pueden utilizar técnicas de enmascaramiento para obtener bits individuales de un número.

Los operadores de manipulación de bits son:

```
& ("and") | ("or") ^ ("xor") ~ ("not")
```

estos operadores trabajan sobre patrones de bits.

### Ejemplo

```
int n;
int cuartoBitMenorPeso = (n & 8);
```

producirá un 1 si el cuarto bit empezando por la derecha, el menos significativo, de `n` es 1, y 0 si no lo es.

|          |          |
|----------|----------|
| xxxx1xxx | xxxx0xxx |
| & 1000   | & 1xxx   |
| _____    | _____    |
| 1        | 0        |

Los operadores `>>` y `<<` desplazan un patrón de bits a la derecha o a la izquierda. Estos operadores son muy útiles cuando se necesita construir patrones de bits para enmascarar bits.

El operador `>>>` rellena los bits superiores con cero, mientras que `>>` amplía el bit de signo en los bits de mayor peso.

## Precedencia de operadores

En general, el orden de evaluación de primero a último es:

- Operadores de incremento y decremento.
- Operadores aritméticos.
- Comparaciones.
- Operadores lógicos.
- Expresiones de asignación.

## 21.7 Funciones matemáticas

La clase `Math` contiene una colección de funciones matemáticas que se pueden necesitar, dependiendo del tipo de programación que deseé realizar. Por ejemplo, para calcular  $x^n$

```
Math.pow(x, n)
```

Sin embargo para calcular  $x^2$  puede ser más eficiente calcular  $x*x$ .

Para extraer la raíz cuadrada de un número se puede utilizar el método `sqrt`:

```
double x = 9.0;
double y = Math.sqrt(x);
System.out.println("y = " + y); // se visualiza 3
```

### Ejemplo

Calcular el valor de  $(-b + \sqrt{b^2 - 4ac}) / 2a$

```
(-b + Math.sqrt(b*b -4*a*c)) / (2*a)
```

La tabla 21.3 muestra los métodos matemáticos de la clase `Math`.

**Tabla 21.3** / Métodos matemáticos.

| Función                        | Devuelve                                 |
|--------------------------------|------------------------------------------|
| <code>Math.sqrt(x)</code>      | raíz cuadrada de x ( $x \geq 0$ )        |
| <code>Math.pow(x, y)</code>    | $x^y$                                    |
| <code>Math.exp(x)</code>       | $e^x$                                    |
| <code>Math.log(x)</code>       | logaritmo natural ( $\ln(x)$ , $x > 0$ ) |
| <code>Math.round(x)</code>     | entero más próximo a x (long)            |
| <code>Math.ceil(x)</code>      | entero más pequeño $\geq x$ (double)     |
| <code>Math.floor(x)</code>     | entero más próximo $\leq x$ (double)     |
| <code>Math.abs(x)</code>       | valor absoluto de x                      |
| <code>Math.max(x, y)</code>    | valor mayor de x y y                     |
| <code>Math.min(x, y)</code>    | valor menor de x y y                     |
| <code>Math.sin(x)</code>       | seno de x (x en radianes)                |
| <code>Math.cos(x)</code>       | coseno de x (x en radianes)              |
| <code>Math.tan(x)</code>       | tangente de x (x en radianes)            |
| <code>Math.asin(x)</code>      | arco seno de x                           |
| <code>Math.acos(x)</code>      | arco coseno de x                         |
| <code>Math.atan(x)</code>      | arco tangente de x                       |
| <code>Math.atan2(y, x)</code>  | arco cuya tangente es y/x                |
| <code>Math.toRadians(x)</code> | convierte x grados a radianes            |
| <code>Math.toDegrees(x)</code> | convierte x radianes a grados            |

## 21.8 Cadenas

Las cadenas en Java son secuencias de *caracteres Unicode*. Java no tiene un tipo cadena incorporado. Sin embargo, existe en la biblioteca estándar de Java, una clase denominada *String* que está disponible cuando se programa en Java. Objetos de tipo *String* son cadenas de caracteres que se escriben dentro de dobles comillas. Cada cadena entrecomillada es una instancia de la *clase String*.

```
String a = ""; // cadena vacía
String saludo = "Hola";
```

Se puede declarar una variable de tipo cadena.

```
String ciudad;
ciudad = "Cazorla"
```

Se pueden juntar declaración y asignación.

```
String ciudad = "Cazorla" ;
```

Se puede visualizar el objeto representado por la variable cadena, mediante

```
System.out.println(ciudad);
```

### Concatenación de cadenas

Java, al igual que la mayoría de los lenguajes de programación, permite usar el operador `+` para unir (concatenar) dos cadenas.

#### Ejemplo

```
String nombre = "Sierra de Horche";
String frase;
frase = nombre + " es muy bella";
System.out.println(frase);
```

Se visualiza la frase

```
Sierra de Horche es muy bella
```

Cuando se concatena una cadena, con un valor que no es una cadena, el valor se convierte a cadena y se concatena. Por ejemplo:

```
int edad = 19;
String nombre = "Paloma" + edad;
```

hacen que *nombre* se convierta en la cadena *Paloma19*. Esta característica se utiliza con frecuencia en sentencias de salida. Por ejemplo:

```
System.out.println("La solución es " + resultado);
```

y si *resultado* es 44, se imprime:

```
La solución es 44
```

### Las cadenas son inmutables

En Java, un objeto de tipo *String* es un *objeto inmutable* lo que significa que los caracteres del objeto de *String* no se pueden cambiar. No existe ningún método que cambie el valor de un objeto, como "Paloma" y en caso de desear cambiarla deberá recurrir a concatenar la subcadena que desea reemplazar.

```
String pueblo = "Lupiana";
pueblo = "Pueblo de " + pueblo;
```

La sentencia de asignación ha cambiado la variable *pueblo* a *Pueblo de Lupiana*.

También se pueden realizar los cambios mediante el método *substring* y la clase *StringBuffer* que tiene métodos para modificar sus objetos cadena.

## Subcadenas de la clase `String`

El método `substring` extrae una subcadena de una cadena. Las sentencias

```
String saludo = "Hola Mackoy";
String cad = saludo.substring(0, 4);
 ↑ ↑
 primera primera
 posición posición
 de copia de no copia
```

crean la subcadena, de la posición 0, carácter más a la izquierda, a la posición 3 inclusive; es decir la cadena `cad = "Hola"`. La figura 21.2 muestra los números de posición de la subcadena.

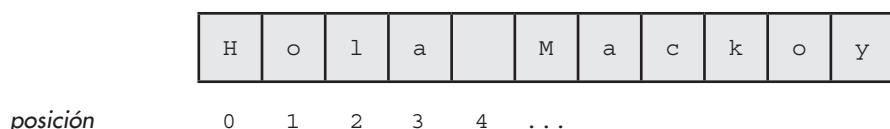


Figura 21.2 Posiciones de la cadena.

Una ventaja añadida a los índices del método `substring` es que permite encontrar fácilmente la longitud de la subcadena. La cadena `cad.substring(n, m)` tiene siempre la longitud  $m-n$ .

### Ejemplo

La subcadena `Hola` de longitud 4, es decir  $m-n = 4-0 = 4$

## Longitud y comparación de cadenas

El método `length ( )` devuelve el número de caracteres de un objeto `String` como un valor de tipo `int`.

### Ejemplo

```
String saludo = "hola Mortimer";
System.out.println("La longitud es " + saludo.length ());
```

Después de la ejecución de las sentencias anteriores se visualiza `La longitud es 13`.

El método `length` se utiliza para encontrar el número de caracteres de una cadena; se puede utilizar en cualquier parte que se pueda utilizar un valor de tipo `int`, como se ha hecho en el ejemplo anterior.

Otra operación muy usual cuando se trabaja con cadenas, es comparar dos cadenas para ver si son iguales o distintas. El método `equals` devuelve `true` si son iguales y `false` en caso contrario.

### Sintaxis

```
boolean equals(unacadena);
devuelve true si el objeto de la cadena que llama y unacadena son iguales.
```

### Ejemplo

```
s.equals(t) devuelve true si las cadenas s y t son iguales; false en caso contrario.
s, t variables cadena o constantes cadena.
"Hello".equals(saludo) sentencia válida.
```

Otras comparaciones realizadas con `equals`:

```
String saludo = "Hola";
saludo.equals("Hola"); devuelve true
```

```
saludo.equals("Adios"); devuelve false
saludo.equals("hola"); devuelve false ya que Hola y hola no son iguales
```

Hola y hola no son iguales ya que una cadena comienza con una letra mayúscula y otra empieza con una letra minúscula.

El método `equalsIgnoreCase` permite comparar dos cadenas y ver si son idénticas sin considerar la diferencia entre letras iguales mayúsculas y minúsculas.

### Sintaxis

```
boolean equalsIgnoreCase(unaCadena);
"Hola".equalsIgnoreCase("hola") devuelve true
String nombre = ";flanagan!";
nombre.equalsIgnoreCase(";Flanagan!"); devuelve true
```

### Precaución

No utilizar el operador `==` para comprobar si dos cadenas son iguales. En caso de hacerlo, el operador determina si las cadenas se almacenan en la misma posición.

Normalmente, si las cadenas están en la misma posición, deben ser iguales; pero es posible almacenar múltiples copias de cadenas idénticas en sitios diferentes.

### Ejemplo

```
String nombre = "Pedro";
if (nombre == "Pedro")
...
// probablemente es false

if (nombre.substring(0,3) == "Ped")
...
// probablemente es false
```

### Métodos de la clase String

La clase `String` tiene un número de métodos muy útiles que se pueden utilizar para aplicaciones de procesamiento de cadenas. Además, existen cientos de clases en las bibliotecas estándar, con muchos más métodos. Por consiguiente, es esencial que se familiarice con la documentación API en línea que le permite examinar todas las clases y métodos de la biblioteca estándar. La documentación API es parte del JDK, está en formato HTML en la dirección

<http://java.sun.com/javase/reference/api.jsp>

se encuentra la especificación API

Java Platform, Standards Edition 6 API Specification

La clase `String` de Java contiene más de 50 métodos. La clase `java.lang.String` pertenece al paquete `java.lang`. A continuación se muestra una lista de métodos significativos.

- `char charAt(posicion)`

Devuelve el carácter del objeto cadena que se encuentra en `posicion`. Las posiciones se cuentan 0, 1, 2, etcétera.

### Ejemplo

```
String saludo = ";Hola Mackoy!";
saludo.charAt(0) devuelve ;
saludo.charAt(1) devuelve H
```

- `int compareTo (Unacadena)`

Compara el objeto cadena llamador y el argumento cadena para ver cuál viene antes en el orden lexicográfico. El orden lexicográfico son los caracteres ordenados según los códigos ASCII o Unicode. En estos códigos todas las letras mayúsculas y todas las letras minúsculas están en orden alfabético-numérico pero todas las letras mayúsculas preceden a las letras minúsculas. De este modo, la ordenación lexicográfica es la misma que la ordenación alfabética pero teniendo en cuenta mayúsculas y minúsculas. Si la cadena llamadora es alfabéticamente menor devuelve un valor negativo. Si las cadenas son iguales devuelve cero y si el argumento es alfabéticamente menor devuelve un número positivo.

### Ejemplo

```
String entrada = "albaricoque";
entrada.compareTo("margarita"); devuelve un número negativo
entrada.compareTo("albaricoque "); devuelve cero
entrada.compareTo("amnesia "); devuelve un número positivo
```

- `int compareToIgnoreCase (Unacadena)`

Igual que `compareTo` pero tratando las letras mayúsculas y las minúsculas como si fueran iguales (es decir, las letras mayúsculas se tratan igual que si fueran letras minúsculas al hacer la comparación). Por consiguiente, si ambas cadenas constan totalmente de letras, la comparación se realiza por estricto orden alfabético. Si la cadena llamadora está primero devuelve un valor negativo. Si las dos cadenas son iguales devuelve cero. Si el argumento está primero devuelve un número positivo.

### Ejemplo

```
String entrada = "albaricoque";
entrada.compareToIgnoreCase("margarita"); devuelve un número negativo
entrada.compareToIgnoreCase("Albaricoque "); devuelve cero
"Margarita".compareToIgnoreCase(entrada); devuelve un número positivo
```

- `boolean endsWith (Unacadena)`

Devuelve `true` si la cadena termina en `Unacadena`.

- `boolean equals (Unacadena)`

Devuelve `true` si el objeto cadena llamador y `Unacadena` son iguales. En caso contrario devuelve `false`.

- `boolean equalsIgnoreCase (Unacadena)`

Devuelve `true` si el objeto cadena llamador y `Unacadena` son iguales, considerando que las letras mayúsculas y minúsculas son iguales.

```
String nombre = "Alvaro";
nombre.equalsIgnoreCase("alvaro"); devuelve true
```

- `int indexOf (String cad, int indicefinal)`

Devuelve el índice (posición) de la primera ocurrencia de la cadena `cad` en el objeto cadena llamador que ocurre después de la posición `indicefinal`. Las posiciones se cuentan 0, 1, 2, 3, ... . Devuelve -1 si no encuentra `cad`.

```
String nombre = "Luis, Luis Mackoy";
nombre.indexOf("Luis",1); devuelve 6
nombre.indexOf("Luis",0); devuelve 0
nombre.indexOf("Luis",9); devuelve -1
```

- `int lastIndexOf (String cad)`

Devuelve el índice (posición) de la última ocurrencia de la cadena `cad` en el objeto cadena llamador. Las posiciones se cuentan 0, 1, 2, 3, ... . Devuelve -1 si no se encuentra `cad`.

- ```

String nombre = "Magina, Sierra Magina ";
nombre.indexOf("Magina",0);           devuelve 0
nombre.lastIndexOf("Magina");        devuelve 13

• int length( )
Devuelve la longitud de la cadena.

• boolean startsWith(String prefijo)
Devuelve true si la cadena comienza con prefijo.

• String substring(int inicioIndice)
Devuelve la subcadena del objeto cadena comenzando desde inicioIndice hasta el final de
dicho objeto. Las posiciones se cuentan 0, 1, 2, 3, ....
String prueba = "Mnopqrs";
prueba.substring(3)                 devuelve "pqrs"

• String substring(int inicio, int fin)
Devuelve la subcadena del objeto cadena que comienza en la posición inicio hasta la posición
fin-1.
String prueba = "Mnopqrs";
prueba.substring(3,6)               devuelve "pqr"

• String toLowerCase( )
Devuelve una nueva cadena que contiene todos los caracteres de la cadena original, con los
caracteres en mayúsculas convertidos a letras minúsculas.
String sierra = "Sierra de Lupiana";
sierra.toLowerCase( );              devuelve "sierra de lupiana"

• String toUpperCase( )
Devuelve una nueva cadena que contiene todos los caracteres de la cadena original, con los
caracteres letras minúsculas convertidos a letras mayúsculas.
String nombre = "Gerardo Cebollero";
nombre.toUpperCase( );             devuelve "GERARDO CEBOLLERO"

• String trim( )
Devuelve una nueva cadena eliminando todos los espacios en cabeza y cola de la cadena original.
String miRio = " Rio Magdalena ";
miRio.trim( );                   devuelve "Rio Magdalena"

```

21.9 Códigos de caracteres ASCII y Unicode

El código de caracteres por excelencia siempre ha sido el código *ASCII* (apéndice B). El conjunto de caracteres *ASCII* es una lista de todos los caracteres utilizados normalmente en los teclados de las computadoras (inglés, español, francés, catalán, etc.), más unos pocos caracteres especiales. En esta lista a cada carácter se le asigna un número de modo que los caracteres se pueden almacenar en función del correspondiente número o código.

Java y muchos otros lenguajes de programación utilizan el conjunto de caracteres *Unicode*. El código de caracteres *Unicode* incluye el conjunto de caracteres ASCII más muchos otros caracteres utilizados en lenguajes con alfabetos diferentes al inglés, español, etcétera.

A recordar

El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode.

La ventaja del conjunto de caracteres *Unicode* es que facilita el manejo de otros lenguajes distintos al inglés, español, etc. Por ejemplo, se pueden utilizar letras del alfabeto griego o árabe.

21.10 Entrada y salida

En este apartado se tratan la salida a pantalla y la entrada del teclado, **E/S de consola**. La nueva versión 5.0 y posteriores incorpora la clase `Scanner` definida en el paquete `java.util` muy adecuada para entrada por consola.

Salida a la consola

Java utiliza paquetes y *sentencias de importación* (`import`). Los paquetes son biblioteca de clases Java. Las sentencias de importación ponen disponibles las clases del paquete en los programas. Existen dos formas de producir salidas a la pantalla: 1) `System.out.println` y 2) `System.out.printf`.

- `System.out.println`

Ya se ha utilizado anteriormente `System.out.println` para salida a pantalla.

`System.out` es un objeto que es una parte del lenguaje Java y `println` es un método invocado por ese objeto. Cuando se utiliza `System.out.println` para salida, los datos se proporcionan con un argumento entre paréntesis y la sentencia termina con un punto y coma.

Sintaxis

```
System.out.println(elemento_1+elemento_2+ ... + elemento_n)
```

Ejemplo

```
System.out.println("Bienvenido a Carchalejo");
System.out.println("Presupuesto = " + cantidad + " euros");
```

- `System.out.print`

La única diferencia entre `System.out.println` y `System.out.print` es que con `println` la siguiente salida salta a una nueva línea, mientras que con `print` la siguiente salida se sitúa en la misma línea.

Ejemplo

```
System.out.print("Veracruz ");
System.out.print(" Puerto La Cruz");
System.out.println(" y ");
System.out.print("Playa del Cura ");
```

producen la siguiente salida

```
Veracruz Puerto La Cruz y
Playera del cura
```

A recordar

```
System.out.println(...); equivale a
System.out.print(... + "\n");
utilizando el carácter de escape \n (salto de línea)
```

Salida formateada con `printf`

En la versión 5.0, Java incluyó un método denominado `printf`, inspirado en la función clásica de la biblioteca C, que se puede utilizar para producir la salida en un formato específico. La sentencia `System.`

`out.printf` tiene un primer argumento cadena, conocida como **especificador de formato** y el segundo argumento es el valor de salida en el formato establecido.

```
System.out.printf("%8.3f", x);  
imprime x con una anchura de campo de 8 caracteres y una precisión de 3 caracteres.
```

Ejemplo

```
double precio = 25.4;  
System.out.printf("$");  
System.out.printf("%6.2f", precio);  
System.out.printf(" unidad");
```

Al ejecutarse las sentencias anteriores visualiza \$25.40 unidad. El dato con un ancho de 6 caracteres y 2 caracteres de precisión.

```
double x = 10000.0/3.0;  
System.out.printf("%9.3f", x);
```

Al ejecutarse las sentencias anteriores visualiza x con una anchura de 9 caracteres y una precisión de 3 caracteres (un espacio en blanco a la izquierda y ocho caracteres):

3333.333

Cada uno de los *especificadores de formato* que comienza con un carácter % se reemplaza con el argumento correspondiente. El carácter de conversión con el que termina un especificador de formato indica el tipo de valor a formatear: f es un número en coma flotante, s es una cadena y d es un entero decimal.

Entrada

Los programas Java construidos hasta este momento no son interactivos ya que no reciben ninguna entrada de datos. En esta sección aprenderemos un método para leer la entrada del usuario.

En la versión 5.0, Java incluyó una clase para simplificar la entrada de datos por el teclado. Esta clase se llama `Scanner` y se conecta a `System.in`.

Para leer la entrada de la consola, se debe construir primero un objeto de `Scanner`, pasando simplemente el objeto `System.in` al constructor `Scanner` (más adelante se explicarán los constructores y el operador `new` en detalle).

```
Scanner entrada = new Scanner(System.in);
```

A continuación se pueden utilizar diferentes métodos de la clase `Scanner` para leer la entrada: `nextInt` o `nextDouble` leen el siguiente entero y siguiente valor de coma flotante.

```
System.out.print("Introduzca cantidad: ");  
int cantidad;  
cantidad = entrada.nextInt();  
System.out.print("Introduzca precio: ");  
double precio = entrada.nextDouble();
```

Cuando se llama a uno de los métodos anteriores, el programa espera hasta que el usuario teclea un número y pulsa la tecla *Intro (Enter)*.

El método `nextLine` lee una línea de entrada.

```
System.out.print("¿ Cual es su nombre?");  
String nombre;  
nombre = entrada.nextLine();
```

El método `next` se emplea cuando se desea leer una palabra sin espacios.

```
String apellido = entrada.next();
```

La clase `Scanner` está definida en el paquete `java.util`; en Java siempre que se utiliza una clase que no está definida en el paquete básico `java.lang` se necesita utilizar una directiva `import`. La primera línea que se pone cerca del principio del archivo, indica a Java dónde encontrar la definición de la clase `Scanner`:

```
import java.util.Scanner
```

Esta línea significa que la clase Scanner está en el paquete `java.util`; `util` es la abreviatura de *utility* (*utilidad* o *utilería*), pero en código Java siempre se utiliza la abreviatura `util`. Un paquete, como ya conoce el lector, es una biblioteca de clases. La sentencia `import` pone disponible la clase dentro del programa.

```
import java.util.Scanner;

/**
 * Este programa muestra la entrada por consola
 * y ha sido creado el 24 de marzo
 */
public class EntradaTest
{
    public static void main(String [ ] args)
    {
        Scanner entrada = new Scanner(System.in);
        // obtener la primera entrada
        System.out.print("¿ Cual es su nombre? ");
        String nombre = entrada.nextLine ( );
        // leer la segunda entrada
        System.out.print("¿ Cual es su edad? ");
        int edad = entrada.nextInt( );
        // visualizar salida
        System.out.println("Buenos días " + nombre +
                           "; años " + edad);
    }
}
```

Sintaxis de entrada de teclado utilizando Scanner

- Hacer disponible la clase `Scanner` para utilizarla en su código. Incluir la siguiente línea al comienzo del archivo que contiene su programa.

```
import java.util.Scanner;
```

- Antes de introducir algo por teclado se debe crear un objeto de la clase `Scanner`.

```
Scanner nombreObjeto = new Scanner(System.in);
nombreObjeto es cualquier identificador Java (no palabra reservada).
Scanner teclado = new Scanner(System.in);
```

- Los métodos `nextInt`, `nextDouble` y `next` leen un valor de tipo `int`, un valor de tipo `double` y una palabra, respectivamente.

Sintaxis

```
variable_int = nombreObjeto.nextInt( );
variable_double = nombreObjeto.nextDouble( );
variable_cadena = nombreObjeto.next( );
variable_cadena = nombreObjeto.nextLine( );
```

Ejemplo

```
int edad;
edad = teclado.nextInt( );
double precio;
precio = teclado.nextDouble( );
String rio;
rio = teclado.next( );
```

21.11 Flujo de control

Java, al igual que cualquier lenguaje de programación, soporta sentencias condicionales y bucles (lazos) para determinar el flujo de control.

Las construcciones de flujo de control de Java son idénticas a C y C++ con algunas excepciones. No existe la “fatídica” `goto` pero existe una versión de `break` con etiqueta que se puede utilizar para salir o interrumpir un bucle anidado, posible uso de “`goto`” en C. Una novedad que se añadió a Java 5.0 y versiones posteriores, es la incorporación de una variante del bucle `for`, “`for each`”, que no tiene ninguna sentencia similar en C y C++, y que es similar al bucle `for each` del lenguaje C#.

Bloques de sentencias

Un *bloque* o una *sentencia compuesta* es cualquier número de sentencias simples que están delimitadas por una pareja de paréntesis, tipo llave. Los bloques definen el ámbito de sus variables. Los bloques pueden estar *anidados* dentro de otro bloque.

Sintaxis de bloque de sentencias

```
{
    sentencia1
    sentencia2
    ...
}
```

Ejemplo

```
{
    double salarioNeto = salarioBruto - impuestos;
    salario = salarioNeto;
}
```

Un ejemplo dentro del método `main`.

```
public static void main(String [ ] args)
{
    int m;
    ...
    {
        int n;
        ...
    } // n está definido en este bloque
}
```

Aunque en C++ es posible redefinir una variable en el interior de un bloque anidado, esta característica puede ser una fuente de errores de programación. Por esta razón, Java no permite la redefinición de una misma variable en dos bloques anidados.

Ejemplo

Las siguientes sentencias dentro de un bloque anidado producirán un error en la compilación.

```
public static void main(String[ ] args)
{
    int m;
    ...
    {
        int m; // error, no se puede redefinir m en un bloque
        int n;
        ...
    }
}
```

Sentencias condicionales

Las sentencias condicionales en Java son: `if`, `if-else` y `switch`.

Sintaxis sentencia `if`

```
if (condición)
    sentencia
```

condición es una expresión lógica (booleana)
sentencia debe estar encerrada por {} si es
bloque de sentencias

Sintaxis sentencia `if-else`

```
if (condición)
    sentencia1
else
    sentencia2
```

Ejemplo

```
if (ventas >= objetivo)
{
    rendimiento = "Éxito";
    bono = 0.005 * (ventas - objetivo) + 400;
}
else
{
    rendimiento = "Fracaso";
    bono = 0;
}
```

Sentencias `if-else` multicamino

Sintaxis

```
if (condición1)
    sentencia1
else if (condición2)
    sentencia2
.
.
.
else if (condiciónn)
    sentencian
else
    sentenciax
```

Ejemplo

```
if (numDeAlumnos < 100)
    System.out.println("Menos de 100 alumnos");
else if (numDeAlumnos < 500)
    System.out.println("Entre 100 y 500 alumnos");
```

```
else if (numDeAlumnos < 1000)
    System.out.println("Entre 500 y 1000 alumnos");
else
    System.out.println("Más de 1000 alumnos");
```

Sentencia switch

Java tiene una sentencia `switch` idéntica a las sentencias `switch` de C y C++.

Sintaxis

```
switch (condicion)
{
    case etiqueta_1
        sentencia_1
        ...
        break;
    case etiqueta_2
        sentencia_2
        ...
        break;
    .
    .
    .
    case etiqueta_n
        sentencia_n
        ...
        break;
    default:
        sentencias
        ...
}
```

Ejemplo

```
int nivel
...
switch (nivel)
{
    case 1: nivelProfesional = "Amateur";
    break;
    case 2: nivelProfesional = "Intermedio";
    break;
    case 3: nivelProfesional = "Profesional";
    break;
    default: nivelProfesional = "Desconocido";
}
```

Precaución

La sentencia `switch` es muy propensa a la producción de errores debido a que en los casos de selección de múltiples alternativas, si se olvida añadir un sentencia `break` al final de una alternativa, la ejecución fallará, aunque el compilador no emitirá un mensaje de error y se irá a la siguiente alternativa. Por esta razón debe extremar las precauciones en el uso de `switch`.

Bucles (Lazos)

Los mecanismos de bucles o sentencias repetitivas o iterativas son similares a sus equivalentes de otros lenguajes de programación. Las tres sentencias Java de bucles son: `while`, `do-while` y `for`. Como ya se ha comentado anteriormente, a partir de Java 5 se incorpora una variante del bucle `for`, que es una extensión o mejora del bucle `for`, y denominada “`for each`”.

Sentencia `while`

La sentencia `while`, recordemos, ejecuta una sentencia (o un bloque de sentencias) mientras una *condición* es verdadera.

Sintaxis

```
1. while (condicion)
   sentencia

2. while (condicion)
{
   sentencia_1
   sentencia_2
   .
   .
   .
   sentencia_n
}
```

Ejemplo

```
while (saldo < objetivo)
{
    anyos++;
    double interes = saldo * tasa /100;
    saldo = saldo + interes;
}
System.out.println(anyos + ", años.");
```

Recordemos que un bucle `while` realiza la comprobación en la parte superior del bucle; por consiguiente, puede presentarse alguna situación en la cual el bucle no se ejecutará. Si se desea asegurar la ejecución del bloque al menos una vez, se necesitará desplazar la condición o expresión lógica al final del bloque. Esta característica se realiza en el bucle `do-while`.

Sentencia `do-while`

Sintaxis

```
do
{
   sentencia_1
   sentencia_2
   .
   .
   .
   sentencia_n
} while (condicion)
```

Ejemplo

```
int contador = 10;
do
{
    System.out.println("Hola amigo");
    contador = contador - 1;
} (contador > 0);
```

Sentencia for

Sintaxis

```
for (inicialización; condición; actualización)
    sentencia;
```

Ejemplo

```
1. int siguiente, suma = 0;
   for (siguiente = 0; siguiente <= 10; siguiente++)
   {
       suma = suma + siguiente;
       System.out.println("suma " + siguiente + " es " + suma);
   }
2. for (int i = 1; i <= n; i++)
{
    double interes = saldo * tasa/100;
    saldo += interes;
}
```

Regla

Las sentencias de control condicionales o selectivas y repetitivas o iterativas en Java funcionan de modo similar a C/C++

Sentencias break y continue

Si se desea alterar el flujo de control de los bucles `for`, `while` y `do-while` existen dos métodos. Estos dos métodos consisten en insertar una sentencia `break` o una sentencia `continue`. La sentencia `break` termina el bucle. La sentencia `continue` termina la iteración en curso del cuerpo del bucle. Las sentencias anteriores se pueden utilizar con cualquiera de las sentencias del bucle de Java.

Sintaxis

```
break;
break etiqueta;
continue;
```

Ejemplo

```
while (mes <= 100)
{
    saldo = saldo + cuota;
    double interes = saldo * tasaInteres/100;
    saldo += interes;
    if (saldo >= objetivo) break;
    mes++;
}
```

A diferencia de C++, Java ofrece una sentencia `break` etiqueta que permite romper el flujo de control de bucles anidados.

Ejemplo

```
bucleExterno: // etiqueta
do
{
    ...
    while (siguiente >= 0)
    {
        if (siguiente < -559)
            break bucleExterno;
        ...
    }
    ...
    problema = ...
} while (problema.equalsIgnoreCase("si"));
```

En el ejemplo anterior `bucleExterno` etiqueta el bucle externo, que es un bucle `do`. Si el valor de la variable `siguiente` es menor que `-559`, entonces se ejecuta la sentencia `break` y termina el bucle `do`.

Nota

Es posible aplicar una etiqueta a cualquier sentencia, incluso en una sentencia `if` o en un bloque.

```
etiqueta:
{
    ...
    if (condicion) break etiqueta; // salida del bloque
    ...
}
```

Ejemplo de `continue`

```
Scanner teclado = new Scanner(System.in);
for (int contador = 1; contador <= 500; contador++)
{
    System.out.print("Introduzca un número positivo");
    int n;
    n = teclado.nextInt();
    if (n < 0) continue;
    suma += n; // no se ejecuta si n < 0
}
```

Método `exit()`

La sentencia `break` termina un bucle o la sentencia `switch`, pero no termina el programa; en caso de desear terminar el programa, Java tiene el método `exit` definido en la clase `System` (clase predefinida que se incorpora automáticamente) que termina un programa tan pronto se invoca.

Ejemplo

```
System.out.println("Introduzca un número negativo");
int numNegativo = teclado.nextInt();
if (numNegativo >= 0)
{
    System.out.println(numNegativo + "no es negativo");
    System.out.println("programa abortado");
    System.exit(0);
}
```

El bucle "for each"

A partir de Java 5.0 se ha introducido una construcción de bucles muy potente que permite al programador iterar a través de cada elemento de un *array* o *arreglo*, así como otras colecciones de elementos sin tener que preocuparse por los valores de los índices del bucle. El bucle `for each` establece una variable dada a cada elemento de la colección y a continuación ejecuta las sentencias del bloque.

Sintaxis mejorada del bucle for

```
for (variable : colección)
    sentencia;
```

La expresión *colección* debe ser un arreglo (array) o un objeto de una clase que implemente la interfaz `Iterable`, como lista de arreglos, `ArrayList`, etc.

Ejemplo

Visualizar cada elemento de un arreglo en una lista independiente.

```
int [ ] m = new int[100]; // array
...
for (int elemento : m)
    System.out.println(elemento);
```

El bucle se lee así: “para (for) cada (each) elemento de *m*”. Los diseñadores de Java consideraron para nombrar el bucle, en primer lugar `foreach` e `in`, pero al final optaron por continuar con el código antiguo y la palabra reservada `for` para evitar conflictos de nombres de métodos o variables ya existentes como `System.in` ya utilizado en este capítulo.

El efecto anterior del recorrido de colecciones se puede conseguir con un bucle `for` estándar, aunque el bucle “`for each`” es más conciso y menos propenso a errores. Un bucle equivalente al anterior es:

```
for (int j = 0; j < m.length; j++)
    System.out.println(m[ j ]);
```

Nota

- El “`for each`” se puede interpretar como “por cada valor de ... hacer las siguientes acciones”.
- La variable del bucle “`for each`” recorre los elementos de la colección o el arreglo (*array*) y no los valores de los índices.

El bucle “`for each`” es una mejora sustancial sobre el bucle tradicional en el caso de que se necesita procesar todos los elementos de una colección. Posteriormente se amplía el concepto del bucle “`for each`” al tratar los arreglos.

21.12 Arreglos (arrays)

Recordemos que un arreglo (*array*) es una estructura de datos que almacena una colección de valores del mismo tipo. En esencia, un arreglo es una colección de variables del mismo tipo. Se accede a través de un índice entero.

```
double[ ] puntos;
int [ ] a;
```

En Java, la sentencia anterior solo declara la variable *a* o la variable *puntos*. Se debe utilizar el operador `new` para crear el arreglo.

```
int [ ] m = new int[50]; // arreglo de 50 elementos de tipo entero
double [ ] puntos = new double[10]; // arreglo de 10 elementos
```

Regla

Las variables de tipo arreglo se pueden definir con dos formatos diferentes:

1. int a [];
2. int [] a;

Sintaxis

Se declara un nombre de un arreglo y se crea un arreglo de un modo similar a la creación y proceso de dar nombre a objetos de clases.

```
tipo_base [ ] nombre_arreglo = new tipo_base[longitud];
```

Ejemplos

```
char [ ] linea = new char[80];
double [ ] calificaciones = new double[400];
Alumno [ ] listas = new Alumno[120];
```

Los arreglos linea y calificaciones son de tipo char y double, mientras que listas es una arreglo de tipo Alumno (Alumno es una clase).

La longitud de un arreglo (array) o número de elementos del mismo se calcula utilizando la sentencia arreglo.length.

Ejemplo

```
int [ ] a = new int[100];
for (int i = 0; i < 100; i++)
    a[ i ] = i; // el arreglo se rellena con enteros de 0 a 99
for (int i = 0; i < a.length; i++)
    System.out.println(a[ i ]);
```

Regla

- Una vez que se ha creado un arreglo no se puede cambiar su tamaño, aunque sí, lógicamente, sus elementos individuales.
- Si se necesita expandir el tamaño de un arreglo mientras se ejecuta un programa, se debe utilizar una estructura de datos diferentes que en Java 5 y versiones posteriores se denomina *lista de arreglos*.

Acceso a los elementos de un arreglo

El acceso a los elementos de un arreglo se realiza de igual modo que en otros lenguajes de programación.

```
arreglo[índice]
```

Ejemplo

```
nombres[3];
lista[5]
a[a.length - 1];
```

Arreglos y objetos

En Java un arreglo (array) es considerado un objeto. Esto significa que se pueden considerar los tipos arreglos como clases. Sin embargo, aunque un arreglo de tipo double[] es una clase, la sintaxis para crear un objeto arreglo es un poco diferente. Para crear un arreglo, se utiliza la sintaxis:

```
double [ ] m = new double[10];
```

Se puede considerar la expresión `new double[10]` como una invocación de un constructor que utiliza una sintaxis no estándar.

Como ya se ha visto anteriormente, cada arreglo tiene una variable de instancia denominada `length`. Al igual que sucede con cualquier otro tipo de clases, las variables arreglo contienen direcciones de memoria, conocidas en Java como *referencias* y por consiguiente, los tipos arreglo son tipos referencia.

Advertencia

- Existe cierto debate entre expertos de Java sobre la consideración de que los tipos arreglos son o no clases; sin embargo, existe práctica unanimidad, en que los arreglos son, en sí mismos, objetos.
- Los tipos arreglos son tipos referencia y una variable de un tipo arreglo contiene la dirección donde se almacena el objeto arreglo en memoria. Esta dirección de memoria se denomina *referencia* al objeto arreglo.

Inicialización de arreglos y arreglos anónimos

Java tiene un formato especial abreviado para crear un objeto arreglo e inicializar el mismo, sin recurrir a la llamada a `new`. La sintaxis es:

```
int [ ] numPrimos = {1,3,5,7,11,13,17,19};
```

En Java, existe la posibilidad de crear objetos y clases anónimas, y en particular *arreglos anónimos*. Cuando una expresión como

```
new DemoClase("Paloma",19);
```

no se asigna a una variable se conoce como un *objeto anónimo*. La expresión crea una referencia a un objeto de la clase. Se denomina *anónimo* debido a que el objeto no se asigna a una variable para ser referenciada por su nombre.

Es el caso de un arreglo, se puede inicializar un *arreglo anónimo* con la siguiente declaración

```
new int [ ] {1,3,5,7,9};
```

Esta expresión asigna un arreglo nuevo y lo rellena con los valores internos a las llaves. Cuenta el número de valores iniciales y establece el tamaño correspondiente del arreglo.

Se puede utilizar la sintaxis para reinicializar un arreglo sin crear una nueva variable. Así, por ejemplo, las declaraciones

```
int [ ] anonimo = {1,3,5,7,11,13,17,19};  
numPrimos = anonimo;
```

que equivale a la expresión

```
numPrimos = new int [ ] {1,3,5,7,11,13,17,19};
```

Arreglos multidimensionales

Los arreglos multidimensionales, como conoce el lector, utilizan más de un índice para acceder a los elementos de dicho arreglo.

La declaración de un arreglo de dos dimensiones en Java es muy sencilla:

```
char [ ][ ]pagina = new char[50][80];
```

o su equivalente, no abreviado:

```
char [ ][ ]pagina;  
pagina = new char[50][80];
```

El arreglo `pagina` tiene dos índices, el primer índice va de 0 a 49 y el segundo índice de 0 a 79. Las variables indexadas tienen dos índices:

```
pagina[5][8] o bien, pagina[15][18]
```

Sintaxis

La declaración y creación de un arreglo multidimensional es:

```
tipo_base [ ]...[ ] nombre_arreglo = new tipo_base[lon_1]...[lon_n];  
donde lon_1, lon_2, ... son las longitudes de cada dimensión.
```

Ejemplos

1.

```
double [ ] [ ] saldo  
saldo = new double[ANIOS] [INTERESES];
```
2. En el caso de conocer los elementos del arreglo, se puede utilizar una notación abreviada.

```
int [ ] [ ] cuadroMagico =  
{  
    {4,15,20,14},  
    {5, 7, 8,19},  
    {9,18, 7, 6},  
    {2, 4, 6, 8}  
};
```

Arreglos irregulares o triangulares

En Java es relativamente fácil crear arreglos irregulares, de modo que filas diferentes puedan tener distintos números de columnas, al contrario de los arreglos regulares en los que cada fila tiene igual número de columnas. Así, por ejemplo, se pueden crear del siguiente modelo del binomio de Newton.

$$\begin{array}{cccccccccc} & & & & & 1 & & & & \\ & & & & 1 & & 1 & & & \\ & & & 1 & & 2 & & 1 & & \\ & & 1 & & 3 & & 3 & & 1 & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 & & & & \end{array}$$

Ejemplo

Crear un arreglo regular num de 3 filas y 5 columnas. Su sintaxis es

```
double [ ] [ ] num = new double[3] [5];
```

Esta expresión es equivalente a

1.

```
double [ ] [ ] num;
```
2.

```
num = new double [3] [ ];
```
3.

```
num[0] = new double [5];
```
4.

```
num[1] = new double [5];
```
5.

```
num[2] = new double [5];
```

La línea 2 declara un arreglo con 3 entradas, cada una de las cuales puede ser un arreglo de tipo double a su vez de cualquier longitud. Las líneas 3, 4 y 5 crean arreglos de tipo double de longitud 5. Se ha creado un arreglo de tipo base double con tres filas y cinco columnas.

Ejemplo

Si se desea hacer que cada arreglo tenga una longitud diferente, se puede crear un *arreglo irregular*. Las siguientes declaraciones crean un arreglo irregular num1 en el que cada fila tiene una longitud (columnas) diferente.

```
double [ ] [ ] num1;  
num1 = new double [3] [ ];  
num1[0] = new double [5];
```

```
num2[1] = new double [6];
num3[2] = new double [7];
```

Listas de arreglos (ArrayList)

Los arreglos son una construcción básica. En esta sección se explica la clase `ArrayList` que le permite reunir (coleccionar) objetos como si fueran tipos de datos básicos. La lista de arreglos ofrece dos ventajas:

- La lista de arreglos puede crecer y reducirse a medida que se necesita.
- La clase `ArrayList` proporciona métodos para muchas tareas comunes, como inserción y eliminar elementos.

La clase `ArrayList` es una clase genérica: `ArrayList<T>` reúne objetos de tipo genérico `T`.

Ejemplo

```
1. Cantidad [ ] numeros = {5.0,4.5,7.5,14.0};
   ArrayList<Cantidad> listaArticulos = new ArrayList<Cantidad>();
   Creación, e inicialización de una lista de arreglos Cantidad.
2. Definir una lista de arreglos de cuentas bancarias.
   ArrayList<CuentaBancaria> cuentas = new ArrayList<CuentaBancaria>();
```

Bucle for each

Java 5.0 incorporó un nuevo tipo de bucle `for`, ya estudiado con anterioridad, que puede iterar a través de todos los elementos de una colección, incluso aunque no haya índices de los elementos, como sucede con los arreglos. Este nuevo tipo de bucle se denomina *bucle for each o bucle for mejorado*. Estos bucles se utilizan muy eficientemente en las clases colección.

Sintaxis

```
for (tipo_base_arreglo elemento : nombre_arreglo)
  sentencia;
for (tipo_base_colección elemento : nombre_colección)
  sentencia;
```

Ejemplo

```
1. ArrayList<String> cadenas = new ArrayList<String>();
   for (String cd : cadenas)
     System.out.println(cd);
2. double [ ] b = new double[30];
   // llenar el arreglo
   for (double e : b)
     suma += e;
```

Regla

El bucle `for each` puede hacer su código mucho más claro y mucho menos propenso a errores. Si no se utiliza la variable con índice en un bucle `for` para una aplicación diferente a realizar bucles (ciclos) a través de los elementos de un arreglo, es preferible utilizar un bucle `for each`.

Ejemplo

El bucle

```
for (double elemento : m) // el array m tiene el tipo base double
  elemento = 0.0;
```

es preferible a

```
for (int i = 0; i < a.length; i++)
a[i] = 0.0;
```

Estos dos bucles realizan las mismas acciones, pero la segunda requiere un índice `i` para numerar los elementos del arreglo. La primera opción establece cada elemento del arreglo a 0.0 y significa: “Para cada elemento de `m` se pone a 0.0”.

Se puede utilizar también el bucle `for` mejorado para recorrer todos los elementos de un arreglo lista.

Ejemplo

El siguiente arreglo calcula el valor total de todas las cuentas de un banco.

```
ArrayList <CuentaBanco> cuentas = new ArrayList <CuentaBanco>();
double suma = 0.0;
for (CuentaBanco c : cuentas)
{
    suma = suma + c.leerSaldo();
}
```

Este bucle es equivalente al siguiente bucle ordinario:

```
double suma = 0.0;
for (int j = 0; j < cuentas.size(); j++)
{
    CuentaBanco c = cuentas.get(j);
    suma = suma + c.leerSaldo();
}
```

21.13 Applets

Los programas que se escriben en Java son de dos tipos: **aplicaciones** y **applets**. Una *aplicación* se ejecuta en el entorno local que constituye nuestra computadora; se caracteriza por tener una clase principal en la que se encuentra el método `main()`, que es el punto de entrada al programa desde el sistema operativo. Una aplicación usa todos los recursos de que dispone la computadora en que se ejecuta.

El segundo tipo de programas son los *applets*, que están pensados para que puedan navegar por la red de redes, descargarse desde cualquier computadora y ejecutarse. Es la máxima expresión de la portabilidad, “escribir una vez, ejecutar mil veces”.

Un *applet* es un tipo especial de programa Java que se incluye en una página HTML. La página HTML debe indicar al navegador cuáles *applets* cargar y dónde poner cada *applet* en la página web. Para utilizar un *applet*, se emplea una etiqueta que indica al navegador dónde obtener los archivos de clases y cómo se posiciona el *applet* en la página web (tamaño, posición, ...); el navegador, recupera, a continuación, los archivos de clases de internet (o de un directorio en la máquina del usuario) y se ejecuta automáticamente el *applet*.

Las expectativas que trajeron consigo los primeros *applets* nunca se han cumplido. En un principio, cuando se desarrollaron los *applets* se requería utilizar el navegador Hot Java de Sun para visualizar páginas web que contenían *applets*. Los *applets* de Java se hicieron realmente populares cuando Netscape incluyó una máquina virtual Java en su navegador *Navigator*. Desgraciadamente, desapareció *Navigator* y apareció Explorer de Microsoft, de modo que la solución no tuvo éxito y las propuestas de Microsoft no llegaban a soportar siempre las versiones actualizadas.

Para superar estos problemas, Sun lanzó una herramienta denominada los “Java Plug-in” que proporcionó mecanismos para conexión de una gran variedad de navegadores facilitándoles ejecutar *applets* de Java utilizando un entorno en tiempo de ejecución que Sun proporciona.

Recomendación

Para ejecutar los *applets* en un navegador se necesita instalar la versión actual del Java *Plug-in* y asegurarse que su navegador está conectado con el *Plug-in*. La descarga e información de la configuración se encuentra en: [//java.com](http://java.com)

Desde el punto de vista técnico un *applet* es muy simple: los usuarios descargarán *bytecode* de Java desde internet y lo ejecutan en sus propias máquinas. Para que se ejecute un *applet* es necesario que el navegador de internet interprete una página HTML, en la que se demanda la ejecución de la aplicación *applet*.

21.14 Creación de un applet

Para crear un *applet* al menos hay que definir una clase que herede de la clase *Applet* (paquete *java.applet*). Actualmente, desde Java 2, se recomienda que la clase creada herede de la clase *JApplet* (paquete *javax.swing*).

Un *applet* es una clase derivada de la clase *JApplet*, que a su vez deriva de la clase *Applet*. Normalmente los *applets* tienen componentes gráficos, que son objetos de clases agrupadas en el paquete *javax.swing*. Por ello se recomienda que todos los *applets* que se creen deriven¹ de *JApplet*. La figura 21.3 muestra la jerarquía de clases en la que se encuentra *JApplet*.

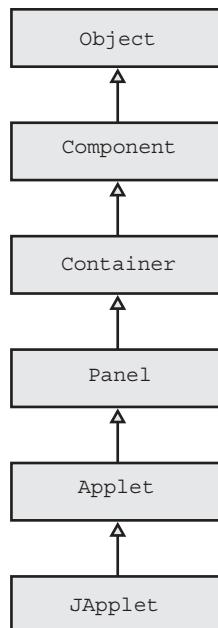


Figura 21.3 Jerarquía de Applet y JApplet.

Se crea un *applet* con un campo de texto.

```

import java.awt.*;
import javax.swing.*; // paquete con las clases de componentes
                      // gráficos, también JApplet
public class EjemApplet extends JApplet
{
    public void init ( )
    {
        JTextField a = new JTextField("Hola a todos, menos a uno");
        add(a);
    }
}
  
```

Ejemplo 21.1

¹ En el capítulo 23 se estudia la herencia en Java.

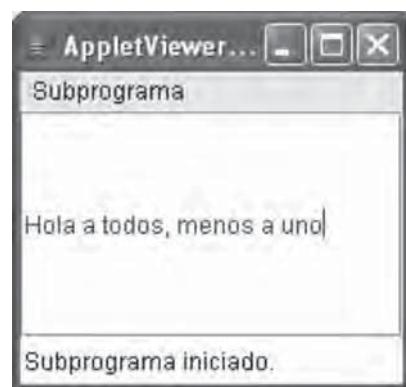


Figura 21.4 Visualización del applet del ejemplo 21.1.

La clase `JApplet` define el ciclo de vida de un *applet*. La ejecución de un *applet* la realiza el navegador empleado por el usuario (Netscape, Explorer, ...), o bien por la herramienta `appletviewer` suministrada en el kit del JDK, utilizada para probar los *applets*.



Ejemplo 21.2

Muestra la cadena "Hoy es el día de la Tierra" en un applet.

La clase `DiaTierraApplet` deriva² de la clase `JApplet`; esta se encuentra en el paquete `javax.swing`. La clase redefine el comportamiento del método `paint()` que escribe la cadena. Para escribir la cadena se llama al método `drawString()` de la clase `Graphics`. Cuando `appletviewer`, o un navegador, ejecuta el applet (incrustado en un documento HTML) llama al método `paint()` y le pasa el contexto gráfico, representado por `Graphics`.

```
import java.awt.Graphics;
import javax.swing.*;
public class DiaTierraApplet extends JApplet
{
    public void paint (Graphics g)
    {
        g.drawString ("Hoy es el día de la Tierra", 15,15);
    }
}
```

El método `drawString()` pinta la cadena a partir de las *coordenadas*, expresadas en pixeles (15,15).

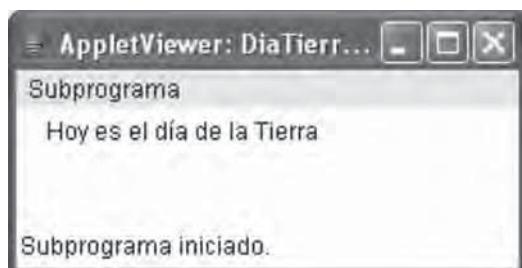


Figura 21.5 Visualización del applet del ejemplo 21.2.

² Java implementa la herencia o extensión de clases con la palabra `extends`. En el capítulo 23 se estudia la herencia.

Documento HTML para applet

El lenguaje HTML está formado por etiquetas que indican al explorador acciones a realizar. Para incrustar un *applet* se utiliza la etiqueta homónima `<applet>` (también `<object>`). Se puede escribir en mayúsculas o en minúsculas. Esta etiqueta incorpora atributos para especificar el nombre de la clase *applet*, el tamaño (ancho y alto) de la ventana y otras características. Por ejemplo:

```
<html>
  <head>
    <title>EstoEsUnApplet</title>
  </head>
  <body>
    <applet code ="EstoEsUnApplet.class" width="200" height="200">
    </applet>
  </body>
</html>
```

El navegador cuando analiza este HTML se encuentra la etiqueta `<applet>`, carga, del *path* actual, el *applet* y crea una página web de tamaño 200 * 200 pixeles en la que se ponen los elementos especificados en la clase.

Los atributos que obligatoriamente se deben especificar entre `<applet>` y `</applet>` son `code`, `width` y `height`. Con el atributo `code` se especifica el nombre del archivo con la clase *applet* y debe tener la extensión `.class`. Los atributos `width` y `height` especifican el tamaño, en pixeles, de la ventana en la que se visualiza el *applet*. Se puede redimensionar la ventana del *applet* con el método `resize ()` de `JApplet`; sin embargo, los navegadores no permiten cambiar el tamaño de la ventana. El cambio de tamaño solo se pone de manifiesto con `appletviewer`.

Otros atributos que se pueden utilizar en la etiqueta `<applet>`:

- **align:** especifica la alineación del *applet*. Alguna de las opciones son: `left`, `right`, `top`, `middle`, ...
- **archive:** se utiliza para especificar el nombre del archivo comprimido (`.jar`) donde se encuentran las imágenes, sonidos y otras clases necesarias para ejecutar el *applet*.
- **codebase:** indica el directorio base a partir del cual se encuentra el archivo `.class` con el *applet*. De modo predeterminado, asume el directorio del documento *HTML*.
- **name:** especifica el nombre del *applet*. Se utiliza cuando dos *applets* de una misma página se comunican entre sí.

A continuación se escribe el documento `MiQuiniela.html` para ejecutar el *applet* `MiQuiniela-Applet.class` (los archivos `.class` contienen el *bytecode* resultado de la compilación). La ventana que se crea es de 150*150 pixeles, el nombre que se le da es *Quiniela* y el directorio base (relativo al directorio donde se encuentra el `.html`) es *applets*.

```
<applet code =" MiQuinielaApplet.class "
        width="150"
        height="150"
        align=middle
        name="Quiniela"
        codebase="applet">
</applet>
```

21.15 Ciclo de vida de un applet

La clase `JApplet` hereda de la clase `Applet` cuatro métodos que determinan el ciclo de vida de un applet, son: `init ()`, `start ()`, `stop ()` y `destroy ()`. Estos métodos definidos en el API de Java no hacen nada, sin embargo son vitales en la ejecución del *applet* ya que son llamados automáticamente y pueden ser redefinidos en nuestra clase derivada de `JApplet`. A continuación se describe cada uno de estos métodos.

void init ()

Este método lo llama en forma automática el navegador una vez que ha cargado el objeto *applet* para su ejecución. Normalmente, en este método se sitúa código para procesar parámetros (*param*) que están en el documento *HTML*. Todas las acciones de inicialización se colocan en este método, como descargar una canción o un dibujo. Este método se llama una única vez en la ejecución del *applet*.

void start ()

Una vez que termina la ejecución del método *init* (), se invoca automáticamente a *start* (). Un *applet* es una página web, por consiguiente se puede tapar con otra página, o minimizar; cada vez que el usuario vuelve a la página del *applet* se llama a *start* (). El método *start* () de la clase *JApplet* está vacío, no hace nada; se puede redefinir en la clase derivada para comenzar la ejecución de “algo” relativo a la página que se está viendo, por ejemplo, empezar la animación, o iniciar el sonido de una sintonía.

void stop ()

Este método es llamado automáticamente cuando se oculta la página del *applet*. Un *applet* puede hacerse “no visible” por una acción del que usa la página, como por ejemplo minimizar, o ir “página atrás”, o cargar otra página. Cuando esto ocurre se llama al método *stop* (). La definición de *stop* () en *JApplet* no hace nada, está vacía; se puede redefinir en la clase derivada para lo que se considere oportuno, generalmente para detener acciones que se inician en *start* (), como detener una animación o parar la reproducción de una canción.

void destroy ()

Al cerrar la página web con el *applet* se llama automáticamente a *destroy* (). Lo normal es liberar los recursos, los hilos de ejecución, que todavía tenga asignado el *applet*. Hay que tener en cuenta que siempre que se cierra una página con *applet*, primero se llama a *stop* () y, a continuación a *destroy* ().

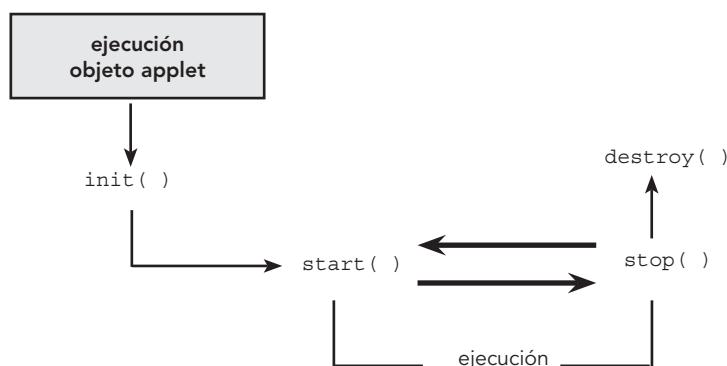


Figura 21.6 Ciclo de vida de un *applet*.

**Ejemplo 21.3**

Se construye un *applet* en el que se redefinen los cuatro métodos que intervienen en el ciclo vital de un *applet*.

init () pone un color inicial de fondo e inicializa un contador. *start* () incrementa el mismo contador y cambia el color del fondo. *paint* () también se redefine, con el fin de dibujar un rectángulo (*drawRect* ()) y un círculo (*drawArc* ()) cuyas dimensiones dependen del contador. *stop* () muestra por consola el valor del contador y lo incrementa. Por último, el método *destroy* () muestra en la consola el valor final del contador.

```

import java.awt.*;
import javax.swing.*;
public class GraficoApplet extends JApplet
{
  
```

```

Color color;
int cont= 0;
public void init( )
{
    color = new Color(192, 192, 192); // lightGray
    setBackground(color);
    cont++;
}
public void start ( )
{
    if (cont % 3 == 0)
        color = Color.BLUE;
    else if (cont % 3 == 1)
        color = Color.RED;
    else
        color = Color.MAGENTA;
    setBackground(color);
    ++cont;
}
public void stop( )
{
    System.out.println(" Contador = " + (++cont));
}

public void paint(Graphics g)
{
    g.drawRect(0+cont,0+cont, 20+4*cont, 40+5*cont);
    g.drawOval(0+cont,0+cont, 20+5*cont, 40+5*cont);
}
public void destroy( )
{
    System.out.println(" Fin del applet, Contador = " + (++cont));
}
}

```



Dibujar imágenes en un applet

Además de los cuatro métodos que constituyen la vida de un *applet*, hay otros métodos, propios o heredados, de la clase `JApplet` para visualización de imágenes o texto que son de interés para construir un *applet*.

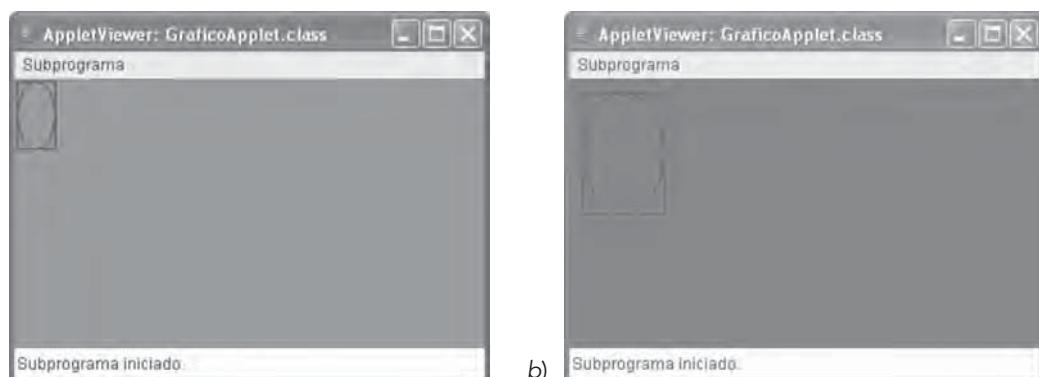


Figura 21.7 a) Imagen inicial del applet. b) Imagen del applet cuando contador = 10.

void paint (Graphics g)

El método `paint ()` es invocado automáticamente siempre que el *applet* se dibuja, o se deba volver a dibujar. No se programa una llamada a `paint ()`, sino que es el *applet* que le llama automáticamente, recibe el argumento *contexto gráfico* (`Graphics`). Una vez que `init ()` termina de ejecutarse y comienza la ejecución de `start ()`, se llama a `paint ()`. También, cuando el *applet* necesita repintarse. Por ejemplo, si el usuario ha minimizado la pantalla con el *applet*, y después la ha repuesto se produce una llamada a `paint ()`.

Con el *contexto gráfico* (`Graphics`) se pueden dibujar imágenes, texto, Las medidas de las imágenes y coordenadas siempre se hacen en pixeles. La coordenada `(0, 0)` es la esquina superior izquierda de la página con el *applet*.

void resize (int ancho, int alto)

El tamaño del *applet* se especifica en el documento HTML asociado. El método `resize ()` modifica el tamaño del *applet*; funciona perfectamente con *appletviewer*; sin embargo, con la mayoría de los navegadores no es aplicable para no interferir con sus sistemas gráficos.

void repaint ()

Se utiliza para cambiar la apariencia de un *applet*. Como no se puede llamar directamente a `paint ()` ya que no se dispone del contexto gráfico, la llamada a `repaint ()` pasa control al método `update ()`, el cual llama a `paint ()` pasando el contexto gráfico.

El método `update ()` borra cualquier dibujo de la página web del *applet* y después llama a `paint ()`.

El método `repaint ()` está sobrecargado con el fin de volver a dibujar un área del *applet*, delimitada por coordenadas `(x, y)` del origen, el ancho y el alto: `repaint (x, y, ancho, alto)`.



Ejemplo 21.4

Se construye un *applet* que cambia de tamaño y apariencia. Los cambios dependen del valor de una variable que aumenta de valor cada vez que se ejecuta el método `start ()`. Para que se ejecute `start ()`, minimizar el *applet* y, a continuación, abrirlo.

La ejecución de `start ()` incrementa a la variable `cont` en 1, y en 10 a las variables `ancho` y `alto`. Cuando `cont` es múltiplo de 2 (`cont % 2 == 0`) la llamada a `resize ()` ajusta el tamaño del *applet* al valor actual de `ancho` y `alto`. Cuando `cont` es múltiplo de 3 se pone el color rojo y se llama a `repaint ()`. Cuando `cont` es múltiplo de 5 la llamada a `resize ()` aumenta el tamaño del *applet*.

```
import java.awt.*;
import javax.swing.*;
public class AppletResize extends JApplet
{
    Color color;
    int cont;
    int ancho, alto;
    public void init ( )
    {
        color = Color.BLUE;
        setBackground (color);
        ancho = 60;
        alto = 40;
        cont = 1;
    }
    public void start ( )
    {
        ancho += 10;
        alto += 10;
        System.out.println("cont: " + cont + "; ancho" + ancho);
        if (cont % 2 == 0)
            resize (ancho,alto);
    }
}
```

```

else if (cont % 3 == 0)
{
    color = Color.RED;
    setBackground (color);
    repaint (0,0,60,40);
}
else if (cont % 5 == 0)
    resize (ancho+100, alto+150);
if (ancho > 150)
    init ( );
++cont;
}
public void paint (Graphics g)
{
    g.fillRoundRect (0, 0, ancho/3, alto/4, 10+cont, 5+2*cont);
    g.fillOval (ancho/3, alto/4, ancho/3, alto/3);
}
}

```

Clase Graphics

Graphics es una clase abstracta definida en el paquete `java.awt`, que proporciona el contexto gráfico. La clase dispone de métodos para dibujar todo tipo de elementos.

Los métodos de dibujo más interesantes de Graphics:

- `void drawString (String cad, int x, int y);` Escribe, pinta una cadena a partir de las coordenadas (x, y).
- `void drawLine (int x1, int y1, int x2, int y2);` Dibuja una línea entre los puntos (x1, y1), (x2, y2).
- `void drawRect (int x, int y, int ancho, int alto);` Dibuja un rectángulo. La posición de la esquina superior izquierda es (x, y).
- `void fillRect (int x, int y, int ancho, int alto);` Dibuja, llena un rectángulo con el color actual. La posición de la esquina superior izquierda es (x, y).
- `void fillRoundRect (int x, int y, int ancho, int alto, int arcoAncho, int arcoAlto);` Dibuja, llena un rectángulo redondeado con el color actual.
- `void drawOval (int x, int y, int ancho, int alto);` Dibuja una elipse a partir de la posición (x, y).
- `void drawArc(int x, int y, int ancho, int alto, int anguloInicio, int anguloArco);` Dibuja un arco.
- `boolean drawImage (Image m, int x, int y, ImageObserver b);` Dibuja la imagen referenciada por m, a partir de la posición (x, y).

Parámetros en un applet

A un *applet* se le pueden pasar datos constantes, parámetros, desde el documento HTML. Para este paso de parámetros se utiliza el marcador de HTML `param`. La sintaxis es:

```
<param name="nombre" value="valor asociado">
```

Por ejemplo, se desea construir un *applet* en el que el color y el radio de un círculo se obtienen mediante parámetros; el documento HTML:

```

<applet code="miAppletConColor.class" width="150" height="150"
<param name="radio" value="50"/>
<param name="color" value="red"/>
</applet>

```

La clase *applet* obtiene el valor del parámetro con el método `getParameter ()` de la clase `Applet`, cuya declaración es:

```
String getParameter (String nombreParametro);
```

Por ejemplo, para obtener el valor de los parámetros del HTML anterior:

```
public class miAppletConColor extends JApplet
{
    private String color;
    private int radio;
    public void init ( )
    {
        color = getParameter ("color");
        radio = Integer.parseInt (getParameter("radio"));
        setBackground (color);
        ...
    }
}
```

Los parámetros siempre se ponen con una cadena. La cadena, posiblemente, se tenga que transformar a otro tipo de dato, como se ha hecho con `parseInt ()`. La cadena que se pone con el marcador `param name` debe coincidir exactamente con la cadena del argumento de `getParameter ()`, las mayúsculas y minúsculas son significativas.

`getParameter ()` devuelve `null` si no encuentra la cadena en el HTML. Entonces, es buena práctica preguntar si devuelve `null` para poner un valor de manera predeterminada. Por ejemplo:

```
static final int RADIO = 25;

String cadRadio;
cadRadio = getParameter ("radio");
if (cadRadio != null)
    radio = Integer.parseInt (cadRadio);
else
    radio = RADIO;
```

Seguridad

Los *applets* están pensados para que se puedan ejecutar desde cualquier ordenador conectado a la red. El *applet* una vez descargado se ejecuta en modo local; naturalmente, esto exige establecer restricciones para prevenir acciones dañinas. En general, un *applet* puede mostrar imágenes, texto, reproducir sonidos y obtener respuestas del usuario. Los *applets* no pueden modificar el sistema del usuario, ni obtener características de dicho usuario. Para poner de manifiesto estas restricciones, se dice que un *applet* corre en un espacio seguro, o también en un “recinto controlado” (*sandbox*).

A continuación, se enumeran las restricciones generales de los *applets*:

- Los *applets* no pueden leer ni escribir archivos locales.
- Los *applets* no pueden ejecutar programas locales.
- Los *applets* no pueden comunicarse con otras computadoras. No pueden abrir sockets, salvo con el servidor origen del *applet*.
- Desde un *applet* no se puede obtener información relativa a la computadora local, salvo la versión de Java y el nombre del sistema operativo.

La encargada de verificar todas las instrucciones del *applet* es la máquina virtual Java. Se pueden utilizar *applets firmados* (llevan un certificado de seguridad) para establecer menos restricciones.



Resumen

- Java es un lenguaje de programación sencillo, orientado a objetos, robusto, seguro, neutro (independiente) de la arquitectura, portable, interpretado, multihilo, dinámico, de altas prestaciones y concebido para desarrollo de aplicaciones en la red, con una extensa biblioteca de clases y rutinas para trabajar con protocolos HTTP, FTP, TCP/IP y URL.
- Este capítulo realiza una revisión de la sintaxis básica del lenguaje Java, incluyendo las nuevas propiedades incorporadas a Java SE 5 y posteriores, como sentencia de control del flujo "for each", la clase `String`, la clase `Scanner` y sus métodos asociados.
- Java ha mejorado la sintaxis de C++ y las últimas versiones de C#, lenguaje competidor en el mundo de internet.
- La compilación de una clase o un programa Java produce un código denominado `bytecode`, un lenguaje intermedio, que es el lenguaje máquina de una computadora virtual (es el mismo para todas las computadoras). Cuando se ejecuta el `bytecode`, un programa llamado *intérprete* traduce y ejecuta sus instrucciones en su computadora.
- El capítulo introduce al lenguaje Java y proporciona los elementos básicos para la escritura de programas que faciliten su formación más avanzada en los siguientes capítulos. Se tratan las reglas para crear un programa sencillo en Java, como: comentarios, tipos de datos, variables, operadores, cadenas, entrada y salida, flujo de control y arreglos (*arrays*).

- Si usted ha seguido los capítulos ya explicados de C y C++, tiene ya fundamentos prácticos necesarios para compilar y ejecutar programas. Suponemos que ha instalado el entorno JDK: Le recomendamos que vaya al sitio web de Sun (www.sun.com) y descargue el entorno Java SE 5 o mejor, la última versión Java SE 7. También le recomendamos que aproveche la oportunidad que ofrece Oracle a los estudiantes y desarrolladores, de descarga gratuita de una extensa documentación sobre Java e instale esta documentación en su computadora o en algún servidor al que tenga acceso.
- La dirección para descargar la documentación de Java es:

<http://java.sun.com/javase/downloads/index.jsp>

- Los programas en Java son de dos tipos: aplicaciones y *applets*. Una aplicación se caracteriza por tener una clase principal en la que se encuentra el método `main`, que es el punto de entrada al programa. Un *applet* puede navegar por la red, descargarse y ejecutarse aunque con restricciones de seguridad.
- Para ejecutar un *applet* es necesario crear una página HTML, que utiliza el navegador para conocer la ruta donde se encuentra el *applet* y así poder descargarlo. Además, la página HTML establece el tamaño de la ventana de ejecución, la posición donde poner los objetos que forman el *applet*.



Ejercicios

21.1 Depurar el siguiente programa:

```
// un programa Java sin errores
class Primo
    void main( )
    {
        System.out.println("El lenguaje de
programación Java");
    }
}
```

21.2 Escribir las siguientes expresiones aritméticas como expresiones matemáticas en Java. La potencia puede hacerse con el método `Math.pow()`, por ejemplo

$$(x + y)^2 = \text{Math.pow}(x+y, 2)$$

a) $\frac{x}{y} + 1$ d) $\frac{b}{c+d}$ g) $\frac{xy}{1-4x}$
b) $\frac{x+y}{x-y}$ e) $(a+b)\frac{c}{d}$ h) $\frac{xy}{mn}$
c) $\frac{x+z}{x-y}$ f) $[(a+b)^2]^2$ i) $(x+y)^2 \cdot (a-b)$

21.3 Escribir un programa que lea dos enteros en las variables `x` y `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.

21.4 Escribir un programa que convierte un número dado de segundos en el equivalente de minutos y segundos.

21.5 ¿Qué valor se asigna a `consumo` en la sentencia `if` siguiente si la velocidad es de 120?

```
if (velocidad > 80)
    consumo = 10.00;
else if (velocidad > 100)
    consumo = 12.00;
else if (velocidad > 120)
    consumo = 15.00;
```

21.6 ¿Qué hay de incorrecto en el siguiente código?

```

if (x = 0) System.out.println(x + " = 0");
else System.out.println(x + " != 0");
if (x < y < z) System.out.println(x + "<" +
+ y "<" + z);
System.out.println("Introduzca n:");
n = Integer.parseInt(entrada.readLine
());
if (n < 0)
    System.out.println("El número es
    negativo. Pruebe de nuevo");
else
    System.out.println("Conforme. n = "
+ n);

```

21.7 ¿Cuál es la salida de los siguientes bucles?

1. for (i = 0; i < 10; i++)
 System.out.println("2* " + i + " = "+
 2*i);
2. for (i = 0; i <= 5; i++)
 System.out.println((2*i+1));
 System.out.println();
3. for (i = 1; i < 4; i++)
 {
 System.out.println(i);
 for (j = i; j >= 1; j--)
 System.out.println(j);
 }

Problemas

21.1 Un sistema de ecuaciones lineales

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

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 y y.

21.2 Escribir un programa `ConjuntoDatos` que calcule la suma y media de una secuencia de números coma flotante. Algunos métodos:

- void `sumarValores(double x)`
- int `getSuma()`
- double `getMedia()`

21.3 Escribir un programa que calcule los valores mayor y menor de una secuencia de números leídos del teclado. Diseñar los métodos:

- int `obtenerMayor()`
- int `obtenerMenor()`

21.4 Diseñar e implementar un programa que solicite al usuario una entrada como un dato tipo fecha y a con-

tinuación visualice el número del día correspondiente del año. Ejemplo, si la fecha es 30 12 1999, el número visualizado es 364.

21.5 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.

21.6 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.

21.7 Escribir un *applet* que muestre los 15 símbolos (1, x, 2) de una quiniela.

21.8 Escribir un *applet* que muestre tres círculos concéntricos: el número de color rojo, el segundo de color azul y el tercero verde. El radio del círculo menor que tenga 20 pixeles, los otros dos 30 y 40 pixeles.



Programación orientada a objetos en Java. Clases y objetos

Contenido

- 22.1 Clases y objetos
- 22.2 Declaración de una clase
- 22.3 Paquetes
- 22.4 Constructores
- 22.5 Recolección de objetos
- 22.6 Objeto que envía el mensaje: this

- 22.7 Miembros static de una clase
- 22.8 Clase Object
- 22.9 Tipos abstractos de datos en Java
 - › Resumen
 - › Ejercicios
 - › Problemas

Introducción

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, como `int`, `char` y `float` en Java, C y C++. Lenguajes de programación, como Java, tienen características que permiten ampliar el lenguaje añadiendo sus propios *tipos de datos*.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato, TAD, (abstract data type, ADT)*. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Una **clase** es un tipo de dato que contiene código (métodos) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En el capítulo aprenderá a crear (definir y especificar) y utilizar clases individuales.

Conceptos clave

- › Abstracción
- › Componentes
- › Constructores
- › Especificadores de acceso: `public`, `protected`, `private`
- › Interfaz
- › Ocultación de la información
- › Tipos de datos
- › variables

22.1 Clases y objetos

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard, quien intentando escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento.

Descubrió que los elementos del entorno que trataba de modelar, barcos, mareas y línea de la costa de los fiordos, y las acciones que cada elemento podía ejecutar, constituyan unas relaciones que eran más fáciles de manejar.

Las tecnologías orientadas a objetos han evolucionado mucho pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Martin/Odell definen un objeto como “cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos”. Para realizar esa actividad se añaden a cada objeto de la clase los propios datos y asociados con sus propios métodos miembro que pertenecen a la clase.

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico como una máquina de fax tiene una interfaz de usuario bien definida; esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón “enviar”. El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el periodo de garantía puede anularla.

¿Qué son clases?

En términos prácticos, una *clase* es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch¹ denomina a una clase como “un conjunto de objetos que comparten una estructura y comportamiento comunes”.

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios o métodos*. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos, variables o variables instancia*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable instancia* se suele utilizar en programas orientados a objetos.

22.2 Declaración de una clase

Antes de que un programa pueda crear objetos de cualquier clase, esta debe ser *definida*. La definición de una clase significa que se debe dar a la misma un nombre, darle nombre a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

Las *definiciones o especificaciones* no son códigos de programa ejecutables. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. Normalmente se sitúan en archivos formando los denominados *packages*, utilizando un archivo para varias clases que están relacionadas.

¹ Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones*. Madrid: Díaz de Santos/Addison-Wesley, 1995.

Formato

```
class NombreClase
{
    lista_de_miembros
}
```

NombreClase Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador válido).
lista_de_miembros métodos y datos miembros de la clase.

Definición de una clase llamada `Punto` que contiene las coordenadas `x` y `y` de un punto en un plano.

Ejemplo 22.1

```
class Punto
{
    private int x;      // coordenada x
    private int y;      // coordenada y

    public Punto(int x_,int y_) // constructor
    {
        x = x_;
        y = y_;
    }
    public Punto( ) // constructor sin argumentos
    {
        x = y = 0;
    }
    public int leerX( ) // devuelve el valor de x
    {
        return x;
    }
    public int leerY( ) // devuelve el valor de y
    {
        return y;
    }
    void fijarX(int valorX) // establece el valor de x
    {
        x = valorX;
    }
    void fijarY(int valorY) // establece el valor de y
    {
        y = valorY;
    }
}
```

Declaración de la clase `Edad`.

Ejemplo 22.2

```
class Edad
{
    private int edadHijo, edadMadre, edadPadre ; —— datos

    public Edad( ) {...} —— método especial: constructor
```

```

    public void iniciar(int h,int e,int p) {...} — métodos
    public int leerHijo( ) {...}
    public int leerMadre( ) {...}
    public int leerPadre( ) {...}
}

```

Objetos

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*. Un objeto se crea con el operador new aplicado a un constructor de la clase. Un objeto de la clase Punto inicializado a las coordenadas (2,1):

```
new Punto(2,1);
```

El operador new crea el objeto y devuelve una referencia al objeto creado. Esta referencia se asigna a una *variable* del tipo de la clase. El objeto permanecerá vivo siempre que esté referenciado por una variable de la clase que es instancia.

Formato para definir una referencia

```
NombreClase      varReferencia;
```

Formato para crear un objeto

```
varReferencia = new NombreClase(argumentos_constructor);
```

Toda clase tiene uno o más métodos, denominados *constructores*. Para inicializar el objeto cuando es creado, tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados. En la clase Edad del ejemplo 22.2 el constructor no tiene argumentos, se puede crear un objeto:

```
Edad f = new Edad( );
```

El *operador de acceso* a un miembro (.) selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```

Punto p;
p = new Punto( );
p.fijarX(100);
System.out.println(" Coordenada x es " + p.leerX( ) );

```

A recordar

El operador punto (.) se utiliza con los nombres de los métodos y variables instancia para especificar que son miembros de un objeto.

Ejemplo : Clase DiaSemana, contiene un método visualizar ()

```

DiaSemana hoy; // hoy es una referencia
hoy = new DiaSemana( ); // se ha creado un objeto
hoy.visualizar( ); // llama al método visualizar( )

```

Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*, que significa que no se puede acceder a determinados datos del interior de una clase por métodos externos a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o métodos privados solo se puede acceder desde dentro de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase.

Para controlar el acceso a los miembros de la clase se utilizan tres diferentes *especificadores de acceso*: `public`, `private` y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

Formato

```
class NombreClase
{
    private declaración miembro privado; // miembros privados
    protected declaración miembro protegido; // miembros protegidos
    public declaración miembro público; // miembros públicos
}
```

El especificador `public` define miembros públicos, que son aquellos a los que se puede acceder por cualquier método desde fuera de la clase. A los miembros que siguen al especificador `private` solo se puede acceder por métodos miembro de la misma clase. A los miembros que siguen al especificador `protected` se puede acceder por métodos miembro de la misma clase o de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo *paquete*. Los especificadores `public`, `protected` y `private` pueden aparecer en cualquier orden. Si no se especifica acceso (*acceso en forma predeterminada*) a un miembro de una clase, a este se puede acceder desde los métodos de la clase y desde cualquier método de las clases del *paquete* en que se encuentra.

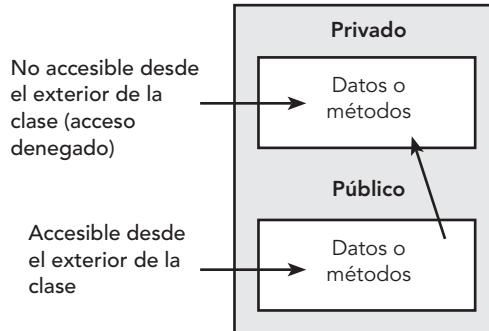


Figura 22.1 Secciones pública y privada de una clase.

Declaración de la clase `Foto` y `Marco` con miembros declarados con distinta visibilidad.
Ambas clases forman parte del paquete `soporte`.

Ejemplo 22.3

```
package soporte;

class Foto
{
    private int nt;
    private char opd;
    String q;
    public Foto(String r) // constructor
    {
        nt = 0;
        opd = 'S';
        q = new String(r);
    }
    public double mtd( ){...}
}
```

```

class Marco
{
    private double p;
    String t;
    public Marco( ) { ... }
    public void poner( )
    {
        Foto u = new Foto("Paloma");
        p = u.mtd( );
        t = "***" + u.q + "***";
    }
}

```

Tabla 22.1 Visibilidad, "x" indica que el acceso está permitido.

Tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de clase del paquete	Miembro de clase de otro paquete
Private	x			
En blanco	x		x	
Protected	x	x	x	
Public	x	x	x	x

Aunque las especificaciones *públicas*, *privadas* y *protegidas* pueden aparecer en cualquier orden, en Java los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, y que usted puede elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contiene los atributos (datos).
2. Se ponen los miembros públicos primero debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El *principio de ocultación* de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permita que los detalles de implementación de los objetos sean ignorados. Por consiguiente, los datos y métodos públicos forman la interfaz externa del objeto, mientras que los elementos *privados* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto. Los elementos de una clase sin especificador y los *protected* tienen las mismas propiedades que los públicos respecto a las clases del paquete.

A recordar

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Métodos de una clase

Los métodos en Java siempre son miembros de clases, no hay métodos o funciones fuera de las clases. La implementación de los métodos se incluye dentro del cuerpo de la clase. La figura 22.2 muestra la declaración completa de una clase.

```

class Producto <----- nombre de la clase
{
    private int numProd; <----- acceso para almacenamiento de datos
    private String nomProd;
    private String descripProd; <----- declaraciones para almacenamiento
    private double precioProd; <----- de datos
    private int numUnidades;

    public Producto ( ) {...} <----- métodos
    public Producto(int n,char [ ] nom,char [ ] des,double p,int nu) { }
    public void verProducto ( ) {...}
    public double obtenerPrecio ( ){...}
    public void actualizarProd(int b) {...}
}

```

Figura 22.2 Definición típica de una clase.

La clase `Racional` define el numerador y denominador característico de un número racional. Por cada dato se proporciona un método miembro que devuelve su valor y un método que asigna numerador y denominador. Tiene un constructor que inicializa un objeto a 0/1.

Ejemplo 22.4

```

class Racional
{
    private int numerador;
    private int denominador;
    public Racional( )
    {
        numerador = 0;
        denominador = 1;
    }
    public int leerN( ) { return numerador; }
    public int leerD( ) { return denominador; }
    public void fijar (int n, int d)
    {
        numerador = n;
        denominador = d;
    }
}

```

11

Ejercicio 22.1

Definir una clase `DiaAnyo` que contenga los atributos `mes` y `día`, el método `igual ()` y el método `visualizar ()`. El mes se registra como un valor entero en la variable `mes` (1, enero, 2, febrero, etc.). El día del mes se registra en la variable entera `día`. Escribir un programa que comprueba si una fecha es su cumpleaños.

El método `main ()` de la clase principal, `Cumple`, crea un objeto `DiaAnyo` y llama al método `igual ()` para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído del dispositivo de entrada.

```
import java.util.*;
```

```

class DiaAnyo
{
    private int mes;
    private int dia;

    public DiaAnyo(int d, int m)
    {
        dia = d;
        mes = m;
    }
    public boolean igual(DiaAnyo d)
    {
        if ((dia == d.dia) && (mes == d.mes))
            return true;
        else
            return false;
    }
    public void visualizar( )
    {
        System.out.println("mes = " + mes + " , dia = " + dia);
    }
}
// clase principal, con método main
public class Cumple
{
    public static void main(String[ ] ar)
    {
        DiaAnyo hoy;
        DiaAnyo cumpleanyos;
        int d, m;
        Scanner entrada = new Scanner(System.in);
        System.out.print("Introduzca fecha de hoy, dia: ");
        d = entrada.nextInt( );
        System.out.print("Introduzca el número de mes: ");
        m = entrada.nextInt( );
        hoy = new DiaAnyo(d,m);
        System.out.print("Introduzca su fecha de nacimiento, dia: ");
        d = entrada.nextInt( );
        System.out.print("Introduzca el número de mes: ");
        m = entrada.nextInt( );
        cumpleanyos = new DiaAnyo(d,m);
        System.out.print( " La fecha de hoy es " );
        hoy.visualizar( );
        System.out.print( " Su fecha de nacimiento es " );
        cumpleanyos.visualizar( );
        if (hoy.igual(cumpleanyos))
            System.out.println( ";Feliz cumpleaños ! " );
        else
            System.out.println( ";Feliz dia ! " );
    }
}

```

Implementación de las clases

El código fuente de la definición de una clase, con todos sus métodos y variables instancia se almacenan en archivos de texto con extensión `.java` y el nombre de la clase, por ejemplo `Racional.java`. Normalmente, se sitúa la implementación de cada clase en un archivo independiente.

Las clases pueden proceder de diferentes fuentes:

- Se pueden declarar e implementar sus propias clases. El código fuente siempre estará disponible; pueden estar organizadas por paquetes.
- Se pueden utilizar clases que hayan sido escritas por otras personas o incluso que se han comprado. En este caso, se puede disponer del código fuente o estar limitado a utilizar el *bytecode* de la implementación. Será necesario disponer del paquete donde se encuentran.
- Se puede utilizar clases de los diversos *packages* que acompañan a su software de desarrollo Java.

Clases públicas

La declaración de una clase puede incluir el modificador `public` como prefijo en la cabecera de la clase. Por ejemplo:

```
public class Examen
{
    // miembros de la clase
}
```

La clase `Examen` puede ser utilizada por las clases que se encuentran en su mismo archivo (*package*), o por clases de cualquier otro paquete o archivo. Habitualmente las clases se definen como `public`, a no ser que se quiera restringir su uso a las clases del paquete. Una clase declarada sin el prefijo `public` establece una restricción importante, y es que solo podrá ser utilizada por las clases definidas en el mismo paquete.

Advertencia

El especificador de acceso `public` es el único que se puede especificar en la cabecera de una clase.

22.3 Paquetes

Los paquetes es la forma que tiene Java de organizar los archivos con las clases necesarias para construir las aplicaciones. Java incorpora varios paquetes, por ejemplo `java.lang`, `java.io`, o `java.util` con las clases básicas para construir programas: `System`, `String`, `Integer`, `Scanner`

Sentencia `package`

¿Cómo se puede definir un paquete?, la sentencia `package` se utiliza para este cometido. En primer lugar se debe incluir la sentencia `package` como primera línea del archivo fuente de cada una de las clases del paquete. Por ejemplo, si las clases `Lapiz`, `Boligrafo` y `Folio` se van a organizar formando el paquete `escritorio`, el esquema a seguir es el siguiente:

```
// archivo fuente Lapiz.java
package escritorio;
public class Lapiz
{
    // miembros de clase Lapiz
}
// archivo fuente Boligrafo.java
package escritorio;
public class Boligrafo
{
    // miembros de clase Boligrafo
}
// archivo fuente Folio.java
```

```
package escritorio;
public class Folio
{
    // miembros de clase Folio
}
```

Formato

```
package NombrePaquete;
```

En segundo lugar, una vez creado el archivo fuente de cada clase del paquete, se deben ubicar estos en un subdirectorio con el mismo nombre que el del paquete. En el esquema anterior se ubicarán los archivos `Lapiz.java`, `Boligrafo.java` y `Folio.java` en el path `escritorio`.

El uso de paquetes tiene dos beneficios importantes:

1. Las restricciones de visibilidad son menores entre clases que están dentro del mismo paquete. Desde cualquier clase de un paquete, los miembros `protected` y los miembros sin modificador de visibilidad son accesibles; sin embargo no lo son desde clases de otros paquetes.
2. La selección de las clases de un paquete se puede abreviar con la sentencia `import` del paquete.

import

Las clases que se encuentran en los paquetes se identifican utilizando el nombre del paquete, el selector punto `(.)` y a continuación el nombre de la clase. Por ejemplo, la declaración de la clase `Arte` con atributos de la clase `PrintStream` (paquete `java.io`) y `Lapiz` (paquete `escritorio`):

```
public class Arte
{
    private java.io.PrintStream salida;
    private escritorio.Lapiz p;
}
```

La sentencia `import` facilita la selección de una clase, permite escribir únicamente su nombre, evitando el nombre del paquete. La declaración anterior se puede abreviar:

```
import java.io.PrintStream;
import escritorio.*;
public class Arte
{
    private PrintStream salida;
    private Lapiz p;
}
```

La sentencia `import` debe aparecer antes de la declaración de las clases, a continuación de la sentencia `package`. Tiene dos formatos:

Formato

```
import identificadorpaquete.nombreClase;
import identificadorpaquete.*;
```

El primer formato especifica una clase concreta. El segundo formato especifica que para todas las clases del paquete no hace falta cualificar el nombre de la clase con el nombre del paquete.

Con frecuencia se utiliza el formato `.*`. Tiene la ventaja de poder simplificar cualquier clase del paquete, se pueden señalar los siguientes problemas:

- Se desconoce qué clases concretas del paquete se están utilizando. Por contra, con una sentencia `import` por cada clase se conocen las clases utilizadas.
- Puede haber colisiones entre nombres de clases declaradas en el archivo y nombres de clases del paquete.

- Mayor tiempo de compilación debido a que el compilador busca la existencia de cualquier clase en el paquete.

Nota

Aunque aparezca la sentencia `import paquete.*;`, el compilador genera *bytecode* solo para las clases utilizadas.

Se crea el paquete numérico con la clase Random y se utiliza la clase en una aplicación.

Ejemplo 22.5

```
package numerico;
public class Random
{
    // ...
}
```

Al utilizar la clase en otro archivo:

```
import java.util.*;
import numerico.*;
```

En el paquete `java.util` se encuentra la clase `Random`, entonces se produce una ambigüedad con la clase `Random` del paquete numérico. Es necesario cualificar completamente el nombre de la clase `Random` de, por ejemplo, `java.util`.

```
java.util.Random aleatorio; // define una referencia
```

22.4 Constructores

Un *constructor* es un método que se ejecuta automáticamente cuando se crea un objeto de una clase. Sirve para inicializar los miembros de la clase.

El constructor tiene el mismo nombre que clase. Cuando se define no se puede especificar un valor de retorno, nunca devuelve un valor. Sin embargo, puede tomar cualquier número de argumentos.

Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero, o más argumentos.
3. No tiene tipo de retorno.

La clase Rectángulo tiene un constructor con cuatro parámetros.

Ejemplo 22.6

```
public class Rectangulo
{
    private int izdo;
    private int superior;
    private int dcha;
    private int inferior;
    // constructor
    public Rectangulo(int iz, int sr, int d, int inf)
    {
        izdo = iz;
        superior = sr;
        dcha = d;
    }
}
```

```

        inferior = inf;
    }
    // definiciones de otros métodos miembro
}

```

Al crear un objeto se pasan los valores de los argumentos al constructor, con la misma sintaxis que la de llamada a un método. Por ejemplo:

```
Rectangulo Rect = new Rectangulo(25, 25, 75, 75);
```

Se ha creado una instancia de `Rectangulo`, pasando valores concretos al constructor de la clase, de esta forma queda inicializado.

Constructor por defecto

Un constructor que no tiene parámetros se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato de la clase con valores en forma predeterminada.

Regla

Java crea automáticamente un constructor por defecto cuando no existen otros constructores. Tal constructor inicializa las variables de tipo numérico (`int`, `float` ...) a cero, las variables de tipo `boolean` a `true` y las referencias a `null`.



Ejemplo 22.7

El constructor por defecto inicializa `x` y `y` a 0.

```

public class Punto
{
    private int x;
    private int y;

    public Punto()           // constructor por defecto
    {
        x = 0;
        y = 0;
    }
}

```

Cuando se crea un objeto `Punto` sus miembros dato se inicializan a 0.

```
Punto P1 = new Punto(); // P1.x ej 0, P1.y es 0
```

Precaución

Tenga cuidado con la escritura de una clase con solo un constructor con argumentos. Si se omite un constructor sin argumento no será posible utilizar el constructor por defecto. La definición `NomClase c = new NomClase()` no será posible.

Constructores sobrecargados

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar el constructor de una clase. De hecho los constructores sobrecargados son bastante frecuentes, proporcionan diferentes alternativas de inicializar objetos.

Regla

Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

La clase `EquipoSonido` se define con tres constructores; un constructor por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.

Ejemplo 22.8

```
public class EquipoSonido
{
    private int potencia;
    private int voltios;
    private int numCd;
    private String marca;

    public EquipoSonido( )      // constructor por defecto
    {
        marca = "Sin marca";
        System.out.println("Constructor por defecto");
    }
    public EquipoSonido(String mt)
    {
        marca = mt;
        System.out.println("Constructor con argumento cadena ");
    }
    public EquipoSonido(String mt, int p, int v)
    {
        marca = mt;
        potencia = p;
        voltios = v;
        numCd = 20;
        System.out.println("Constructor con tres argumentos ");
    }
    public double factura( ){...}

};
```

La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor:

```
EquipoSonido rt, gt, ht;           //define tres referencias
rt = new EquipoSonido ( );         //llamada al constructor por defecto
gt = new EquipoSonido("JULAT");
rt = new EquipoSonido("PARTOLA", 35, 220);
```

22.5 Recolección de objetos

En Java un objeto siempre ha de estar referenciado por una variable; en el momento que un objeto deja de estar referenciado se activa la rutina de recolección de memoria; se puede decir que el objeto es liberado y la memoria que ocupa puede ser reutilizada. Por ejemplo:

```
Punto p = new Punto(1,2);
```

la sentencia `p = null` provoca que el objeto `Punto` sea liberado automáticamente.

El propio sistema se encarga de recolectar los objetos en desuso para aprovechar la memoria ocupada. Para ello hay un proceso que se activa periódicamente y toma los objetos que no están referenciados por ninguna variable. El proceso lo realiza el método `System.gc` (*garbage collection*). Por ejemplo, el siguiente método crea objetos `Contador` y después se liberan al perder su referencia.

```
void objetos( )
{
    Contador k, g, r, s;
    // se crean cuatro objetos
    k = new Contador( );
    g = new Contador( );
    r = new Contador( );
    s = new Contador( );
    /* la siguiente asignación hace que g referencie al mismo
       objeto que k, además el objeto original de g será
       automáticamente recolectado. */
    g = k;
    /* ahora no se activa el recolector porque g sigue apuntando al
       objeto. */
    k = null;
    /* a continuación sí se activa el recolector para el objeto
       original de r. */
    r = new Contador( );

} // se liberan los objetos actuales apuntados por g, r, s
```

Método `finalize()`

El método `finalize()` es especial, se llama automáticamente si ha sido definido en la clase, justo antes que la memoria del objeto “recolectado” vaya a ser devuelta al sistema. El método no es un destructor del objeto, no libera memoria; en algunas aplicaciones se puede utilizar para liberar ciertos recursos del sistema.

Regla

`finalize()` es un método especial con estas características:

- No devuelve valor, es de tipo `void`.
- No tiene argumentos.
- No puede sobrecargarse.
- Su definición es opcional.



Ejercicio 22.2

Se declaran dos clases, cada una con su método `finalize()`. El método `main()` crea objetos de ambas clases; las variables que referencian a los objetos se modifican para que cuando se active la recolección automática de objetos se libere la memoria de estos; hay una llamada a `System.gc()` para no esperar a la llamada interna del sistema.

```
import java.io.*;
class Demo
{
    private int datos;
    public Demo( ){datos = 0;}
    protected void finalize( )
    {
```

```

        System.out.println("Fin de objeto Demo");
    }
}
class Prueba
{
    private double x;
    public Prueba( ){x = -1.0;}
    protected void finalize( )
    {
        System.out.println("Fin de objeto Prueba");
    }
}
public class ProbarDemo
{
    public static void main(String[ ] ar)
    {
        Demo d1, d2;
        Prueba p1, p2;
        d1 = new Demo( );
        p1 = new Prueba( );
        System.gc( ); // no se libera ningún objeto
        p2 = p1;
        p1 = new Prueba( );
        System.gc( ); // no se libera ningún objeto
        p1 = null;
        d1 = new Demo( );
        System.gc( ); // se liberan dos objetos
        d2 = new Demo( );
    } // se liberan los objetos restantes
}

```

22.6 Objeto que envía el mensaje: this

this es una referencia al objeto que envía un *mensaje*, o simplemente, una referencia al objeto que llama a un método (este no debe ser static). Internamente está definida:

```
final NombreClase this;
```

por consiguiente no puede modificarse. Las variables y métodos de las clases están referenciados, implícitamente, por this. Pensemos, por ejemplo, en la siguiente clase:

```

class Triangulo
{
    private double base;
    private double altura;
    public double area( )
    {
        return base*altura/2.0 ;
    }
}

```

En el método `area()` se hace referencia a las variables instancia `base` y `altura`. ¿A la base, altura de qué objeto? El método es común para todos los objetos `Triangulo`. Aparentemente no distingue entre un objeto u otro, sin embargo cada variable instancia implícitamente está cualificada por `this`. Es como si estuviera escrito:

```
public double area( )
```

```

{
    return this.base*this.altura/2.0 ;
}

```

Fundamentalmente `this` tiene dos usos:

- Seleccionar explícitamente un miembro de una clase con el fin de dar más claridad o de evitar colisión de identificadores. Por ejemplo:

```

class Triangulo
{
    private double base;
    private double altura;
    public void datosTriangulo(double base, double altura)
    {
        this.base = base;
        this.altura = altura;
    }
    // ...
}

```

Se ha evitado con `this` la colisión entre argumentos y variables instancia.

- Que un método devuelva el mismo objeto que le llamó. De esa manera se pueden hacer llamadas en cascada a métodos de la misma clase. De nuevo se define la clase `Triangulo`:

```

class Triangulo
{
    // ...
    public Triangulo visualizar ( )
    {
        System.out.println(" Base = " + base);
        System.out.println(" Altura = " + altura);
        return this;
    }
    public double area ( )
    {
        return base*altura/2.0 ;
    }
}

```

Ahora se pueden concatenar llamadas a métodos:

```

Triangulo t = new Triangulo ( );
t.datosTriangulo(15.0,12.0).visualizar ( );

```

22.7 Miembros static de una clase

Cada instancia de una clase, cada objeto, tiene su propia copia de las variables de la clase. Cuando interese que haya miembros que no estén ligados a los objetos sino a la clase, y por lo tanto comunes a todos los objetos, estos se declaran `static`.

Variables static

Las variables de clase `static` son compartidas por todos los objetos de la clase. Se declaran de igual manera que otra variable, añadiendo como prefijo la palabra reservada `static`. Por ejemplo:

```

public class Conjunto
{
    private static int k = 0;
}

```

```
static Totem lista = null;
// ...
}
```

Las variables miembro `static` no forman parte de los objetos de la clase sino de la propia clase. Dentro de las clases se accede a ellas de la manera habitual, simplemente con su nombre. Desde fuera de la clase se accede con el nombre de la clase, el selector y el nombre de la variable:

```
Conjunto.lista = ...;
```

También se puede acceder a través de un objeto de la clase, aunque no es recomendable ya que los miembros `static` no pertenecen a los objetos sino a las clases.

Ejercicio 22.3

Dada una clase se quiere conocer en todo momento los objetos activos en la aplicación.

Se declara la clase `Ejemplo` con un constructor por defecto y otro con un argumento. Ambos incrementan la variable `static` `cuenta`, en 1. De esa manera cada nuevo objeto queda contabilizado. También se declara el método `finalize()`, de tal forma que al activarse `cuenta` se decremente en 1.

El método `main()` crea objetos de la clase `Ejemplo` y visualiza la variable que contabiliza el número de sus objetos.

```
class Ejemplo
{
    private int datos;
    static int cuenta = 0;
    public Ejemplo( )
    {
        datos = 0;
        cuenta++; // nuevo objeto
    }
    public Ejemplo(int g)
    {
        datos = g;
        cuenta++; // nuevo objeto
    }
    //
    protected void finalize( )
    {
        System.out.println("Fin de objeto Ejemplo");
        cuenta--;
    }
}

public class ProbarEjemplo
{
    public static void main(String [ ] ar)
    {
        Ejemplo d1, d2;

        System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
        d1 = new Ejemplo ( );
        d2 = new Ejemplo(11);
        System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
    }
}
```

```

        d2 = d1;
        System.gc( );
        System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
        d2 = d1 = null;
        System.gc( );
        System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
    }
}

```

Una variable `static` suele inicializarse directamente en la definición. Sin embargo, existe una construcción de Java que permite inicializar miembros `static` en un bloque de código dentro de la clase; el bloque debe venir precedido de la palabra `static`. Por ejemplo:

```

class DemoStatic
{
    private static int k;
    private static double r;
    private static String cmn;
    static
    {
        k = 1;
        r = 0.0;
        cmn = "Bloque";
    }
}

```

Métodos `static`

Los métodos de las clases se llaman a través de los objetos. En ocasiones interesa definir métodos que estén controlados por la clase, que no haga falta crear un objeto para llamarlos; son los métodos `static`. Muchos métodos de la biblioteca Java están definidos como `static`. Es, por ejemplo, el caso de los métodos matemáticos de la clase `Math`: `Math.sin()`, `Math.sqrt()`.

La llamada a los métodos `static` se realiza a través de la clase: `NOMBRECLASE.metodo()`, respetando las reglas de visibilidad. También se pueden llamar con un objeto de la clase, aunque no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.

Los métodos definidos como `static` no tienen asignada la referencia `this`, por ello solo pueden acceder a miembros `static` de la clase. Es un error que un método `static` acceda a miembros de la clase no `static`. Por ejemplo:

```

class Fiesta
{
    int precio;
    String cartel;
    public static void main(String [ ] a)
    {
        cartel = "Virgen de los pacientes";
        precio = 1;
    ...
}

```

al compilar da dos errores debido a que desde el método `main()`, definido como `static` se accede a miembros no `static`.

La clase `SumaSerie` define tres variables `static`, y un método `static` que calcula la suma cada vez que se llama.

Ejemplo 22.9

```
class SumaSerie
{
    private static long n;
    private static long m;
    static
    {
        n = 0;
        m = 1;
    }
    public static long suma( )
    {
        m += n;
        n = m - n;
        return m;
    }
}
```

22.8 Clase Object

`Object` es la superclase base de todas las clases de Java; toda clase definida en Java hereda de la clase `Object` y en consecuencia toda variable referencia a una clase se convierte, automáticamente, al tipo `Object`. Por ejemplo:

```
Object g;
String cd = new String("Barranco la Parra");
Integer y = new Integer(72);           // objeto inicializado a 72

g = cd;                      // g referencia al mismo objeto que cd
g = y;                        // g ahora referencia a un objeto Integer
```

La clase `Object` tiene dos métodos importantes: `equals()` y `toString()`. Generalmente, se redefinen en las clases para especializarlos.

equals()

Compara el objeto que hace la llamada con el objeto que se pasa como argumento, devuelve `true` si son iguales.

```
boolean equals(object k);
```

El siguiente ejemplo compara dos objetos, la comparación es `true` si contienen la misma cadena.

```
String ar = new String("Iglesia románica");
String a = "Vida sana";
if (ar.equals(a))           //...no se cumple, devuelve false
```

toString()

Este método construye una cadena que es la representación del objeto, devuelve la cadena. Normalmente se redefine en las clases para así dar detalles explícitos de los objetos de la clase.

```
String toString( )
```

En el siguiente ejemplo un objeto `Double` llama al método `toString()` y asigna la cadena a una variable.

```
Double r = new Double(2.5);
String rp;
rp = r.toString( );
```

Operador instanceof

Con frecuencia se necesita conocer la clase de la que es instancia un objeto. Se tiene que tener en cuenta que en las jerarquías de clases se dan conversiones automáticas entre clases derivadas y su clase base, en particular, cualquier referencia de tipo clase se puede convertir a una variable de tipo `Object`.

Con el operador `instanceof` se determina la clase a la que pertenece un objeto, tiene dos operandos, el primero un objeto y el segundo una clase. Evalúa la expresión a `true` si el primer operando es una instancia del segundo. La siguiente función tiene un argumento de tipo `Object`, por consiguiente puede recibir cualquier referencia y selecciona la clase a la que pertenece el objeto transmitido (`String, Vector, ...`):

```
public hacer (Object g)
{
    if (g instanceof String)
    ...
    else if (g instanceof Vector)
    ...
}
```

A recordar

El operador `instanceof` se puede considerar un operador relacional, su evaluación da como resultado un valor de tipo `boolean`.

22.9 Tipos abstractos de datos en Java

La implementación de un **TAD** en Java se realiza de forma natural con una clase. Dentro de la clase va a residir la representación de los datos junto a las operaciones (métodos de la clase). La interfaz del tipo abstracto queda perfectamente determinado con la etiqueta `public`, que se aplicará a los métodos de la clase que representen operaciones.

Por ejemplo, si se ha especificado el *TAD Punto* para representar la *abstracción punto* en el espacio tridimensional, la siguiente clase implementa el tipo:

```
class Punto
{
    // representación de los datos
    private double x, y, z;
    // operaciones
    public double distancia(Punto p);
    public double modulo();
    public double anguloZeta();
    ...
}
```

Implementación del TAD Conjunto

La clase `Conjunto` implementa el *TAD Conjunto*. Las operaciones que define son: añadir un elemento, retirar un elemento, pertenencia de un elemento y unión. Se puede enriquecer la clase con la operación intersección, diferencia de conjuntos y otras operaciones típicas de los conjuntos. La clase representa los datos de forma genérica, utiliza un arreglo para almacenar los elementos, de tipo `Object`.

Archivo conjunto.java

```
package conjunto;
public class Conjunto
{
    static int M = 20;      // aumento de la capacidad
    private Object [ ] cto;
    private int cardinal;
    private int capacidad;
    // operaciones
```

```
public Conjunto( )
{
    cto = new Object [M];
    cardinal = 0;
    capacidad = M;
}
public boolean esVacio( )
{
    return (cardinal == 0);
}
public void annadir(Object elemento)
{
    if (!pertenece(elemento))
    {
        /* verifica si hay posiciones libres,
        en caso contrario amplía el conjunto */
        if (cardinal == capacidad)
        {
            Object [ ] nuevoCto;
            nuevoCto = new Object [capacidad + M];
            for (int k = 0; k < capacidad; k++)
                nuevoCto[k] = cto[k];
            capacidad += M;
            cto = nuevoCto;
            System.gc( );           // devuelve la memoria no referenciada
        }
        cto[cardinal++] = elemento;
    }
}
public void retirar(Object elemento)
{
    if (pertenece(elemento))
    {
        int k = 0;
        while (!cto[k].equals(elemento))
            k++;

        /* desde el elemento k hasta la última posición
        mueve los elementos una posición a la izquierda */

        for ( ; k < cardinal ; k++)
            cto[k] = cto[k+1];

        cardinal--;
    }
}
public boolean pertenece(Object elemento)
{
    int k = 0;
    boolean encontrado = false;

    while (k < cardinal && !encontrado)
    {
        encontrado = cto[k].equals(elemento);
        k++;
    }
    return encontrado;
}
```

```

        }
    public int cardinal( )
    {
        return this.cardinal;
    }
    public Conjunto union(Conjunto c2)
    {
        Conjunto u = new Conjunto( );
        // primero copia el primer operando de la unión
        for (int k = 0; k < cardinal; k++)
            u.cto[k] = cto[k];
        u.cardinal = cardinal;
        // añade los elementos de c2 no incluidos
        for (int k = 0; k < c2.cardinal; k++)
            u.annadir(c2.cto[k]);
        return u;
    }
    public Object elemento(int n) throws Exception
    {
        if (n <= cardinal)
            return cto[--n];
        else
            throw new Exception("Fuera de rango");
    }
}

```

Aplicación del tipo abstracto de dato Conjunto

A continuación se escribe una aplicación que muestra el uso del tipo conjunto. El programa crea un conjunto de números racionales y realiza operaciones definidas en la clase *Conjunto*. Se añade una operación complementaria para visualizar los elementos del conjunto. También se declara la clase *Racional* para representar a este tipo de números.

```

import java.io.*;
import conjunto.*;
import java.util.Random;

class Racional
{
    private int n, d;
    public Racional(int num, int den) throws Exception
    {
        if (den == 0)
            throw new Exception("Error: denominador 0");
        n = num;
        d = den;
    }
    public Racional( )
    {
        n = 0; d = 1;
    }
    public String toString( )
    {
        String racs;
        racs = n + "/" + d;
        return racs;
    }
}

```

```
class ConjuntoRacional
{
    public static void main(String [ ] ar)
    {
        final int NELEM = 10;
        Conjunto cn1, cn2, t;
        Racional r;
        int n, d;

        cn1 = new Conjunto( );
        cn2 = new Conjunto( );
        Random a = new Random( );
        try {
            for (int j = 1; j <= NELEM; j++)
            {
                n = a.nextInt(15); // numerador;
                d = a.nextInt(15); // denominador;
                if (d != 0)
                {
                    r = new Racional(n,d);
                    cn1.annadir(r);
                }
            }
            mostrar(cn1);
            // se crea un segundo conjunto
            for (int j = 1; j <= NELEM; j++)
            {
                n = a.nextInt(15); // numerador;
                d = a.nextInt(15); // denominador;
                if (d != 0)
                {
                    r = new Racional(n,d);
                    cn2.annadir(r);
                }
            }
            mostrar(cn2);
            // union de conjuntos
            System.out.println(" Conjunto unión");
            t = cn1.union(cn2);
            mostrar(t);
        }
        catch (Exception er)
        {
            System.err.println("Error en ejecución: " + er);
        }
    }
    static void mostrar(Conjunto c) throws Exception
    {
        System.out.println("\t\tElementos del conjunto ");
        for (int k = 1; k <= c.cardinal( ); k++)
        {
            System.out.print(c.elemento(k) + " ");
            if (k % 10 == 0) System.out.println( );
        }
        System.out.println( );
    }
}
```



Resumen

- Los tipos abstractos de datos (TAD) describen un conjunto de objetos con la misma representación y comportamiento. La implementación de un tipo abstracto de datos está oculta. Por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de dato sin cambiar su interfaz. En Java, los tipos abstractos de datos se implementan mediante **clases**.
- Una **clase** es un tipo de dato definido por el programador que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos *componentes*: un conjunto de atributos o variables instancia y un conjunto de comportamientos (métodos). Los atributos también se llaman variables instancia o miembros dato y los comportamientos se llaman métodos miembro.

```
class Circulo
{
    private double centroX;
    private double centroY;
    private double radio;
    public double superficie( ){ }
}
```

- Un objeto es una instancia de una clase y una variable cuyo tipo sea la clase es una referencia a un objeto de la clase.

```
Circulo unCirculo; // variable del tipo clase
Circulo [ ] tipocirculo = new Circulo[10];
// array de referencias
```

- La definición de una clase, en cuanto a visibilidad de sus miembros, tiene tres secciones: *pública*, *privada* y *protegida*.
- La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Se recomienda la declaración de los constructores en la sección pública.
- La sección privada contiene los métodos miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estos métodos miembro y atributos dato son accesibles solo por los métodos miembro del objeto.
- Los miembros de una clase con visibilidad *protected* son accesibles para cualquier usuario de la clase

que se encuentre en el mismo package; también son accesibles para las clases derivadas. El acceso *por defecto*, sin modificador, tiene las mismas propiedades que el acceso *protected* para las clases que se encuentran en el mismo package.

- Un **constructor** es un método miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class Complejo
{
    public Complejo (double x, double y) { }
    public Complejo(complejo z) { }
}
```

- El **constructor** es un método especial que se invoca cuando se crea un objeto. Se utiliza, normalmente, para inicializar los atributos de un objeto. Por lo general, al menos se define un constructor sin argumentos, llamado constructor por defecto. En caso de no definirse constructor, implícitamente queda definido un constructor sin argumentos que inicializa cada miembro numérico a 0, los miembros de tipo boolean a true y las referencias a null.
- El proceso de crear un objeto se llama *instanciación* (creación de instancia). En Java se crea un objeto con el operador new y un constructor de la clase.

```
Circulo C = new Circulo( );
```

- En Java la liberación de objetos es automática, cuando un objeto deja de estar referenciado por una variable es candidato a que la memoria que ocupa sea liberada y posteriormente reutilizada. El proceso se denomina *garbage collection*, el método `System.gc()` realiza el proceso.
- Los paquetes son agrupaciones de clases relativas a un tema. El sistema suministra paquetes con clases que facilitan la programación. El paquete `java.lang` se puede afirmar que es donde se encuentran las clases más utilizadas, por esa razón es automáticamente incorporado a los programas.
- Los miembros de una clase definidos como `static` no están ligados a los objetos de la clase sino que son comunes a todos los objetos, son de la clase. Se cualifican con el nombre de la clase, por ejemplo:

```
Math.sqrt(x)
```



Ejercicios

22.1 ¿Qué está mal en la siguiente definición de la clase?

```
import java.io.*;
class Buffer
{
    private char datos [ ];
    private int cursor ;
    private Buffer(int n)
    {
        datos = new char [n]
    };
    public static int long( return cursor,);
    public String contenido ( ) { }
}
```

22.2 Dado el siguiente programa, ¿es legal la sentencia de main ()?

```
class Punto
{
    public int x, int y;
    public Punto(int x1, int y1) {x = x1 ; y
        = y1;}
};
class CreaPunto
{
    public static void main(String [ ] a)
    {
        new Punto(25, 15); //¿es legal esta
                           sentencia ?
        Punto p = new Punto ( );
        //¿es legal esta sentencia ?
    }
}
```

22.3 Suponiendo contestado el ejercicio anterior, ¿cuál será la salida del siguiente programa?

```
class CreaPunto
{
    public static void main(String [ ] a)
    {
        Punto q;
        q = new Punto(2, 1);
        System.out.println("x = " + p.x + "\ty
                           = " + p.y);
    }
}
```

22.4 Dada la siguiente clase, escribir el método `finalize ()` y un programa que cree objetos, que después se pierdan las referencias a los objetos creados y se active el método `finalize ()`.

```
class Operador
{
    public float memoria;
    public Operador ( )
```

```
{
    System.out.println("Activar maquina
operator");
    memoria = 0.0F;
}
public float sumar(float f)
{
    memoria += f;
    return memoria;
}
```

22.5 Se desea realizar una clase `vector3d` que permita manipular vectores de tres componentes (coordenadas x, y, z) de acuerdo con las siguientes normas:

- Solo posee un método constructor.
- Tiene un método miembro `equals ()` que permite saber si dos vectores tienen sus componentes o coordenadas iguales.

22.6 Realizar la clase `Complejo` que permita la gestión de números complejos (un número complejo consta de dos números reales `double`: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- establecer () permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- imprimir () realiza la visualización formateada de un `Complejo`.
- agregar () (sobrecargado) para añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

22.7 Añadir a la clase `Complejo` del ejercicio 22.6 las siguientes operaciones:

- Suma: $a + c = (A+C, (B+D)i)$.
- Resta: $a - c = (A-C, (B-D)i)$.
- Multiplicación: $a*c = (A*C-B*D, (A*D+B*C)i)$
- Multiplicación: $x*c = (x*C, x*D)i$, donde x es real.
- Conjugado: $\bar{a} = (A, -Bi)$.

donde $a = A+Bi$; $c = C+Di$

22.8 Implementar la clase `Hora`. Cada objeto de esta clase representa una hora específica del día, almacenando las horas, minutos y segundos como enteros. Se ha de incluir un constructor, métodos de acceso, un método `adelantar (int h, int m, int s)` para adelantar la hora actual de un objeto existente, un método `reiniciar (int h, int m, int s)` que reinicializa la hora actual de un objeto existente y un método `imprimir ()`.



Problemas

22.1 Implementar la clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, métodos de acceso, una método `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, una método `adelantar (int d, int m, int a)` para avanzar a una fecha existente (día, d, mes, m, y año a) y un método `visualizar ()`. Escribir un método de utilidad, `normalizar ()`, que asegure que los miembros dato están en el rango correcto $1 \leq \text{año}, 1 \leq \text{mes} \leq 12, \text{día} \leq \text{días}(\text{Mes})$, donde `días(Mes)` es

otro método que devuelve el número de días de cada mes. Escribir un programa Java que cree objetos de tipo `Fecha` y realice las operaciones; `reiniciar`, `adelantar`, `normalizar`, y mostrar en pantalla, la fecha.

22.2 Ampliar el programa anterior de modo que pueda aceptar años bisiestos. **Nota:** un año es bisiesto si es divisible entre 400, o si es divisible entre 4 pero no entre 100. Por ejemplo los años 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.



Programación orientada a objetos en Java. Herencia y polimorfismo

Contenido

- 23.1 Clases derivadas
- 23.2 Herencia pública
- 23.3 Constructores en herencia
- 23.4 Métodos y clases no derivables: atributo `final`
- 23.5 Conversiones entre objetos de clase base y clase derivada

23.6 Métodos abstractos

- 23.7 Polimorfismo
- 23.8 Interfaces
 - › Resumen
 - › Ejercicios
 - › Problemas

Introducción

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base a clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, de bajo costo, adaptable y reutilizable.

Conceptos clave

- › Clase abstracta
- › Clase base
- › Clase derivada
- › Constructor
- › Herencia
- › Herencia múltiple
- › Herencia simple
- › Ligadura dinámica
- › Método abstracto
- › Polimorfismo
- › Relación es-un

23.1 Clases derivadas

La *herencia* o *relación es-un*, es la relación que existe entre dos clases, una es la clase denominada *derivada* que se crea a partir de otra ya existente, denominada *clase base*. La nueva clase *hereda* de la clase ya existente. Por ejemplo, si existe una clase *Figura* y se desea crear una clase *Triángulo*, esta clase puede derivarse de *Figura* ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. *Triángulo* es-un tipo de *Figura*. Otro ejemplo, puede ser *Programador* que es-un tipo de *Empleado*.

Evidentemente, la *clase base* y la *clase derivada* tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría

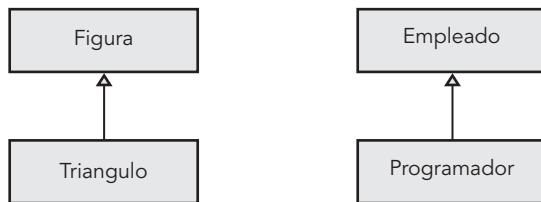


Figura 23.1 Clases derivadas.

mucho de lo que ya se ha escrito para la clase base. Java soporta el mecanismo de *extensión* (*extends*) que permite crear clases derivadas o clases que son extensión de otra clase, de modo que la nueva clase *hereda* todos los miembros datos y los métodos que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir la palabra reservada `extends` y a continuación el nombre de la clase base de la que se deriva. El formato de la declaración es:

class *nombre* *clase* extends *nombre* *clase* *base*

Regla

En Java se debe incluir la palabra reservada `extends` en línea de la declaración de la clase derivada. Esta palabra reservada produce que todos los miembros no privados (`private`) de la clase base se hereden en la clase derivada.

Ejemplo 23.1

Declaracin de las clases Programador y Triangulo.

```
1. class Programador extends Empleado
{
    public miembro público
    // miembros públicos
    private miembro privado
    // miembros privados
}
2. class Triangulo extends Figura
{
    public
    // miembros públicos
    protected
    // miembros protegidos
    ...
}
```

Una vez creada la clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

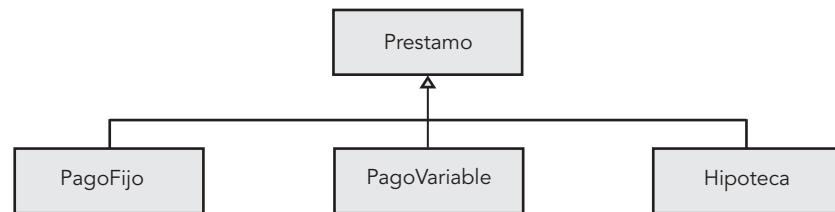
```
class Director extends Empleado
{
    public nuevo método
    ...
    private nuevo miembro
    ...
}
```

La declaración de la clase `Director` solo tiene que especificar los nuevos miembros (métodos y datos). Todos los métodos miembro y los miembros dato de la clase `Empleado` (no privados) son heredados automáticamente por la clase `Director`. Por ejemplo, el método `calcular_salario()` de `Empleado` se aplica automáticamente a `Director`:

```
Director d = new Director();
d.calcular_salario(325000);
```

Considerar una clase `Prestamo` y tres clases derivadas de ella: `PagoFijo`, `PagoVariable` e `Hipoteca`.

Ejemplo 23.2



La clase `Prestamo` es la clase base de la `PagoFijo`, `PagoVariable` e `Hipoteca`, se considera que es una *clase abstracta*, en ella se agrupan los métodos comunes a todo tipo de préstamo: hipoteca, La declaración de `Prestamo`:

```
abstract class Prestamo
{
    final int MAXTERM = 22;
    protected float capital;
    protected float tasaInteres;
    public void prestamo(float p, float r) { ... }
    abstract public int crearTablaPagos(float mat [ ] [ ]);
}
```

Las variables `capital` y `tasaInteres` no se repiten en la clase derivada. La declaración de las clases derivadas:

```
class PagoFijo extends Prestamo
{
    private float pago; // cantidad mensual a pagar por cliente
    public PagoFijo (float x, float v, float t) { ... }
    public int crearTablaPagos(float mat [ ] [ ]) { ... }
}
class PagoVariable extends Prestamo
{
    private float pago; // cantidad mensual a pagar por cliente
    public PagoVariable (float x, float v, float t) { ... }
    public int crearTablaPagos(float mat [ ] [ ]) { ... }
}
class Hipoteca extends Prestamo
{
    private int numRecibos;
    private int recibosPorAnyo;
    private float pago;
    public Hipoteca(int a, int g, float x, float p, float r) { ... }
    public int crearTablaPagos(float mat [ ] [ ]) { ... }
}
```

Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es la siguiente:

```
Nombre de la clase derivada      Palabra reservada para indicar derivación
class ClaseDerivada extends ClaseBase
{
    // miembros específicos de la clase derivada
}
```

Nombre de la clase base

Los miembros `private` de clase base son los únicos que no hereda la clase derivada, no se puede acceder a ellos desde métodos de clase derivada. Los miembros con visibilidad `public`, `protected` o la visibilidad *por defecto* (sin especificador de visibilidad) se incorporan a la clase derivada con la misma visibilidad que tienen en la clase base. Por ejemplo:

```
package personas;
public class Persona
{
    // miembros de la clase
}
```

La clase `Persona` se puede utilizar en otros *paquetes* como clase base, o bien para crear objetos.

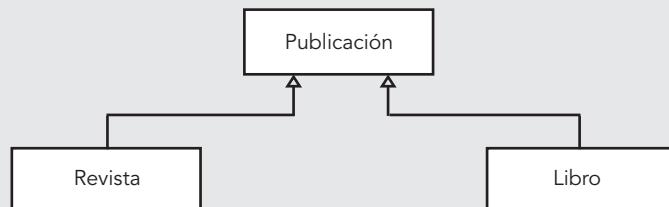
```
package empresa;
import personas.*;
public class Becario extends Persona
{
    //
}
```



Ejercicio 23.1

Representar la jerarquía de clases de publicaciones que se distribuyen en una librería: revistas, libros, etcétera.

Todas las publicaciones tienen en común la editorial y su fecha de publicación. Las revistas tienen una determinada periodicidad lo que implica el número de ejemplares que se publican al año, y, por ejemplo, el número de ejemplares que se ponen en circulación controlados oficialmente (en España la OJD). Los libros tienen como características específicas el código de ISBN y el nombre del autor.



```
class Publicacion
{
    public void nombrarEditor(String nomE){...}
    public void ponerFecha(long fe) {...}
    private String editor;
    private long fecha;
}
class Revista extends Publicacion
```

```

{
    public void fijarnumerosAnyo(int n) {...}
    public void fijarCirculacion(long n) {...}
    private int numerosPorAnyo;
    private long circulacion;
}
class Libro extends Publicacion
{
    public void ponerISBN(String nota) {...}
    public void ponerAutor(String nombre) {...}
    private String isbn;
    private String autor;
}

```

Con esta declaración, un objeto `Libro` contiene miembros datos y métodos heredados de la clase `Publicacion`, así como `isbn` y nombre del `autor`. En consecuencia serán posibles las siguientes operaciones:

```

Libro lib = new Libro ( );
lib.nombrarEditor ("McGraw-Hill");
lib.ponerFecha (990606);
lib.ponerISBN ("84-481-2015-9");
lib.ponerAutor ("Mackoy, José Luis");

```

Para objetos de tipo `Revista`:

```

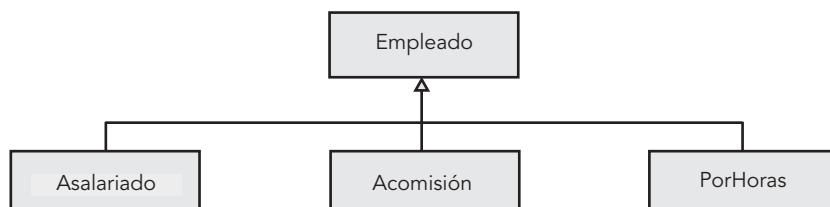
Revista rev = new Revista ( );
rev.fijarNumerosAnyo (12);
rev.fijarCirculacion (300000);

```

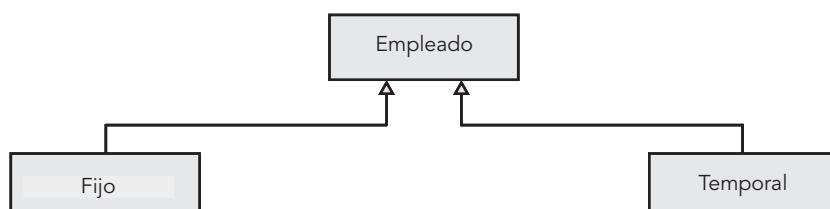
Diseño de clases derivadas

En el diseño de una aplicación orientada a objetos no siempre resulta fácil establecer la relación de herencia más óptima entre clases. Consideremos, por ejemplo a los empleados de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (*discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

Una clasificación de los empleados basada en el modo de pago puede dividirlos: empleados con salario mensual fijo, empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas.



Si el criterio de clasificación es la duración del contrato, los empleados se dividen en: fijos o eventuales.



Una dificultad añadida a la que se enfrenta el diseñador es que un mismo objeto, en el supuesto anterior un mismo empleado, puede pertenecer a diferentes grupos. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con contrato fijo puede ser remunerado por horas y un empleado temporal mediante comisiones. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones?, ¿esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada solo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

Sobrecarga de métodos en la clase derivada

La sobrecarga de métodos se produce cuando al definir un método en una clase tiene el mismo nombre que otro de la misma clase pero distinto número o tipo de argumentos. En la sobrecarga no interviene el tipo de retorno. La siguiente clase tiene métodos sobrecargados:

```
class Ventana
{
    public void copiar(Ventana w) {...}
    public void copiar(String p, int x, int y) {...}
}
```

Una clase derivada puede redefinir un método de la clase base sin tener exactamente la misma cabecera (*signatura*), teniendo el mismo nombre pero distinta la lista de argumentos. Esta redefinición en la clase derivada no oculta al método de la clase base, sino que da lugar a una sobrecarga del método heredado en la clase derivada.

La clase `VentanaEspecial` derivada de `Ventana`, define el método `copiar()` con distintos argumentos que los métodos `copiar()` de `Ventana`; no se anulan los métodos sino que están sobrecargados en la clase derivada.

```
class VentanaEspecial extends Ventana
{
    public void copiar(char c,int veces,int x,int y) {...}
}
```



Ejemplo 23.3

Se declara una clase base con el método `escribe()` y una clase derivada con el mismo nombre del método pero distintos argumentos. La clase con el método `main()` crea objetos y realiza llamadas a los métodos sobrecargados de la clase derivada.

```
class BaseSobre
{
    public void escribe(int k)
    {
        System.out.print("Método clase base, argumento entero: ");
        System.out.println(k);
    }
    public void escribe(String a)
    {
        System.out.print("Método clase base, argumento cadena: ");
        System.out.println(a);
    }
}
class DerivSobre extends BaseSobre
{
    public void escribe(String a, int n)
    {
        System.out.print("Método clase derivada, dos argumentos: ");
        System.out.println(a + " " + n);
    }
}
```

```

    }
}

public class PruebaSobre
{
    public static void main(String [ ] ar)
    {
        DerivSobre dr = new DerivSobre ( );
        dr.escribe("Cadena constante ",50);
        dr.escribe("Cadena constante ");
        dr.escribe(50);
    }
}

```

Ejecución

Método clase derivada, dos argumentos: Cadena constante 50
 Método clase base, argumento cadena: Cadena constante
 Método clase base, argumento entero: 50

23.2 Herencia pública

En una clase existen secciones *públicas*, *privadas*, *protegidas* y la visibilidad por defecto que se denomina *amigable*. Java considera que la herencia es siempre pública. *Herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y protegidos de su clase base, los elementos con visibilidad *amigable* son accesibles desde cualquier clase del mismo paquete, no son visibles en clases derivadas de otros paquetes.

Una clase derivada no puede acceder a variables y métodos privados de su clase base. Una clase base utiliza elementos protegidos para de esa manera ocultar los detalles de la clase respecto a clases no derivadas de otros paquetes (véase tabla 23.1).

Las clases para ser visibles desde otro paquete se declara con el modificador `public`, en caso contrario la clase está restringida al paquete donde se declara.

Formato

modificador **class** ClaseDerivada **extends** ClaseBase
 {
 // miembros propios de la clase derivada
 }

opcional

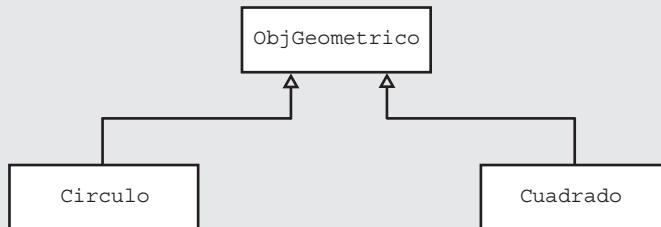
Tabla 23.1 Acceso a variables y métodos según visibilidad.

Tipo de elemento	¿Accesible a clase de paquete (package)?	¿Accesible a clase derivada?	¿Accesible a clase derivada de otro paquete?
public	sí	sí	sí
protected	sí	sí	sí
private	no	no	no
<i>(default)</i>	sí	sí	no



Ejercicio 23.2

Considérese la siguiente jerarquía:



La declaración de las clases se agrupa en el paquete `figuras`.

```

package figuras;
public class ObjGeometrico
{
    public ObjGeometrico(double x, double y)
    {
        px = x;
        py = y;
    }
    public ObjGeometrico()
    {
        px = py = 0;
    }
    public void imprimirCentro()
    {
        System.out.println("(" + px + "," + py + ")");
    }
    protected double px, py;
}
  
```

Un círculo se caracteriza por el centro y su radio. Un cuadrado también por su centro y uno de sus cuatro vértices. Entonces, las clases `Circulo` y `Cuadrado` se declaran derivadas de `ObjGeometrico`.

```

package figuras;
public class Circulo extends ObjGeometrico
{
    public Circulo(double x, double y, double r)
    {
        super(x,y);           // llama a constructor de la clase base
        radio = r;
    }
    public double area()
    {
        return PI * radio * radio;
    }
    private double radio;
    private final double PI = 3.14159;
}
package figuras;
public class Cuadrado extends ObjGeometrico
{
  
```

```

public Cuadrado(double xc, double yc, double t1, double t2)
{
    super(xc,yc);      // llama a constructor de la clase base
    x1 = t1;
    y1 = t2;
}
public double area( )
{
    double a, b;
    a = px - x1;
    b = py - y1;
    return 2 * (a * a + b * b);
}
private double x1, y1;
}

```

Todos los miembros públicos de la clase base `ObjGeometrico` también son públicos en la clase derivada `Cuadrado`. Por ejemplo, se puede ejecutar:

```
Cuadrado c = new Cuadrado (3, 3.5, 26.37, 3.85);
c.imprimirCentro ( );
```

La siguiente aplicación utiliza las clases `Cuadrado` y `Circulo`:

```

import figuras.*;
public class PruebaFiguras
{
    public static void main (String [ ] ar)
    {
        Circulo cr = new Circulo (2.0, 2.5, 2.0);
        Cuadrado cd = new Cuadrado (3.0, 3.5, 26.37, 3.85);
        System.out.print ("Centro del circulo : ");
        cr.imprimirCentro ( );
        System.out.println ("Centro del cuadrado : ");
        cd.imprimirCentro ( );
        System.out.println ("Area del circulo : " + cr.area ( ) );
        System.out.println ("Area del cuadrado : " + cd.area ( ) );
    }
}

```

Ejecución

```

Centro del circulo : (2.0,2.5)
Centro del cuadrado : (3.0,3.5)
Area del circulo : 12.5666
Area del cuadrado : 3.9988

```

Regla

La herencia en Java es siempre pública, los miembros de la clase derivada, heredados de la clase base, tienen la misma protección que en la clase base. La herencia pública modela directamente la relación *es-un*.

23.3 Constructores en herencia

Un objeto de una clase derivada consta de la porción correspondiente de su clase base y de los miembros propios. En consecuencia, al construir un objeto de clase derivada primero se construye la parte de su clase base, llamando a su *constructor*, y a continuación se inicializan los miembros propios de la clase derivada.

Regla

1. El constructor de la clase base se invoca antes del constructor de la clase derivada.
2. Si una clase base es, a su vez, una clase derivada, se invocan siguiendo la misma secuencia: constructor base, constructor derivada.
3. Los métodos que implementan a los constructores no se heredan.
4. Si no se especifica el constructor de la clase base, se invoca al constructor sin argumentos.



Ejemplo 23.4

Se declaran dos clases base, una clase derivada de una clase base y una clase derivada de una clase base que a su vez es derivada.

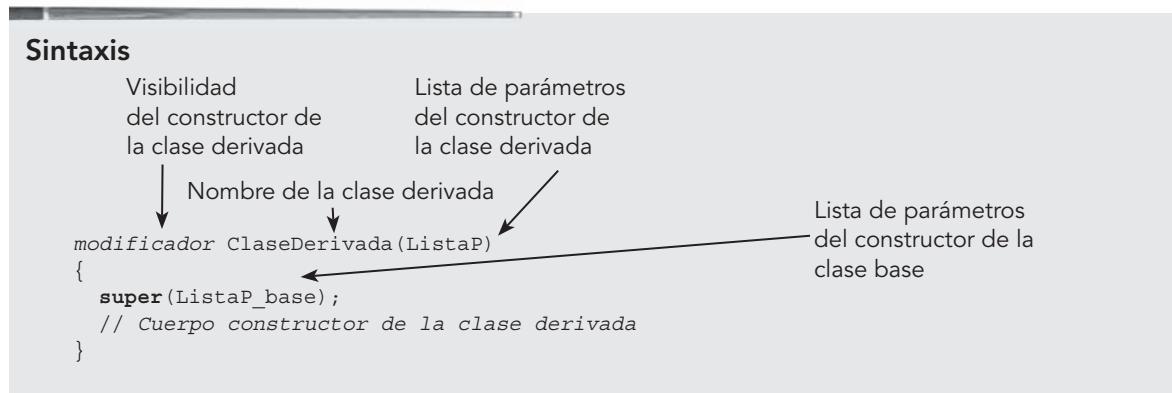
```
class B1
{
    public B1( ) { System.out.println("Constructor-B1"); }
}
class B2
{
    public B2( ) { System.out.println("Constructor-B2"); }
}
class D extends B1
{
    public D( ) { System.out.println("Constructor-D"); }
}
class H extends B2
{
    public H( ) { System.out.println("Constructor-H"); }
}
class Dh extends H
{
    public Dh( ) { System.out.println("Constructor-Dh"); }
}
class Constructor
{
    public static void main(String [ ] ar)
    {
        D d1 = new D( );
        System.out.println("_____ \n");
        Dh d2 = new Dh( );
    }
}
```

Ejecución

```
Constructor-B1
Constructor-D
_____
Constructor-B2
Constructor-H
Constructor-Dh
```

Sintaxis

La primera línea del constructor de la clase derivada debe incluir una llamada al constructor de la clase base; esta llamada se hace a través de `super()`. Los argumentos que se vayan a transmitir se incluyen en la lista de argumentos de la clase base.



Por ejemplo:

```

class Persona
{
    protected String nombre;
    public Persona(String nm) // constructor de Persona
    {
        nombre = new String(nm);
    }
}
Clase derivada de Persona:
class Juvenil extends Persona
{
    private int edad;
    public Juvenil(String suNombre, int ed)
    {
        super(suNombre); // llamada a constructor de clase base
        edad = ed;
    }
}
  
```

La llamada al constructor de la clase base ha de ser la primera sentencia del cuerpo del constructor de la clase derivada. Si este no tiene la llamada explícita, supone que se hace una llamada al constructor sin argumentos de la clase base, pudiendo dar error de no existir tal constructor.

La clase `Punto3D` es una clase que deriva de la clase `Punto`.

En la clase `Punto3D` se definen dos constructores, el primero sin argumentos, que inicializa un objeto al punto tridimensional $(0, 0, 0)$; esto lo hace en dos etapas, primero llama al constructor por defecto de `Punto`, y a continuación asigna 0 a z (tercera coordenada). El segundo constructor llama, con `super(x1, y1)`, al constructor de `Punto`.

```

class Punto
{
}
  
```

Ejemplo 23.5



```

public Punto ( )
{
    x = y = 0;
}
public Punto(int xv, int yv)
{
    x = xv;
    y = yv;
}
protected int x, y;
}
class Punto3D extends Punto
{
    public Punto3D ( )
    {
        // llamada implícita al constructor por defecto de Punto
        // Se podría llamar explícitamente: super(0,0);
        fijarZ(0);
    }
    public Punto3D(int x1, int y1, int z1)
    {
        super(x1,y1);
        fijarZ(z1);
    }
    private void fijarZ(int z) {this.z = z;}
    private int z;
}

```

Referencia a la clase base: `super`

Los métodos heredados de la clase base pueden ser llamados desde cualquier método de la clase derivada, simplemente se escribe su nombre y la lista de argumentos. Puede ocurrir que haya métodos de la clase base que no interese que sean heredados en la clase derivada, debido a que se quiera que tenga una funcionalidad adicional. Para ello se sobreescribe el método en la clase derivada. Los métodos en las clases derivadas con la *misma signatura* (*igual nombre, tipo de retorno y número y tipo de argumentos*) que métodos de la clase base, *anulan* (reemplazan) a las versiones de la clase base. El método de la clase base ocultado puede ser llamado desde cualquier método de la clase derivada con la referencia `super` seguida de un punto (.) el nombre del método y la lista de argumentos: `super.metodo(argumentos)`. La palabra reservada `super` permite acceder a cualquier miembro de la clase base (siempre que no sea privado).

Regla

Una clase derivada oculta un método de la clase base redefiniendo el método con la misma signatura: mismo nombre, igual tipo de retorno, y la misma lista de argumentos.



Ejemplo 23.6

La clase `Fecha` define el método `escribir()`, la clase `FechaJuliana` hereda de la clase `Fecha` y sobreescribe el método.

```

class Fecha
{
    private int d, m, a;
    public Fecha(int dia, int mes, int anyo)
    {
        d = dia;
    }
    public void escribir()
    {
        System.out.println("Día: " + d);
        System.out.println("Mes: " + mes);
        System.out.println("Año: " + anyo);
    }
}

```

```

        m = mes ;
        a = anyo;
    }
    public Fecha( ) { }
    public void escribir( )
    {
        System.out.println("\n" + d + " / " + m + " / " + a);
    }
}
class FechaJuliana extends Fecha
{
    private int numDias;
    public FechaJuliana(int dia, int mes, int anyo){...}
    public void escribir( )
    {
        super.escribir( ); // llamada al método de la clase Fecha
        System.out.println("Dias transcurridos: " + numDias);
    }
}

```



23.4 Métodos y clases no derivables: atributo `final`

En el contexto de herencia la palabra reservada `final` se emplea para proteger la redefinición de los métodos de la clase base. Un método con el atributo `final` no puede ser redefinido en las clases derivadas. Por ejemplo:

```

class Ventana
{
    final public int numpixels ( )
    {
        //...
    }
    public void rellenar ( )
    {
        //...
    }
}

```

El método `numpixels ()` no puede ser redefinido por una clase derivada de `Ventana`, incluso si el compilador de Java detecta un intento de redefinición genera un mensaje de error.

El concepto de protección que implica `final` también se extiende a las clases. Una clase que se desea no sea clase base de otras clases se declara con el atributo `final`.

```

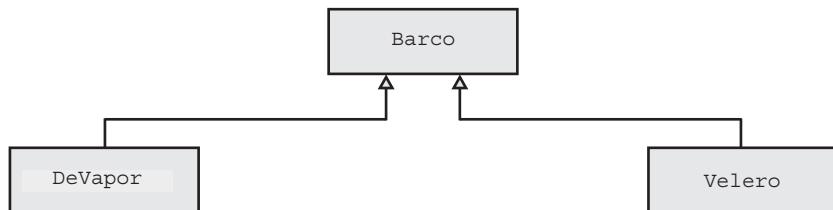
final class Sorteo
{
    //...
}

```

La clase `Sorteo` no puede estar formando parte de una jerarquía de clases, ha de ser una clase independiente. Las clases que *envuelven* a los tipos básicos (`Integer`, `Boolean` ...) son clases que no se pueden derivar, están declaradas como `final`.

23.5 Conversiones entre objetos de clase base y clase derivada

Al declarar una clase como *extensión* o *derivada* de otra clase, los objetos de la clase derivada son a su vez objetos de la clase base. En la siguiente jerarquía:



Un objeto *Velero* puede verse según la figura 23.2.

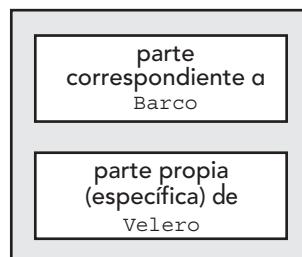


Figura 23.2 Ejemplo de objeto derivado.

Un objeto *Velero* es a su vez un objeto *Barco*, entonces se puede afirmar que todo *Velero* es un *Barco*. Por esa razón el lenguaje Java convierte automáticamente una referencia a objeto de la clase derivada a referencia a clase base. Por ejemplo:

```

Barco mr;
Velero v = new Velero ( );
DeVapor w = new DeVapor ( );
mr = v; // conversión automática
mr = w; // conversión automática
  
```

La conversión inversa, de un objeto de la clase base a referencia de la clase derivada, no es posible, es un error.



Ejercicio 23.3

Se declaran las clases correspondientes a la jerarquía *Barco*, *DeVapor* y *Velero*; con un método común, *alarma* (), que es redefinido en cada clase derivada. En el programa se define un arreglo de referencias a *Barco*, se crean objetos de las clases derivadas *DeVapor* y *Velero*, asigna esos objetos al arreglo y por último se llama al método redefinido.

```

class Barco
{
    public Barco ( )
    {
        System.out.print("\tSe crea parte de un barco. ");
    }

    public void alarma ( )
    {
        System.out.println("\tS.O.S desde un Barco");
    }
}
class DeVapor extends Barco
{
}
  
```

```

public DeVapor( )
{
    System.out.println("Se crea la parte del barco de vapor. ");
}
public void alarma( )
{
    System.out.println("\tS.O.S desde un Barco de Vapor");
}
}
class Velero extends Barco
{
    public Velero( )
    {
        System.out.println("Se crea la parte del barco velero. ");
    }
    public void alarma( )
    {
        System.out.println("\tS.O.S desde un Velero");
    }
}
public class AlarmasDeBarcos
{
    public static void main(String [ ] ar)
    {
        Barco [ ] bs = new Barco[2];
        DeVapor mss = new DeVapor( );
        Velero vss = new Velero( );
        bs[0] = mss; bs[1] = vss;
        for (int i = 0; i < 2; )  bs[i++].alarma( );
    }
}

```

Ejecución

```

Se crea parte de un barco. Se crea la parte del barco de vapor.
Se crea parte de un barco. Se crea la parte del barco velero.
S.O.S desde un Barco de Vapor
S.O.S desde un Velero

```

23.6 Métodos abstractos

Si la palabra reservada `abstract` precede a la declaración de un método, este método se denomina *abstracto*, y le indica al compilador que será definido (implementado su cuerpo) en una clase derivada (no necesariamente en la derivada inmediata). El uso común de los métodos abstractos es la declaración de clases abstractas y la implementación del *polimorfismo*.

Por ejemplo, en el contexto de figuras geométrica, la clase `Figura` es la clase base de la que derivan otras clases, como `Rectangulo`, `Circulo` y `Triangulo`. Cada figura debe tener la posibilidad de calcular su área y poder dibujarla; por ello, la clase `Figura` declara los métodos `calcularArea()` y `dibujar()` abstractos y así obligar a las clases derivadas a redefinirlos, a particularizarlos.

```

class Figura
{
    public abstract double calcularArea( );
    public abstract void dibujar( );
}

```

```

    // otros métodos miembro que definen una interfaz a todos los
    // tipos de figuras geométricas
}

```

La clase `Figura` es una clase abstracta, toda clase con uno o más métodos abstractos es abstracta y en Java se declara con la palabra reservada `abstract`. La declaración correcta de `Figura`:

```
abstract class Figura
```

Las clases `Circulo` y `Rectangulo`, derivadas de `Figura`, deben definir los métodos `calcularArea()` y `dibujar()` en cada clase. Para la clase `Circulo`:

```

class Circulo extends Figura
{
    public double calcularArea()
    {
        return (PI * radio * radio);
    }
    public void dibujar()
    {
        // ...
    }
    private double xc, yc;      // coordenada del centro
    private double radio;       // radio del círculo
}

```

Una clase que no redefina un *método abstracto* heredado se convierte en clase abstracta.

Clases abstractas

Las clases abstractas representan conceptos generales, engloban las características comunes de un conjunto de objetos. `Persona`, en un contexto de trabajadores, es una clase abstracta que engloba las propiedades y métodos comunes a todo tipo de persona que trabaja para una empresa. En Java el modificador `abstract` declara a una clase abstracta:

```
abstract class NombreClase { // ... }
```

Por ejemplo,

```

public abstract class Persona
{
    private String apellido;
    //
    public void identificacion(String a, String c){ ... }
}

```

Las clases abstractas declaran métodos y variables instancia, y normalmente tienen métodos abstractos. Una clase que tiene un método abstracto debe declararse abstracta.

Una característica importante es que no se pueden definir objetos, instanciar, de una clase abstracta. El compilador da un error siempre que se intenta crear un objeto de una clase abstracta.

```

public abstract class Metal { ... }
Metal mer = new Metal(); // error: no se puede instanciar de clase
                        // abstracta

```

Las clases abstractas están en lo más alto de la jerarquía de clases, son superclases base, y por consiguiente siempre se establece una conversión automática de clase derivada a clase base abstracta.

Se define un *array* de la clase abstracta *Figura* y se crean objetos de las clases concretas *Rectangulo* y *Circulo*.

Ejemplo 23.7

```
Figura [ ] af = new Figura [10];
for (int i = 0; i < 10; i++)
{
    if (i%2 ==0)
        af[i] = new Rectangulo ( );
    else
        af[i] = new Circulo ( );
}
```

Normas de las clases abstractas

- Una clase abstracta se declara con la palabra reservada *abstract* como prefijo en la cabecera de la clase.
- Una clase con al menos un método abstracto es una clase abstracta y hay que declararla como abstracta.
- Una clase derivada que no redefine un método abstracto es también clase abstracta.
- Las clases abstractas pueden tener variables instancia y métodos no abstractos.
- No se pueden crear objetos de clases abstractas.

Ligadura dinámica mediante métodos abstractos

La llamada a los métodos de las clases derivadas que son una redefinición de un método abstracto se hace de igual forma que cualquier otro método de la clase. Considérese la siguiente jerarquía de clases:

```
abstract class Libro
{
    abstract public int difusion ( );
    abstract public void escribirDescripcion ( );
}
class LibroImpreso extends Libro
{
    public int difusion ( ) { ... }
    public void escribirDescripcion ( ) { ... }
}
class LibroElectronico extends Libro
{
    public int difusion ( ) { ... }
    public void escribirDescripcion ( ) { ... }
}
```

Y la llamada a los métodos a través de una referencia a *Libro*:

```
final int NUMLIBROS = 11;
Libro [ ] w = new Libro[NUMLIBROS]; //referencias a 11 objetos libro
// creación de objetos libro impreso o electrónico
w[0] = new LibroImpreso("La logica de lo impensable",144,67.0);
w[1] = new LibroElectronico("Catarsis",220);
// escribe la descripción de cada tipo de libro
for (int i = 0 ; i < NUMLIBROS; i++)
    w[i].escribirDescripcion ( );
```

En estas llamadas Java no puede determinar cuál es la implementación específica del método *escribirDescripcion ()* que se ha de llamar. Se determina en tiempo de ejecución, es la *ligadura dinámica* o *vinculación tardía*, según el objeto (*LibroImpreso* o *LibroElectronico*) al que referencia *w[i]*.

Norma

Se puede utilizar una variable referencia a la clase base en lugar de una referencia a cualquier clase derivada, sin una conversión explícita de tipos.

11

Ejercicio 23.4

Se declaran la clase base abstracta A, con un método abstracto y otro no. Las clases derivadas de A redefinen el método abstracto. En `main ()` se crean objetos y se asignan a una variable de la clase A.

```
abstract class A
{
    public abstract void dinamica ( );
    public void estatica ( )
    {
        System.out.println("Método con ligadura estática de la clase A");
    }
}
// define clase B, derivada de A, redefiniendo el método abstracto
class B extends A
{
    public void dinamica ( )
    {
        System.out.println("Método dinámico de clase B");
    }
}
// la clase C redefine el método abstracto
class C extends A
{
    public void dinamica ( )
    {
        System.out.println("Método dinámico de clase C");
    }
}
// clase con el método main
public class Ligadura
{
    static public void main(String [ ] ar)
    {
        A a;
        B b = new B ( );
        C c = new C ( );
        System.out.print ("Métodos llamados con objeto b desde");
        System.out.println (" referencia de la clase A");
        a = b;
        a.dinamica ( );
        a.estatica ( );
        System.out.print ("Métodos llamados con objeto c desde");
        System.out.println (" referencia de la clase A");
        a = c;
        a.dinamica ( );
        a.estatica ( );
    }
}
```

```

    }
}

```

Ejecución

```

Métodos llamados con objeto b desde referencia de la clase A
Método dinámico de la clase B
Método con ligadura estática de la clase A
Métodos llamados con objeto c desde referencia de la clase A
Método dinámico de la clase C
Método con ligadura estática de la clase A

```

23.7 Polimorfismo

En POO, el *polimorfismo* permite que diferentes objetos respondan de modo distinto al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de herencia.

El polimorfismo se establece con la ligadura dinámica de métodos. Con la ligadura dinámica no es preciso decidir el tipo de objeto hasta el momento de la ejecución. En el ejercicio 23.4 se declara `abstract` el método `dinamica()` en la clase `A`, se ha indicado al compilador que este método se puede llamar por una referencia de `A` mediante la ligadura dinámica. La variable `a` en un momento referencia a un objeto de `B` y en otro momento a un objeto de `C`. El programa determina el tipo de objeto de `a` en tiempo de ejecución, de tal forma que el mismo *mensaje*, `dinamica()`, se comporta de manera diferente según que `a` refiera a un objeto de `B` o a un objeto de `C`.

*El polimorfismo se puede representar con un array de elementos que se refieren a objetos de diferentes tipos (clases), como sugiere Meyer.*¹

Uso del polimorfismo

La forma de usar el polimorfismo es a través de referencias a la clase base. Si, por ejemplo, se dispone de una colección de objetos `Archivo` en un arreglo, este almacena referencias a objetos `Archivo` que apuntan a cualquier tipo de archivo. Cuando se actúa sobre estos archivos, simplemente basta con recorrer el arreglo e invocar al método apropiado mediante la referencia a la instancia. Naturalmente, para realizar esta tarea los métodos de la clase base deben ser declarados como abstractos en la clase `Archivo`, que es la clase base, y redefinirse en las clases derivadas (`ASCII`, `Grafico...`).

Para poder utilizar polimorfismo en Java se deben seguir estas reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por los métodos miembro declarados como abstractos en la clase base.
2. Las implementaciones específicas de los métodos abstractos se deben hacer en las clases derivadas. Cada clase derivada puede tener su propia versión del método. Por ejemplo, la implementación del método `añadir()` varía de un tipo de archivo a otro.
3. Las instancias de estas clases se manejan a través de una referencia a la clase base. Este mecanismo es la ligadura dinámica y es la esencia de polimorfismo en Java.

Realmente no es necesario declarar los métodos en la clase base abstractos si después se redefinen (misma *signatura*) en la clase derivada.

Ventajas del polimorfismo

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación. Este es el caso de Java.

¹ Meyer, B., *Object-Oriented Software Construction*. Prentice-Hall, New York, 1998.

Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas.** El uso más común del polimorfismo es derivar clases especializadas de clases que han sido definidas. Por ejemplo, una clase Cuadrado es una especialización de la clase Rectángulo (cualquier cuadrado es un tipo de rectángulo). Esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.
- **Estructuras de datos heterogéneos.** A veces es muy útil poder manipular conjuntos similares de objetos. Con polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos, que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.
- **Gestión de una jerarquía de clases.** Las jerarquías de clases son colecciones de clases altamente estructuradas, con relaciones de herencia que se pueden *extender* fácilmente.

23.8 Interfaces

Java incorpora una construcción del lenguaje, llamada *interface* (*interfaz*), que permite declarar un conjunto de constantes y de cabeceras de métodos abstractos. Estos deben de implementarse en las clases y constituyen su interfaz. En cierto modo, es una forma de declarar que todos los métodos de una clase son públicos y abstractos, con ello se especifica el comportamiento común de todas las clases que implementen la interfaz. La declaración de una *interface* es similar a la de una clase; en la cabecera se utiliza la palabra reservada *interface* en vez de *class*. Por ejemplo:

```
public interface NodoG
{
    boolean igual(NodoG t);
    NodoG asignar(NodoG t);
    void escribir(NodoG t);
}
```

La interfaz *NodoG* define tres métodos abstractos y además públicos. Sin embargo no debe especificarse ni *abstract* ni *public* ya que todos los métodos de una *interface* lo son.

Sintaxis

```
acceso interface NombreInterface
{
    constante1;
    ...
    constanten;
    tipo1 nombreMetodo1(argumentos);
    ...
    tipon nombreMetodon (argumentos);
}
acceso      visibilidad de la interfaz definida, normalmente public.
```

Regla

En una *interface* todos los métodos declarados son, de manera predeterminada, públicos y abstractos; por ello no está permitido precederlos de modificadores.



Ejemplo 23.8

Se declara un conjunto de métodos comunes a la estructura *ArbolB*. Además, la constante entera que indica el número máximo de claves.

```
public interface ArbolBAbstracto
{
```

```

final int MAXCLAVES = 4;
void insertar(Object clave);
void eliminar(Object clave);
void recorrer ( );
}

```

Esta interfaz muestra los métodos que toda estructura *árbol B* debe implementar.

Implementación de una interface (interfaz)

La *interface* (interfaz) especifica el comportamiento común que tiene un conjunto de clases. Dicho comportamiento se implementa en cada una de las clases, es lo que se entiende como *implementación de una interface*. Se utiliza una sintaxis similar a la derivación o extensión de una clase, con la palabra reservada `implements` en lugar de `extends`.

```

class NombreClase implements NombreInterfaz
{
    // definición de atributos
    // implementación de métodos de la clase
    // implementación de métodos de la interfaz
}

```

La clase que implementa una interfaz tiene que especificar el código (la implementación) de cada uno de los métodos de la *interface*. De no hacerlo la clase se convierte en clase abstracta y entonces debe declararse `abstract`. Es una forma de obligar a que cada método de la *interface* se defina.

Ejercicio 23.5

Considérese una jerarquía de barcos, todos tienen como comportamiento común `msgeSocorro()` y `alarma()`. Las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` implementan el comportamiento común.

Se declara la interfaz `Barco`:

```

interface Barco
{
    void alarma ( );
    void msgeSocorro(String av);
}

```

Las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` implementa la interfaz `Barco`:

```

class BarcoPasaje implements Barco
{
    private int eslora;
    private int numeroCamas = 101;
    public BarcoPasaje ( )
    {
        System.out.println("Se crea objeto BarcoPasaje.");
    }
    public void alarma ( )
    {
        System.out.println("!!! Alarma del barco pasajero !!!");
    }
    public void msgeSocorro(String av)
    {
        alarma ( );
        System.out.println("!!! SOS SOS !!!" + av);
    }
}

```

```

        }
    }
    class PortaAvion implements Barco
    {
        private int aviones = 19;
        private int tripulacion;
        public PortaAvion(int marinos)
        {
            tripulacion = marinos;
            System.out.println("Se crea objeto PortaAviones.");
        }
        public void alarma ( )
        {
            System.out.println("!!! marineros a sus puestos !!!");
        }
        public void msgSocorro(String av)
        {
            System.out.println("!!! SOS SOS !!! " + av);
        }
    }
    class Pesquero implements Barco
    {
        private int eslora;
        private double potencia;
        private int pescadores;
        String nombre;
        public Pesquero(int tripulacion)
        {
            pescadores = tripulacion;
            System.out.println("Se crea objeto Barco Pesquero.");
        }
        public void alarma ( )
        {
            System.out.println("!!! Alarma desde el pesquero " +
                               nombre + " !!!");
        }
        public void msgSocorro(String av)
        {
            System.out.println("!!! SOS SOS !!!, " + av);
        }
    }
}

```

Múltiples interfaces

Java no permite que una clase derive de dos o más clases, no permite la *herencia múltiple*. Sin embargo, una clase sí puede implementar más de una *interface*, sí puede tener el comportamiento común de varias *interfaces*. Sencillamente, a continuación de la palabra reservada `implements` se escriben las *interfaces* separadas por comas. La clase tiene que implementar los métodos de todas las *interfaces*.

Sintaxis

```

class NombreClase implements Interfaz1, Interfaz2,...,Interfazn
{
    // ...
}

```

Regla

Una clase implementa tantas *interfaces* como se desee, se deben implementar todos los métodos como `public` debido a que Java no permite reducir la visibilidad de un método cuando se *sobrescribe*.

Jerarquía de interface (interfaz)

Las *interfaces* se pueden organizar en forma jerárquica, de tal forma que los métodos sean heredados. A diferencia de las clases que solo pueden heredar de una clase base (*herencia simple*), las interfaces pueden heredar de tantas interfaces como se precise. También se utiliza la palabra reservada `extends` para especificar la herencia de interfaz.

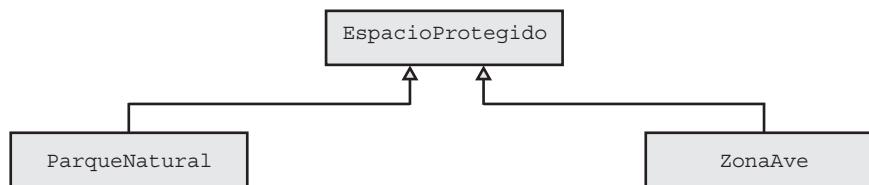
Sintaxis

```
interface SuperBase1 {...}
interface Base1 extends SuperBase1 {...}
interface Base2 extends SuperBase1 {...}

interface ComunDerivado extends Base1, Base2 {...}
```

Herencia de clases e implementación de interface

Las *interfaces* no son clases: especifican un comportamiento (métodos) que va a tener la clase que lo implementa. Por ello, una clase puede heredar de su clase base y a la vez implementar una interfaz. En la siguiente jerarquía de clases:



la clase `ParqueNatural` hereda de la clase `EspacioProtegido` y además implementa la interfaz `Parque`. El esquema para implementar este diseño es el siguiente:

```
public interface Parque {...}
public class EspacioProtegido {...}
public class ZonaAve extends EspacioProtegido {...}
public class ParqueNatural extends EspacioProtegido implements Parque{...}
```

Regla

Una clase puede heredar de otra clase e implementar una interfaz. Se ha de especificar en primer lugar la clase de la que hereda (`extends`) y a continuación la interfaz que implementa (`implements`).

Variables interface

Las *interface* al no ser clases tampoco pueden instanciarse objetos. Sí se pueden declarar variables de tipo *interface*; cualquier variable de una clase que implementa a una *interface* se puede asignar a una variable del tipo de *interface*.



Ejemplo 23.9

Se declara la interfaz `Bolsa` y clases que implementan la interfaz.

```
interface Bolsa
{
    Bolsa insertar(Object elemento);
}
public class Bolsa1 implements Bolsa
{
    public Bolsa insertar(Object e) { ... }
}
public class Bolsa2 implements Bolsa
{
    public Bolsa insertar(Object e) { ... }
}
```

En un método se puede definir una variable del tipo `interface Bolsa` y asignar un objeto `Bolsa1`, o bien un objeto `Bolsa2`.

```
Bolsa q;
q = new Bolsa1();
q.insertar("Manzana");
...
q = new Bolsa2();
q.insertar (Racional(2.5));
```



Resumen

- La relación entre clases **es-un tipo de** indica relación de herencia. Por ejemplo, **una revista es un tipo de publicación**.
- La relación **es-un** también se puede expresar como generalización-especialización, es una relación transitiva; así un *becario* es un tipo de *trabajador* y un *trabajador* un tipo de *persona*, por consiguiente un *becario* es una *persona*. Esta manera de relacionarse las clases se expresan en Java con la derivación o extensión de clases.
- Una clase nueva que se crea a partir de una clase ya existente, utilizando la propiedad de la herencia, se denomina *clase derivada* o *subclase*. La clase de la cual se hereda, se denomina *clase base* o *superclase*.
- En Java la herencia siempre es simple, además siempre es pública. La clase derivada hereda todos los miembros de la clase base excepto los **miembros privados**.
- Un objeto de una clase derivada se crea siguiendo este orden: primero se crea la parte del objeto correspondiente a la clase base y a continuación se crea la parte propia de la clase derivada. Para llamar al constructor de la clase base desde el constructor de la clase derivada se emplea la palabra reservada `super` ().
- El polimorfismo es una de las propiedades fundamentales de la orientación a objetos. Esta propiedad signi-

fica que el envío de un *mensaje* puede dar lugar a acciones diferentes dependiendo del objeto que lo recibe.

- Para implementar el polimorfismo, un lenguaje debe admitir el enlace entre la llamada a un método y el código del método en tiempo de ejecución, es la **ligadura dinámica** o **vinculación tardía**. Esta propiedad se establece en el contexto de la herencia y la redefinición de los métodos polimórficos en cada clase derivada.
- Un método abstracto (`abstract`) declarado en una clase convierte a esta en clase abstracta. Con los métodos abstractos se obliga a la clase derivada a su redefinición, en caso contrario la clase derivada también es abstracta. No se puede instanciar objetos de clases abstractas.
- Java permite declarar métodos con la propiedad de no ser redefinibles, el modificador `final` se utiliza para este cometido. También, con el modificador `final` se puede hacer que una clase no forme parte de una jerarquía.
- Las *interfaces* de Java declaran constantes y operaciones comunes a un conjunto de clases. Las operaciones son métodos abstractos que deben definir las clases que implementan la *interface*.



Ejercicios

- 23.1 Implementar una clase *Automovil* (*Carro*) dentro de una jerarquía de herencia. Considere que, además de ser un *Vehículo*, un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte*, etcétera.
- 23.2 Implementar una jerarquía de herencia de animales tal que contenga al menos tres niveles de derivación y ocho clases.
- 23.3 Implementar una jerarquía de clases de los distintos tipos de ficheros. Codificar en Java la cabecera de las clases y los métodos que se consideren polimórficos.
- 23.4 ¿Describir las diversas utilizaciones de la referencia *super*?
- 23.5 ¿Qué diferencias se pueden encontrar entre *this* y *super*?
- 23.6 Declarar una interfaz con el comportamiento común de los objetos *Avión*.
- 23.7 Declarar la interfaz *Conjunto* con las operaciones que puede realizar todo tipo de conjunto.



Problemas

- 23.1 Definir una clase base *Persona* que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: *Estudiante*, *Empleado*, *Estudiante_empleado*. Escribir un programa que lea del dispositivo estándar de entrada los datos para crear una lista de personas: *a*) general; *b*) estudiantes; *c*) empleados; *d*) estudiantes empleados. El programa deberá permitir ordenar alfabéticamente por el primer apellido.

- 23.2 Implementar una jerarquía *Empleado* de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos tres niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para empleados de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a empleados. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o a la Secretaría de Hacienda.

- 23.3 Se quiere realizar una aplicación para que cada profesor de la universidad gestione las fichas de sus alumnos. Un profesor puede impartir una o varias asignaturas y dentro de cada asignatura puede tener distintos grupos de alumnos. Los alumnos pueden ser *presente*,

ausente o *a distancia*. Al comenzar las clases, se entrega al profesor un listado con los alumnos por cada asignatura. Escribir un programa de tal forma que el listado de alumnos se introduzca por teclado y se den de alta calificaciones de exámenes y prácticas realizadas. Se podrán obtener listados de calificaciones una vez realizados los exámenes y porcentajes de aprobados.

- 23.4 El programa siguiente muestra las diferencias entre llamadas a un método redefinido y otro no.

```
class Base
{
    public void f ( ) {System.out.println("f
    () : clase base !");}
    public void g ( ) {System.out.println("g
    () : clase base !");}
}
class Derivada1 extends Base
{
    public void f ( )
    {System.out.println("f ( ) : clase De-
    rivada !");}
    public void g(int k)
    {System.out.println( "g ( ) : clase De-
    rivada !" + k);}
}
class Derivada2 extends Derivada1
{
    public void f ( )
    {System.out.println( "f ( ) : clase De-
    rivada2 !");}
}
```

```
public void g ( )
    {System.out.println ( "g( ) :clase De-
        rivada2 !" ;)
    }
class Anula
{
    public static void main(String ar [ ] )
    {
        Base b = new Base ( );
        Derivada1 d1 = new Derivada1 ( );
        Derivada2 d2 = new Derivada2 ( );
        Base p = b;
        p.f ( );
        p.g ( );
        p = d1;
        p.f ( );
        p.g ( );
        p = d2;
        p.f ( );
        p.g ( );
    }
}
```

¿Cuál es el resultado de ejecutar este programa? ¿Por qué?



Colecciones

Contenido

- 24.1 Colecciones en Java
- 24.2 Clases de utilidades: Arrays y Collections
- 24.3 Comparación de objetos:
Comparable y Comparator
- 24.4 Vector y Stack

24.5 Iteradores de una colección

- 24.6 Listas
- 24.7 Colecciones parametrizadas
 - › Resumen
 - › Ejercicios
 - › Problemas

Introducción

Java dispone de un conjunto de clases para agrupar objetos; se conocen como colecciones. Cada clase organiza los objetos de una forma particular, como un mapa, una lista, un conjunto..., en definitiva, cada clase implementa un *Tipo abstracto de datos*. Las colecciones son importantes para el desarrollo de programas profesionales dado que facilita considerablemente el diseño y construcción de aplicaciones basadas en estructuras de datos como vectores, listas, conjuntos, mapas. La clase `Vector` procede de la versión original de Java. En Java 2 potenciaron las colecciones con nuevas clases e interfaces, todas ellas en el paquete `java.util`.

24.1 Colecciones en Java

Las colecciones proporcionan programación genérica para muchas estructuras de datos. Una colección es una agrupación de objetos relacionados que forma una única entidad, por ejemplo un arreglo de objetos, un conjunto... . El arreglo, el vector, la matriz, en general la *Colección* es en sí mismo otro objeto que se debe crear. Por ejemplo:

```
class Pueblo { .... }
class Puerto { ... }
Pueblo [ ] col1 = new Pueblo[100];
Puerto [ ] col2 = new Puerto [100];
LinkedList <String> concad =
  new LinkedList <String> ( ); // Lista de cadenas
```

Conceptos clave

- › Clases contenedoras
- › Iterador
- › Object
- › Tipo abstracto

Las colecciones incluyen *clases contenedoras* para almacenar objetos, para acceder a los objetos en el interior de los contenedores y algoritmos para manipular los objetos (métodos de clases).

Las clases *Colección* guardan objetos de cualquier tipo, de hecho el elemento base es `Object` y por consiguiente, debido a conversión automática, se podrá añadir a la colección un objeto de cualquier tipo. El ejemplo 24.1 crea una colección básica, secuencial, con un arreglo, para guardar objetos de diferentes tipos.



Ejemplo 24.1

Considerar una aplicación en la que se debe almacenar n objetos del tipo `Punto2D`, `Punto3D` y `PuntoPolar`.

Se realiza un almacenamiento secuencial, para ello se declara una arreglo de tipo `Object`. De esa forma se puede asignar cualquier objeto. El problema surge al recuperar los elementos del arreglo, se debe hacer un *cast* al tipo clase, que puede ser `Punto2D`, `Punto3D` y `PuntoPolar`.

```
class Punto2D { ... }
class Punto3D { ... }
class PuntoPolar { ... }
```

Declaración y creación del array de n elementos:

```
final int N = 99;
Object [ ] rr = new Object [N];
int i = 0;
```

Asignación secuencial de objetos:

```
mas = true;
while (mas && (i < N) )
{
    int opc;
    opc = menu( );
    if (opc == 1)
        rr[i++] = new Punto2D( );
    else if (opc == 2)
        rr[i++] = new Punto3D( );
    if (opc == 3)
        rr[i++] = new PuntoPolar( );
    else mas = false;
}
```

Asignar elementos en un arreglo es una operación muy eficiente. Una de las limitaciones de los arreglos es el tamaño; al ser de tamaño fijo, si se llena hay que ampliarlo. Por contra, una característica importante de las clases *Colección* es que se redimensionan automáticamente, el programador se despreocupa de controlar el número de elementos, puede colocar tantos elementos como sea necesario.

Norma

Cuando se vaya a trabajar con tipos de datos simples, o cuando se conozca el tipo de dato de los elementos y el tamaño final, resulta muy eficiente utilizar arreglos (arrays). En cualquier otro caso, será conveniente utilizar alguna de las colecciones del paquete `java.util`.

Tipos de colecciones

En las primeras versiones del JDK, Java 1.0 y 1.1, se incorporan colecciones básicas, aunque en la mayoría de las ocasiones son suficientes. Las más importantes son `Vector`, `Stack`, `Dictionary`, `HashTable` y la interfaz `Enumeration` para recorrer los elementos de una colección.

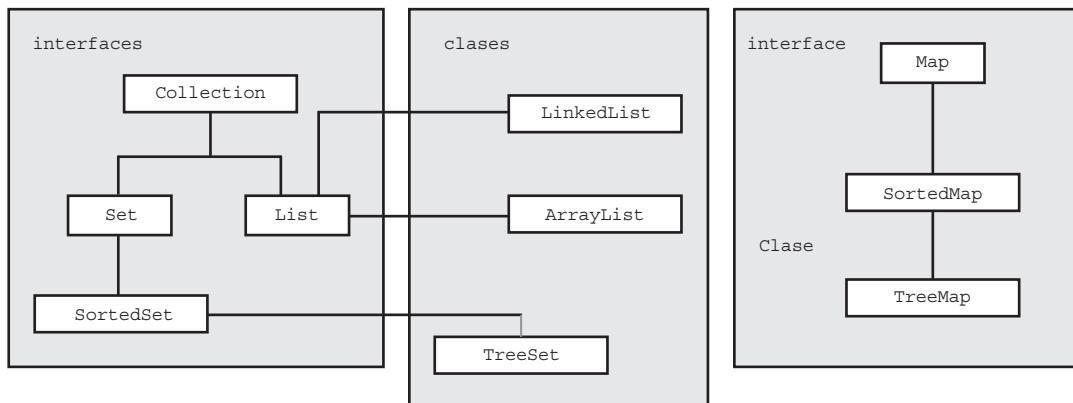


Figura 24.1 Algunos componentes de la jerarquía de colecciones.

Java 2 (versiones 1.2 y posteriores) incorpora nuevas clases colección, dentro del paquete `java.util` (API de las colecciones). Hay tres tipos generales de colecciones: *conjuntos*, *listas* y *mapas*; las interfaces `Set`, `List` y `Map` describen las características generales de estas. Además, la interfaz `Collection` especifica el comportamiento común de las colecciones.

El diseño de estas colecciones tuvo muy en cuenta los problemas de seguridad que pudieran surgir con la ejecución de múltiples tareas.¹ De tal forma que si varios hilos acceden a una de estas estructuras se garantiza que los accesos están sincronizados, es decir que hasta que un hilo no termina de procesar los elementos de una colección no hay otro hilo que procese a la misma colección. El problema de todo esto es que ralentiza la ejecución de la aplicación, además, aun cuando no sean necesario los comportamientos sincronizados, no hay forma de desactivar las comprobaciones de sincronización.

A continuación se escribe la declaración de métodos de la clase `Stack`, todos ellos con el comportamiento de sincronización.

```

public synchronized int size ( )
public synchronized boolean isEmpty ( )
public synchronized boolean contains (Object elemento)
public synchronized Object put (Object clave, Object valor)
public synchronized Object remove (Object clave)
  
```

Las nuevas clases *Colección*, añadidas en Java 2, se han diseñado sin el comportamiento de sincronización, al menos directamente. Por ejemplo, obsérvese la siguiente declaración de métodos de la colección `ArrayList`:

```

public int size ( )
public boolean isEmpty ( )
public boolean contains (Object elemento)
public Object get (int index)
public Object set (int index, Object elemento)
public boolean add (Object elemento)
public Object remove (int index)
  
```

Ahora bien, hay aplicaciones en las que interesa proteger a una colección de modificaciones simultáneas por más de un *hilo* (Thread). Entonces, para añadir la cualidad de sincronización a las colecciones Java 2 dispone de un mecanismo que dota de ese comportamiento a una colección. Esto está implementado mediante métodos `static` de la clase `Collections`, por ejemplo:

```

ArrayList vd = new ArrayList ( );
A continuación se le dota de sincronización,
vd = Collections.synchronizedList (vd) ;
  
```

¹ Tareas, hilos, se estudian en el capítulo 25.

Para un mapa el método `Collections.synchronizedMap ()`, para un conjunto `Collections.synchronizedSortedSet ()`; y así para otras colecciones.

24.2 Clases de utilidades: `Arrays` y `Collections`

La clase `Arrays` agrupa algoritmos útiles que se aplican, en general, a arreglos de los tipos primitivos. `Collections` también es una clase de utilidades, de métodos `static` que implementan algoritmos aplicados a todo tipo de colecciones.

Clase `Arrays`

Java 2 incorpora la clase `Arrays` para disponer de métodos que trabajen con arreglos de cualquier tipo. Estos métodos implementan algoritmos de búsqueda, ordenación y de asignación de un valor al *array* completo. Todos los métodos son `static` (métodos de clase). No se pueden crear objetos de la clase `Arrays`, el constructor que tiene es privado.

Ordenación de arrays

El método de ordenación, `sort ()`, está sobrecargado, de tal forma que se puede ordenar un *array* de cualquier tipo primitivo y, en general, de tipo `Object`. `sort ()` implementa el algoritmo de ordenación *quicksort* que asegura una eficiencia $n \log n$. Por ejemplo,

```
double [ ] w = {12.4, 5.6, 3.5, -2.0, 6.0, -4.5, 22.0};
// Llamada a sort ( ) para ordenar w
Arrays.sort(w);
```

A continuación se muestran algunos de los métodos de ordenación:

```
public static void sort (double [ ] w);
public static void sort (int [ ] w);
public static void sort (long [ ] w);
public static void sort (Object [ ] w);
public static void sort (Object [ ] w, Comparator cmp);
```

Se puede ordenar un *subarray*, para ello se especifica el índice *inicio* (inclusive) y *final* (exclusive):

```
// ordena los elementos w [inicio] .. w [final-1]
public static void sort(double [ ] w, int inicio, int final );
public static void sort(int [ ] w, int inicio, int final );
...

```

Norma

Para ordenar un arreglo de objetos los elementos deben implementar la *interfaz Comparable* ya que el criterio de ordenación está determinado por el método:

```
int compareTo(Object a);
```



Ejemplo 24.2

Declarar la clase `Racional` para representar un número racional (numerador, denominador) y llenar un *array* de objetos de esa clase. A continuación, ordenar de forma creciente el arreglo de números racionales.

La clase `Racional` tiene dos atributos de tipo `int`: `numerador` y `denominador`. Implementa la *interfaz Comparable* para poder realizar la ordenación con el método `Arrays.sort ()`. Por ello es necesario definir el método `int compareTo ()` de tal forma que devuelva `-1`, `0`, `1` si el número racional primer operando (`this`) es menor, igual o mayor, respectivamente que el número racional pasado como argumento.

```
import java.util.*;
class Racional implements Comparable
```

```

{
    private int numerador, denominador;
    public Racional ( ) throws Exception2
    {
        this(0,1);
    }
    public Racional(int n, int d) throws Exception
    {
        numerador = n;
        denominador = d;
        if (denominador == 0) throw new Exception("Denominador 0");
    }
    public String toString ( )
    {
        return numerador + "/" + denominador;
    }
    public int compareTo(Object x) // método de interfaz
    {
        Racional r;
        r = (Racional) x;
        if (valorReal ( ) < r.valorReal ( ))
            return -1;
        else if (valorReal ( ) > r.valorReal ( ))
            return 1;
        else
            return 0;
    }
    private double valorReal ( )
    {
        return (double)numerador/(double)denominador;
    }
    // ...
}
// clase principal, crea los objetos de manera aleatoria,
// se escriben en pantalla, a continuación se ordena, y por último
// se vuelve a escribir.
public class OrdenaRacional
{
    static int MR = 7;
    public static void main(String [ ] a)
    {
        Racional [ ] ar = new Racional[MR];
        try {
            // numerador y denominador se genera aleatoriamente
            for (int i = 0; i < MR; i++)
            {
                int n, d;
                n = (int)(Math.random( )* 21 +1);
                d = (int)(Math.random( )* 21 +1);
                ar [ i ] = new Racional(n, d);
            }
        }
        catch (Exception e) {};
        // listado de los objetos creados
        System.out.println(" Lista de numeros racionales: ");
    }
}

```

² En el capítulo 25 se realiza un estudio de excepciones en Java.

```

        escribe(ar);
        // ordenación del array
        Arrays.sort(ar);
        // listado de los objetos ordenados
        System.out.println(" Lista ordenada de numeros racionales: ");
        escribe(ar);
    }
    static void escribe(Racional [ ] r)
    {
        for (int i = 0; i < r.length; i++)
            System.out.print (r[i] + " ");
        System.out.println ( );
    }
}

```

Ejecución

```

Lista de numeros racionales:
11/5  2/17  13/12  1/19  13/19  3/11  7/19
Lista ordenada de numeros racionales:
1/19  2/17  3/11  7/19  13/19  13/12  11/5

```

Búsqueda de una clave

La operación de búsqueda se realiza sobre un arreglo ordenado. La clase `Arrays` dispone del método `static int binarySearch ()` para realizar la búsqueda de un elemento en un arreglo. El método devuelve la posición del elemento en el *array*, o bien, si no está, `-p`, en el que *p* es la posición de inserción del elemento en el arreglo. El algoritmo que utiliza el método es el de búsqueda binaria que asegura una eficiencia de $\log n$. El método está sobrecargado para cada tipo de dato primitivo, y para *arrays* de cualquier objeto (*Object*) que implemente la *interfaz Comparable*. Por ejemplo,

```

int [ ] w = {14, -5, 3, 2, 6, -4, 22, 4};
// llamada a sort ( ) para ordenar w
Arrays.sort (w);
// búsqueda de un elemento
int k;
k = Arrays.binarySearch (w, 4);
if (k >= 0)
    System.out.println ("Posición de " + 4 + " en la lista: " + k);

```

A continuación se escribe la declaración de alguno de estos métodos:

```

public static int binarySearch (double [ ] w, double clave);
public static int binarySearch (int [ ] w, int clave);
public static int binarySearch (Object [ ] w, Object clave);
public static int binarySearch (Object [ ] w, Object clave, Comparator c);

```

Asignación de un elemento

Otra utilidad de la clase `Arrays` es el método `fill ()` para asignar un elemento a todas las posiciones de un *array*, o bien a un rango del arreglo. El método está sobrecargado, habiendo una declaración para cada tipo de dato primitivo y para *Object*. A continuación se escribe la declaración de `fill ()` para algunos tipos:

```

public static void fill (byte [ ] w, byte val)
public static void fill (byte [ ] w, int inicio, int final, byte val);
public static void fill (char [ ] w, char val)
public static void fill (char [ ] w, int inicio, int final, char val);
public static void fill (int [ ] w, int val);

```

```

public static void fill (int [ ] w, int inicio, int final, int val);
public static void fill (Object [ ] w, Object val);
...

```

Definir un arreglo de enteros, inicializar la primera mitad a -1 y la segunda mitad a +1. Además, inicializar un arreglo de caracteres a la letra 'a' y un arreglo de cadenas a "Paloma".

Ejemplo 24.3

Se definen arreglos de los tipos pedidos y de tamaño constante. Para inicializar la primera mitad del array `a[]` hay que especificar el índice `inicio = 0` y `final = a.length/2`; la otra mitad tiene como `inicio` el anterior `final`.

```

static final int N = 10;
int [ ] iv = new int[N];
char [ ] cv = new char [N];
String [ ] sv = new String [N];
// llenado de los arrays
Arrays.fill (iv, 0, iv.length/2, -1);
Arrays.fill (iv, iv.length/2, iv.length 1);
Arrays.fill (cv, 'a');
Arrays.fill (sv, "Paloma");

```

Clase Collections

Esta clase se encuentra en el paquete `java.util`; está diseñada para trabajar con colecciones: `List`, `Map`, `Set`, en general sobre cualquier `Collection`. Agrupa métodos `static` que implementan algoritmos genéricos de ordenación, búsqueda, máximo y mínimo. Además, dispone de métodos para dotar de la cualidad de sincronización a una colección, y para convertir una colección a "solo lectura".

Ordenación y búsqueda

Los métodos de ordenación se aplican a una lista cuyos elementos implementan la interfaz `Comparable` y que se puedan comparar mutuamente. También, hay una sobrecarga de estos métodos para realizar la comparación con la interfaz `Comparator`.

```

public static void sort(list lista);
public static void sort(List lista, Comparator cmp);
public static int binarySearch(List lista, Object clave);
public static int binarySearch(List lista, Object cl, Comparator cmp);

```

Máximo y mínimo

Los métodos `max()` y `min()` devuelven el máximo y mínimo, respectivamente, de una colección. Para que se pueda realizar la operación, todos los elementos deben implementar la interfaz `Comparable` y ser mutuamente comparables. Es decir, si por ejemplo una colección guarda números complejos y cadenas, difícilmente se podrá obtener el máximo (además de ser absurdo). Los métodos son los siguientes:

```

public static Object max (Collection c);
public static Object min (Collection c);
// sobrecarga que obtiene el máximo o mínimo respecto a un comparador
public static Object max (Collection c, Comparator cmp);
public static Object min (Collection c, Comparator cmp);

```

Sincronización

Para añadir la cualidad de sincronización a las colecciones, `Collections` dispone de métodos que se aplican a cada tipo de colección:

```

public static Map synchronizedMap(Map mapa);
public static Set synchronizedSet(Set cn);

```

Conversión a solo lectura

Estos métodos convierten la colección al modo solo lectura, de tal forma que la operación de añadir (add) o eliminar (remove) un elemento levanta la excepción `UnsupportedOperationException`. Algunos de los métodos son los siguientes:

```
public static List unmodifiableList(List lista);
public static Set unmodifiableSet (Set conjunto);
public static Collection unmodifiable Collection (Collection c);
```

A continuación se escribe un ejemplo, en el que se crea un `Set` y después se convierte a modo *solo lectura*.

```
Set cn = new HashSet( );
cn.add("Marta");
// crea una visión de cn no modificable, de solo lectura
cn = Collections.unmodifiableSet (cn);
```

Utilidades

La clase `Collections` dispone de métodos útiles para ciertos procesos algorítmicos. El método `nCopies ()` crea una lista con n copias de un elemento; el método `copy ()` crea una lista copia de otra; `fill ()` llena todos los elementos de una lista con un objeto.

```
public static List nCopies (int n, Object ob);
public static void copy (List destino, List fuente);
public static void fill (List lista, Object ob);
```

El método `reverse ()` invierte la posición de los elementos de una lista. Si la lista está en orden creciente, la llamada `Collections.reverse (lista)` deja a la lista en orden decreciente.

```
Public static void reverse (List lista);
```

También incluye el método `shuffle ()` para reordenar aleatoriamente los elementos de una lista.

```
Public static void shuffle (List lista);
```



Ejemplo 24.4

Se realizan diversas operaciones utilizando métodos de la clase `Collections`: crear una lista a partir de la copia n veces de una cadena, ordenar una lista de objetos `Integer`, buscar el máximo, etcétera.

Se escribe el método `main ()` con las operaciones `nCopies ()`, `sort ()`, `max ()`, `min ()` y `reverse ()`. La declaración de la lista de objetos especifica que el tipo de los elementos es `Integer`. De esa forma el compilador realiza comprobaciones de tipo. También se puede realizar de forma genérica: `List lista = new ArrayList ();` de tal forma que se podría añadir cualquier tipo de objeto.

```
public static void main(String [ ] args)
{
    int n = 11;
    List listal;
    // Crea una lista formada por n copias de "Marga"
    listal = Collections.nCopies(n, "Marga");
    System.out.println (listal);
    // Crea una lista de objetos Integer, se ordena y se invierte
    List<Integer> lista2 = new ArrayList<Integer>();
    for (int i = 1; i <= n ; i++)
        lista2.add (new Integer ((int)(Math.random( )*100 +1)));
    System.out.println (lista2);
    System.out.println ("Máximo: " + Collections.max(lista2)
                       + " \t Mínimo: " + Collections.min(lista2));
    Collections.sort (lista2);
```

```

        System.out.println (lista2);
        Collections.reverse (lista2);
        System.out.println (lista2);
    }

```

Ejecución

```

[Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga]
[26, 1, 97, 7, 46, 76, 98, 69, 53, 50, 76]
Máximo: 98    Mínimo: 1
[1, 7, 26, 46, 50, 53, 69, 76, 76, 97, 98]
[98, 97, 76, 76, 69, 53, 50, 46, 26, 7, 1]

```

24.3 Comparación de objetos: Comparable y Comparator

Numerosas operaciones con colecciones exigen que sus elementos sean comparables, es decir, que se pueda determinar que un elemento es menor, igual o mayor que otro. Esta propiedad se establece a nivel de clase, implementando la interfaz Comparable, o bien la interfaz Comparator.

Comparable

La interfaz Comparable se utiliza para establecer un orden natural entre los objetos de una misma clase. La declaración de la interfaz es la siguiente (paquete `java.lang`):

```

public interface Comparable
{
    public int compareTo(Object ob);
}

```

Si `compareTo ()` devuelve un negativo significa que el objeto que llama al método es menor que el pasado en el argumento; si devuelve 0 significa que son iguales, y si devuelve un positivo el objeto que llama al método es mayor que el pasado en el argumento. La clase `Racional` del ejemplo 24.2 implementa la interfaz Comparable, el programador de la clase ha fijado la forma de comparar números racionales.

Nota

Todas las clases que representan a tipos primitivos (*wrapper*): `Integer`, `Double`, ..., y la clase `String` implementan la interfaz Comparable. La implementación de `compareTo ()` en `String` distingue mayúsculas de minúsculas.

Comparator

Hay métodos de ordenación y búsqueda de objetos que utilizan la interfaz Comparator para determinar el orden natural entre dos elementos. Esta interfaz se encuentra en el paquete `java.util`. Su declaración es la siguiente:

```

public interface Comparator
{
    public int compare (Object ob1, Object ob2);
    public boolean equals (Object ob);
}

```

El método `compare ()` relaciona dos objetos, no necesariamente del mismo tipo. Devolverá un negativo si el objeto `ob1` es menor que el segundo objeto, *cero* si son iguales, y positivo si `ob1` es mayor que `ob2`. El comportamiento de `compare ()` tiene que ser compatible con el resultado del método `equals ()`, es decir, si este devuelve `true`, `compare ()` tendrá que devolver 0. Por ejemplo:

```

class Cuadro
{
    int largo;
    int ancho;
    ...
}
class Compara implements Comparator
{
    public int compare (Object ob1, Object ob2)
    {
        Cuadro c1 = (Cuadro) ob1;
        Cuadro c2 = (Cuadro) ob2;
        if (c1.largo == c2.largo && c1.ancho == c2.ancho)
            return 0;
        else
            // c1 es menor que c2 si, en superficie, c1 es menor
            return (c1.largo * c1.ancho) - (c2.largo * c2.ancho);
    }
}

```

Se puede ordenar un arreglo, o una lista, de *cuadros* pasando al método `sort ()` el objeto *comparador*:

```

Cuadro [ ] vcdos = new Cuadro[N];
Arrays.sort (vcdos, new Compara ( ));

```

Norma

Los métodos que utilizan un argumento de tipo `Comparator` se llaman pasando una referencia a la clase que implementa la interfaz `Comparator`. Por ejemplo:

```

class Alba implements Comparator {...}
Collections.sort (lista, new Alba( ));

```

24.4 Vector y Stack

Tanto `Vector` como `Stack` son colecciones históricas para guardar objetos de cualquier tipo. Se puede colocar cualquier número de objetos ya que se redimensionan automáticamente.

Vector

El comportamiento de un vector se asemeja al de un arreglo, con la bondad de que no es necesario controlar su tamaño; automáticamente, si fuera necesario, aumenta su capacidad. A partir de Java 2 la clase `Vector` implementa la interfaz `List` para que forme parte de las colecciones de Java. También, la plataforma Java 5 permite establecer el tipo concreto de elemento que puede guardar una colección, y en particular un vector, para así realizar comprobaciones de tipo durante el proceso de compilación. Por ejemplo, un vector de cadenas (`String`):

```

Vector<String> vc = new Vector<String> ( );
vc.addElement ("Lontananza");
vc.addElement (new Integer(12));      // error de compilación

```

Sin embargo, si la declaración es la siguiente:

```
Vector vc = new Vector ( );
```

Se puede añadir cualquier tipo de elemento, no hay comprobación de tipo:

```

vc.addElement ("Lontananza");
vc.addElement (new Integer(12));      // correcto

```

Stack

La clase `Stack` hereda el comportamiento de un vector y además define las operaciones del *tipo abstracto Pila* (*último en entrar primero en salir*). Todas las operaciones se realizan por un único punto, el final (*cabeza* o *top*) de la pila. La declaración de `Stack` es la siguiente:

```
public class Stack extends Vector
{
    ...
}
```

Los métodos definidos por la clase:

public Stack ();	Constructor, crea una pila vacía.
public Object push (Object n);	Añade el elemento n, devuelve n.
public Object pop ();	Devuelve elemento <i>cabeza</i> y lo quita de la pila.
public Object peek ();	Devuelve elemento <i>cabeza</i> sin quitarlo de la pila.
public boolean empty ();	Devuelve true si la pila está vacía.

Los elementos que almacena una colección `stack` son de tipo genérico (tipo `Object`), esto hace necesario realizar conversión de tipo cuando se extraen. La plataforma Java 5 permite parametrizar el tipo de los elementos que guarda el `stack`, de tal forma que el compilador verifica el tipo.

Mediante una pila se analiza si una palabra o frase es palíndromo.

Se lee la palabra del teclado y a la vez cada carácter se guarda en una pila. Los elementos de la pila han de ser de tipo `Character`, por consiguiente su declaración es: `Stack<Character> pila`.

Ejemplo 24.5

```
import java.util.*;
import java.io.*;
public class Palindromo
{
    public static void main(String [ ] args)
    {
        Scanner entrada = new Scanner (System.in)
        Stack<Character> pila;
        String palabra;
        boolean pal;
        pila = new Stack<Character> ( );
        try {
            System.out.println("Palabra o frase: ");
            while ((palabra = entrada.nextLine ( )) != null)
            {
                for (int i = 0; i < palabra.length ( ); i++)
                {
                    pila.push (new Character(palabra.charAt (i)));
                }
                pal = true;
                int i = 0;
                while (pal && !pila.empty ( ))
                {
                    Character q;
                    q = pila.pop ( );
                    pal = q.charValue ( ) ==  palabra.charAt (i++);
                }
                if (pal && pila.empty ( ))
                    System.out.println (palabra + " es un palíndromo");
                else

```

```
        System.out.println (palabra + " no es un palíndromo");
    }
}
catch (Exception e) { ; }
}
}
```

24.5 Iteradores de una colección

Un *iterador* permite acceder a cada elemento de una colección sin necesidad de tener que conocer la estructura de esta. Históricamente, Java 1.0 incorpora el iterador `Enumeration`, posteriormente Java 2 desarrolla dos nuevos iteradores: `Iterator` y `ListIterator`.

Enumeration

La interfaz `Enumeration` declara métodos que recorren una colección. Mediante este tipo de iterador se puede acceder a cada elemento de una colección, pero no permite modificar la colección, es de solo lectura. `Enumeration` forma parte del paquete `java.util`. Su declaración es la siguiente:

```
public interface Enumeration
{
    boolean.hasMoreElements()
    Object.nextElement();
}
```

`nextElement()` devuelve el siguiente elemento. Levanta la excepción de tipo `NoSuchElementException` si no hay más elementos, es decir si ya se ha recorrido toda la colección. La primera llamada devuelve el primer elemento.

hasMoreElements () devuelve true si no se ha accedido a todos los elementos de la colección. Normalmente se diseña un bucle, controlado por este método, para acceder a cada elemento de la colección.

Las colecciones históricas: Vector, Stack, Dictionary, HashTable disponen del método `elements ()` que devuelve un Enumeration, a partir del cual se puede recorrer la colección. La declaración es la siguiente:

Enumeration elements ()

El esquema que se sigue para acceder a cada elemento de una colección consta de los siguientes pasos:

1. Declarar una variable Enumeration.
`Enumeration enumera;`
 2. Llamar al método elements () de la colección.
`enumera = colección.elements ()`
 3. Diseñar el bucle que obtiene y procesa cada elemento.

```
while (enumera.hasMoreElements ( ) )
{
    elemento = (TipoElemento) enumera.nextElement( );
    <proceso de elemento>
}
```



Ejemplo 24.6

24.6 Se crea una pila de diferentes objetos, posteriormente, se recorre con un iterador `Enumeration` y con métodos de `Stack`.

La pila se llena de objetos `String`, `Integer` y `Double`, sin un orden establecido. Para recorrer la pila se crea un enumerador y un bucle hasta que no queden más elementos sin visitar. También se recorre aplicando la operación `pop ()` y controlando que no esté vacía.

```

import java.util.*;
import java.io.*;

public class EnumeradorPila
{
    public static void main(String [ ] args)
    {
        final int N = 8;
        Stack pila = new Stack ( );
        String [ ] palabra =
            {"Urbion", "Magina", "Abantos", "Peralte", "Citores"};
        for (int i = 0; i < N; i++)
        {
            int n;
            n = (int) (Math.random ( )*N*2);
            if (n < palabra.length)
                pila.push (palabra[n]);
            else if (n < N+2)
                pila.push (new Double(Math.pow(n, 2)));
            else
                pila.push (new Integer(n * 3));
        }
        // crea un enumerador de la pila
        Enumeration enumera = pila.elements ( );
        // bucle para recorrer la pila
        System.out.println ("Elementos de la pila " +
            "en el orden establecido por el enumerador:");
        while (enumera.hasMoreElements ( ))
        {
            Object q;
            q = enumera.nextElement ( );
            System.out.print (q + " ");
        }
        // bucle para recorrer la pila
        System.out.println ("\nElementos de la pila en orden LIFO:");
        while (!pila.empty ( ))
        {
            Object q;
            q = pila.pop ( );
            System.out.print (q + " ");
        }
    }
}

```

Ejecución

```

Elementos de la pila en el orden establecido por el enumerador:
81.0 25.0 Urbion 49.0 36.0 36.0 64.0 64.0
Elementos de la pila en orden LIFO:
64.0 64.0 36.0 36.0 49.0 Urbion 25.0 81.0

```

Iterator

Java 2 desarrolla nuevas colecciones y el iterador común `Iterator`. Todo objeto colección se puede recorrer con este iterador. Todas las colecciones tienen el método `iterator ()` que devuelve un objeto `Iterator`.

```

Iterator iter;
iter = Coleccion.iterator ( );

```

La interfaz `Iterator` permite no solo acceder a los elementos, sino también eliminarlos. Pertenece al paquete `java.util`; la declaración es la siguiente:

```
public interface Iterator
{
    boolean hasNext ( );
    Object next ( );
    void remove ( );
}
```

`hasNext ()` devuelve `true` si quedan elementos no visitados, es el equivalente a `hasMoreElements ()`.

`next ()` la primera llamada devuelve el primer elemento, según el orden establecido por el iterador.

`remove ()` elimina de la colección el elemento obtenido por la última llamada a `next ()`. Solo se puede llamar una vez después de `next ()`, en caso contrario, o bien si no ha habido una llamada previa a `next ()`, levanta la excepción `IllegalStateException`. Normalmente las colecciones implementan este método en un bloque sincronizado.

Norma

Se recomienda utilizar `Iterator` en lugar de `Enumeration` usada en colecciones históricas. Los métodos de acceso son más sencillos de recordar y además declara el método `remove ()`.



Ejemplo 24.7

Dada una lista de puntos del plano se quiere eliminar de la lista aquellos cuya coordenada `x` esté fuera del rango `[2, 12]`.

Se da por declarada la clase `Punto` con el método `int getX ()`, y que la lista de puntos ya está creada. La lista es del tipo `ArrayList`. El fragmento de código escrito declara la lista y el iterador; realiza un bucle controlado por `hasNext ()` para acceder a cada elemento y si la coordenada `x` no está en el rango `[2, 12]` lo elimina llamando al método `remove ()`.

```
class Punto {...}
List lista = new ArrayList ( );
Iterator iter;
// se llena la lista de objetos Punto
iter = lista.iterator ( );
while (iter.hasNext ( ) )
{
    Punto q;
    q = (Punto)iter.next ( );
    if (q.getX ( ) < 2 || q.getX ( ) > 12)
    {
        System.out.println ("Punto: " + q + " se elimina");
        iter.remove ( );
    }
}
```

La colección guarda elementos de cualquier tipo (`Object`), esto exige realizar una conversión al tipo concreto (`Punto`) con el que se trabaja. Con Java 1.5 se puede parametrizar el tipo de los elementos, de tal forma que no es necesario realizar la conversión y, además, el compilador verifica el tipo de los elementos añadidos. A continuación se escribe el código con esta característica.

```
List<Punto> lista = new ArrayList<Punto> ( );
Iterator<Punto> iter;
iter = lista.iterator ( );
while (iter.hasNext ( ) )
{
    Punto q;
    q = iter.next ( ); // no es necesario un cast
    ....
```

24.6 Listas

Una lista es una agrupación lineal de elementos, que pueden duplicarse. A una lista se añaden elementos por la *cabeza*, por el *final*, en general por cualquier punto. También, se permite eliminar elementos de uno en uno, o bien todos aquellos que estén en una colección. Hay dos tipos de listas, las secuenciales y las enlazadas. El concepto general de lista está representado por la interfaz `List`; esta interfaz es la raíz de la jerarquía y por conversión automática toda colección de tipo lista se puede tratar con una variable de tipo `List`. Por ejemplo:

```
List lista;
lista = new ArrayList ();
lista = new LinkedList ();
```

La jerarquía de listas se muestra en la figura 24.2. Java 2 ha modificado las clases históricas `Vector` y `Stack` para ubicarlas en esta jerarquía, mantienen la funcionalidad histórica y, además, la funcionalidad heredada de la clase `AbstractList`.

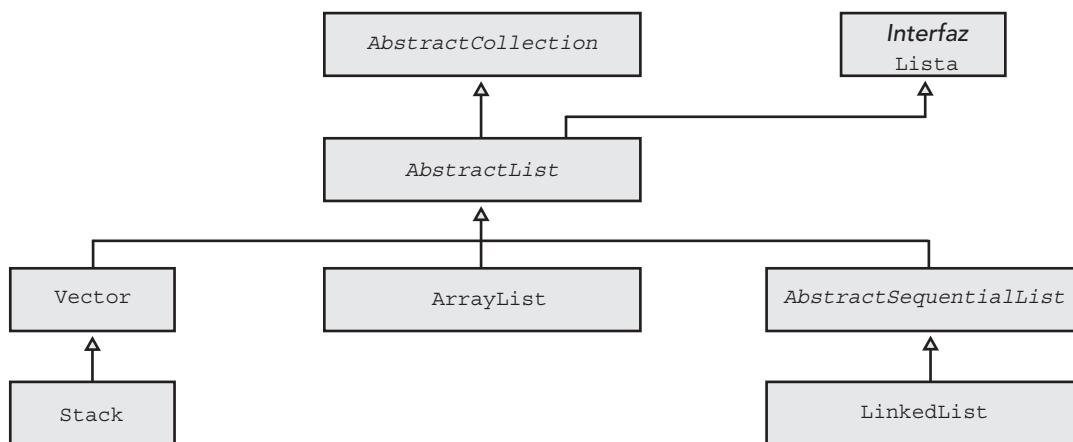


Figura 24.2 Jerarquía de colecciones `List`.

`AbstractList` es una clase abstracta que se utiliza como esqueleto para implementar clases concretas con la característica de acceso aleatorio a los elementos, como un arreglo. Esto quiere decir que se puede acceder a un elemento por un índice. Una clase que derive de esta debe implementar los métodos `get(int indice)`, `size()`, y además, si va a realizar cambios, `set()`, `add()` y `remove()`; los métodos `iterator()`, `listIterator()`, `indexOf()` están definidos por esta clase. La clase `ArrayList` es una clase concreta que deriva de `AbstractList`; se utiliza para almacenar cualquier tipo de elementos, incluso está recomendado su uso en lugar de `Vector`.

ArrayList

Esta clase agrupa elementos como un arreglo. Es equivalente a `Vector`, pero con las mejoras introducidas por Java 2. Se puede acceder a cualquier elemento, insertar o borrar a partir del índice en cualquier posición, aunque un tanto ineficiente si se realiza en posiciones intermedias.

La clase `ArrayList` tiene tres constructores:

```
public ArrayList ();
public ArrayList (int capacidad);
public ArrayList (Collection c);
```

Por ejemplo, se crea una colección con los elementos de un vector:

```
Vector v = new Vector ();
ArrayList al = new ArrayList (v);
```

La clase `ArrayList` implementa los métodos de la interfaz `List`, también el método `clone()` de la interfaz `Cloneable` para poder crear una copia independiente de la colección.

Ejemplo 24.8

Se realizan las operaciones de añadir, eliminar, buscar y reemplazar con una colección de tipo `ArrayList`.

La colección va a estar formada por cadenas (`String`) leídas del teclado. Una vez formada la colección es eliminada una cadena concreta y se reemplaza el elemento que ocupa la posición central. Para realizar una búsqueda se utiliza el método `indexOf ()` que devuelve la posición que ocupa, o bien `-1`; a partir de esta posición se crea un iterador llamando al método `iterator ()` con el fin de recorrer y, a la vez, escribir los elementos.

```
import java.util.*;
import java.io.*;
public class ListaArray
{
    public static void main(String [ ] args)
    {
        Scanner entrada = new Scanner (System.in));
        List av = new ArrayList ( );
        String cd;
        System.out.println ("Datos de entrada (adios para acabar)");
        try {
            do {
                cd = entrada.nextLine ( );
                if (! cd.equalsIgnoreCase ("adios"))
                    av.add(cd);
                else break;
            } while (true);
            System.out.println ("Lista completa:" + av);
            // elimina una palabra
            System.out.println ("Palabra a eliminar: ");
            cd = entrada.nextLine( )
            if (av.remove (cd) )
                System.out.println ("Palabra borrada, lista actual: + av");
            else
                System.out.println ("No esta en la lista la palabra");
            // reemplaza elemento que está en el centro
            av.set(av.size ( )/2, "NuevaCadena");
            System.out.println ("Lista completa:" + av);
            // búsqueda de una palabra
            System.out.println ("Búsqueda de una palabra: ");
            cd = entrada.nextLine( );
            int k = av.indexOf (cd);
            // crea iterador y recorre la lista hacia adelante
            if (k >= 0)
            {
                System.out.println ("Recorre la lista ");
                Iterator ls = av.iterator ( );
                while (ls.hasNext ( ) )
                {
                    System.out.print ((String)ls.next ( ) + " ");
                }
            }
        }
        catch(Exception e) {;}
    }
}
```

24.7 Colecciones parametrizadas

Las colecciones históricas y las desarrolladas en Java 2 guardan, internamente, los elementos en *arrays* de tipo *Object*. Con esto se consigue la máxima generalización ya que *Object* es la clase base de cualquier objeto Java. Esto permite que una colección guarde objetos de cualquier tipo, por ejemplo una lista puede contener cadenas (*String*), números racionales ... :

```
LinkedList lis = new LinkedList();
Racional r = new Racional(3,7);
lis.add(r);
lis.add(newString("Mi globo"));
```

Al recuperar elementos de la lista es necesario discernir el tipo concreto de elemento, por ejemplo:

```
Object q;
Racional t;
q = lis.getFirst();
if (q instanceof Racional)
    t = (Racional) q;
...
```

En muchas aplicaciones de colecciones los elementos son del mismo tipo, un conjunto de enteros, una lista de palabras ... a pesar de lo cual la recuperación de elementos siempre necesita una conversión de *Object* a ese tipo. Por esta razón, y otras como que el compilador pueda realizar comprobaciones de tipo, a partir de Java 5 se amplía la declaración de todas las clases e interfaces relacionadas con colecciones para dotarlas de la posibilidad de parametrizar el tipo que va a tener los elementos de una colección. Por ejemplo, un vector que vaya a guardar elementos de tipo *Complex* se declarará e instanciará:

```
Vector<Complex> zz;
zz = new Vector<Complex>(19);
```

La funcionalidad de *vector* no cambia, aunque el compilador sí comprobará tipos de datos al realizar operaciones, como por ejemplo añadir elementos:

```
Complex nz = new Complex(1,-1);
zz.addElement(new Integer(8)); // error de compilación, tipo mismatch
zz.addElement(nz); // correcto
```

La recuperación de elementos ya no necesita conversión de *Object* a *Complex*:

```
nz = zz.elementAt(0);
```

Declaración de un tipo parametrizado

Al nombre de la colección le sigue el tipo de los elementos entre paréntesis angulares (*<tipo>*):

```
Coleccion<tipo> v;
```

Si se parametrizan dos tipos, como ocurre con los mapas, se separan con coma:

```
Coleccion<tipoClave,tipoValor> cc;
```

Por ejemplo:

```
Stack<Double> pila;
```

Realmente con el tipo parametrizado es como si se hubiera declarado otra clase, en consecuencia las instancias de colecciones parametrizadas se crean con esa clase, es decir *new Coleccion<tipo>* crea la instancia. Por ejemplo:

```
pila = new Stack<Double>();
```

Sintaxis

A continuación del nombre de la clase se especifican los tipos parametrizados entre paréntesis angulares:

```
Coleccion<tipo1...> var;
```

Para crear instancias, *new Coleccion<tipo1...>()* ;



Resumen

- Una colección agrupa objetos relacionados que forman una única entidad, por ejemplo un arreglo de objetos, un conjunto de números complejos, etcétera.
- Las colecciones incluyen clases contenedoras, iteradores para acceder a los objetos en el interior de los contenedores y algoritmos (métodos de clases) para manipular los objetos.
- Las clases tipo *Colección* guardan objetos de cualquier tipo, de hecho el elemento base es *Object* y por consiguiente se podrá añadir a la colección un objeto de cualquier tipo.
- Las colecciones históricas (Java 1) más importantes son *Vector*, *Stack*, *Dictionary*, *HashTable* y la interfaz *Enumeration* para recorrer los elementos de una colección.
- Java 2 establece tres tipos generales de colecciones: *conjuntos*, *listas* y *mapas*; las interfaces *Set*, *List* y *Map* describen las características generales de estos. Además, la interfaz *Collection* especifica el comportamiento común de las colecciones.
- Las clases *Arrays* y *Collections* agrupan algoritmos útiles que se aplican, respectivamente, a arreglos de los tipos primitivos y a todo tipo de colecciones. Por ejemplo:

```
Arrays.sort(w);
Collections.reverse(lista2);
```

Los elementos que se comparan deben implementar la interfaz *Comparable*, que declara el método *compareTo()*, o bien la interfaz *Comparator* que declara el método *compare()*.

- La clase *Vector* se comporta como un arreglo de elementos genéricos que se redimensiona automáticamente. *Stack* representa el tipo abstracto *Pila*, deriva de *Vector*. Ambas son colecciones históricas que se han redefinido para encuadrarlas en el conjunto de colecciones de Java 2.

- La llamada al método *elements()* de una clase colección devuelve un *Enumeration* para recorrer o acceder a sus elementos sin modificarlos. Los métodos de *Enumeration* son los siguientes:

```
boolean hasMoreElements();
Object nextElement();
```

- *Iterator* es otro iterador, definido a partir de Java 2, que permite acceder a los elementos y también eliminarlos. Los métodos que declara son los siguientes:

```
boolean hasNext();
Object next();
void remove();
```

- Las listas son de dos tipos: secuenciales (*ArrayList*) y las enlazadas (*LinkedList*). El concepto general de lista está representado por la interfaz *List*. El iterador *ListIterator* está diseñado para recorrer cualquier lista, en particular una lista enlazada. Con este iterador se puede avanzar o retroceder por los elementos.
- Java 5 amplía la declaración de todas las clases e interfaces relacionadas con colecciones para dotarlas de la posibilidad de parametrizar el tipo de los elementos de una colección. Por ejemplo, un vector que vaya a guardar elementos de tipo *complex* se declarará e instanciará:

```
Vector<Complex> zz;
zz = new Vector<Complex>(19);
```

Al nombre de la colección le sigue el tipo de los elementos entre paréntesis angulares (*<tipo>*), los tipos se separan por coma (,):

```
Coleccion<tipo> v;
Coleccion<tip1,tip2> w;
```



Ejercicios

- 24.1 Declarar la clase *Pagina* y un arreglo de tipo *Pagina* de *n* elementos, donde *n* es un dato de entrada. A continuación asignar a cada elemento del arreglo un objeto *Pagina*.
- 24.2 Modificar la declaración de la clase *Pagina* de tal forma que implemente la interfaz *Comparable*. Crear un arreglo de tipo *Pagina* y llamar al método *sort()* para que esté ordenado.
- 24.3 Declarar un arreglo de tipo *String* de 40 elementos. Llamar al método *fill()* para inicializar la primera mitad a “Domingo” y la segunda a “Lunes”.
- 24.4 ¿Qué diferencias existen entre un iterador de tipo *Enumeration* y otro de tipo *Iterator*?
- 24.5 Señalar las ventajas de utilizar tipos parametrizados en las colecciones. ¿Tiene algún inconveniente?



Problemas

- 24.1** Se desea almacenar en un *vector* los atletas participantes en un cross popular. Los datos de cada atleta: *Nombre, Apellido, Edad, Sexo, Fecha de nacimiento y Categoría (Júnior, Promesa, Sénior, Veterano)*. La clave es el *Apellido* del atleta. Realizar las declaraciones necesarias y una aplicación que cree el vector y posteriormente la muestre por pantalla.
- 24.2** Crear una lista ordenada (de tipo *ArrayList*) cuyos elementos sean autor y la lista de libros de dicho autor. La aplicación dará entrada a los elementos de la lista y realizará operaciones de búsqueda de autor, de libro, listados...
- 24.3** La clase *Racional* representa un número racional, tiene dos atributos: numerador y denominador. Declara la clase *Racional* y la clase *Compara* para definir el método *compara ()* de la interfaz *Comparador* aplicado a dos números racionales (realmente a dos objetos *Racional*). Escribir una aplicación que cree dos colecciones (un *vector* y un *stack*) ordenados de tipo *Racional*, utilizando como comparador *Compara*. Realice operaciones diversas: máximo de los elementos del vector, suma de los elementos del stack, listado de los elementos del vector o de la pila (*stack*) comprendidos en un rango de elementos.



Multitarea y excepciones

Contenido

- 25.1 Manejo de excepciones en Java
- 25.2 Mecanismo del manejo de excepciones
- 25.3 Clases de excepciones definidas en Java
- 25.4 Nuevas clases de excepciones
- 25.5 Especificación de excepciones
- 25.6 Multitarea
- 25.7 Creación de hilos

- 25.8 Estados de un hilo, ciclo de vida de un hilo
- 25.9 Prioridad entre hilos
- 25.10 Hilos *daemon*
- 25.11 Sincronización
 - › Resumen
 - › Ejercicios

Introducción

Uno de los problemas más importantes en el desarrollo de software es la gestión de condiciones de error. No importa lo bueno que sea el software y la calidad del mismo, siempre aparecerán errores por múltiples razones (errores de programación, errores imprevistos de los sistemas operativos, agotamiento de recursos, etc.).

Las *excepciones* son, normalmente, condiciones de errores imprevistos. Estas condiciones suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos típicos son: agotamiento de memoria, errores de rango en intervalos, división por cero, ficheros no existentes, etc. *El manejo de excepciones* es el mecanismo previsto por Java para el tratamiento de excepciones. Normalmente el sistema aborta la ejecución del programa cuando se produce una excepción; Java permite al programador intentar la recuperación de estas condiciones y decidir si continuar o no la ejecución del programa.

Los sistemas operativos actuales permiten la ejecución simultánea de varios programas. Esto es lo que se conoce como *multitarea*. Java puede ejecutar simultáneamente varios procesos, varios hilos de ejecución. La clase que admite los hilos de ejecución es *Thread*.

Conceptos clave

- › Bloque *try*
- › Captura de excepciones
- › *catch*
- › Excepción
- › *finally*
- › Hilo
- › Lanzamiento de una excepción
- › Levantar una excepción
- › Manejo de excepciones
- › Multitarea
- › Sincronización
- › *Thread*
- › *Throw*

25.1 Manejo de excepciones en Java

Una excepción se “levanta” (*raise*) en caso de un error e indica una condición anormal que no se debe encontrar durante la ejecución normal de código. Una excepción indica una necesidad urgente de tomar una acción reparadora (de remedio).

La palabra *excepción* indica aquí una *excepción software*. No se debe confundir con una *excepción hardware*. Una excepción software se produce por una parte de código escrita por un programador. No tiene nada que ver con las excepciones hardware. Una excepción software se inicia en alguna sentencia del código que encuentra una condición anormal.

Una **excepción** es un error de programa que ocurre durante la ejecución. Si ocurre una excepción y está activo un segmento de código denominado *manejador de excepción* para esa excepción, entonces el flujo de control se transfiere al manejador. Si en lugar de ello ocurre una excepción y no existe un manejador para la excepción, ésta se propaga al método invocante; si en éste tampoco se capta la excepción se propaga al que a su vez le llamó; si llega al método por el que empieza la ejecución (`main()`) y tampoco es captada, la ejecución termina.

Una excepción se puede levantar cuando “el contrato” entre el *llamador* y el *llamado* se violan. Por ejemplo, si se intenta acceder a un elemento que está fuera del rango válido de un arreglo se produce una violación del contrato entre el método que controla los índices (operador `[]` en Java) y el llamador que utiliza el arreglo. La función de índices garantiza que devuelve el elemento correspondiente si el índice que se le ha pasado es válido. Pero si el índice no es válido, la función de índice debe indicar la condición de error. Siempre que se produzca tal violación del contrato se debe levantar (*alzar*) una excepción.

Una vez que se levanta una excepción, esta no desaparece aunque el programador lo ignore (una excepción no se puede ignorar ni suprimir). Una condición de excepción se debe *reconocer* y manejar. Una excepción no manejada se propagará dinámicamente hasta alcanzar el nivel más alto de la función (`main` en Java). Si también falla el nivel de función más alto, la aplicación termina sin opción posible.

Precaución

El manejo de errores usando excepciones no evita errores, solo permite la detección y posible recuperación de los mismos.

En general, el mecanismo de excepciones en Java permite:

1. Detección de errores, enérgica y posible recuperación.
2. Limpieza y salida elegante en caso de errores no manejados; y
3. Propagación sistemática de errores en una cadena de llamadas dinámicas.

25.2 Mecanismo del manejo de excepciones

El modelo de un mecanismo de excepciones consta, fundamentalmente, de cinco nuevas palabras reservadas `try`, `throw`, `throws`, `catch` y `finally`.

- `try`, un bloque para detectar excepciones.
- `catch`, un manejador para capturar excepciones de los bloques `try`.
- `throw`, una expresión para levantar (*raise*) excepciones.
- `throws` indica las excepciones que puede elevar un método.
- `finally`, bloque, opcional, situado después de los `catch` de un `try`.

Los pasos del modelo son:

1. Primero, el programador establece un conjunto de operaciones para anticipar errores. Esto se realiza en un bloque `try`.
2. Cuando una rutina encuentra un error, se “lanza” (*throw*) una excepción. El lanzamiento (*throwing*) de una excepción es el acto de *levantar una excepción*.
3. Por último, alguien interesado en una condición de error (para limpieza y/o recuperación) anticipará el error y “capturará” (*catch*) la excepción que se ha lanzado.

El mecanismo de excepciones se completa con:

1. Un bloque `finally` que, si se especifica, siempre se ejecuta al final de un `try`.
2. Especificaciones de excepciones que dictamina cuáles excepciones, si existen, puede lanzar un método.

El modelo de manejo de excepciones

La filosofía que subyace en el modelo de *manejo de excepciones* es simple. El código que trata con un problema no es el mismo código que lo detecta. Cuando una excepción se encuentra en un programa, la parte del programa que detecta la excepción puede comunicar que la expresión ha ocurrido levantando, o lanzando (*throwing*), una excepción.

De hecho, cuando el código de usuario llama a un método incorrectamente o utiliza un objeto de una clase de manera inadecuada, la biblioteca de la clase crea un objeto excepción que contiene información sobre lo que era incorrecto. La biblioteca levanta (*raise*) una excepción, una acción que hace el objeto excepción disponible al código de usuario a través de un *manejador de excepciones*. El código de usuario que maneja la excepción puede decidir qué hacer con el objeto excepción.

El cortafuegos que actúa de puente entre una biblioteca de clases y una aplicación debe hacer varias cosas para gestionar debidamente el flujo de excepciones, reservando y liberando memoria de modo dinámico.

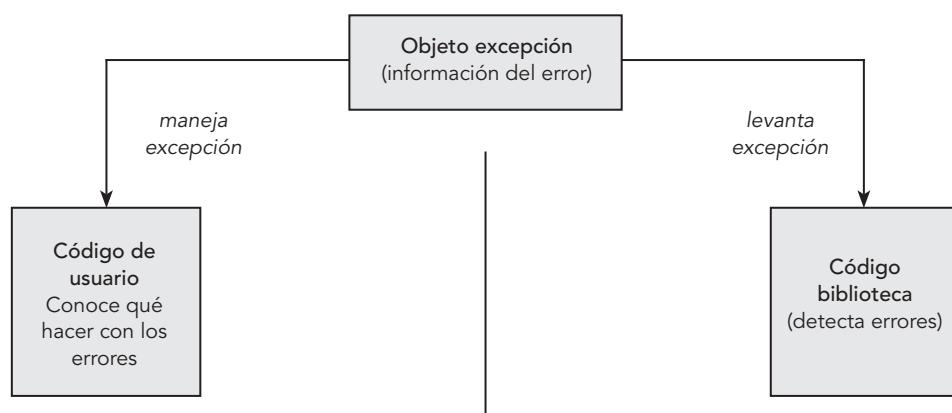


Figura 25.1 Manejo de excepciones construye un “cortafuego”.

Una de las razones más significativas para utilizar excepciones es que las aplicaciones no pueden ignorarlas. Cuando el mecanismo de excepciones levanta una excepción, alguien debe tratarla. En caso contrario, la excepción es “no capturada” (*uncaught*) y el programa termina por omisión. Este poderoso concepto es el corazón del manejo de excepciones y fuerza a las aplicaciones a manejar excepciones en lugar de ignorarlas.

El mecanismo de excepciones de Java sigue un *modelo de terminación*. Esto implica que nunca vuelve al punto en que se levanta una excepción. Las excepciones no son como manejadores de interrupciones que bifurcan a una rutina de servicio antes de volver al *spot* de interrupción. Esta técnica (llamada *resumption*) tiene un alto tiempo suplementario, es propensa a bucles infinitos y es más complicado de implementar que el *modelo de terminación*. Diseñado para manejar solo excepciones síncronas con un solo *hilo* de control, el mecanismo de excepciones implementa un camino alternativo de una sola vía en diferentes sitios de su programa.



Ejemplo 25.1

Se escribe un fragmento de código Java en el que en el método `main()` se define un bloque `try`, en el que es invocado el método `f()` que lanza una excepción.

```

void escucha( ) throws Exception
{
    // código que produce una excepción que se lanza
    // ...
    throw new FException( );
    // ...
}
  
```

```

static public void main(String [ ] a)
{
    try
    {
        escucha( ); // método que puede lanzar cualquier excepción
        // Código normal aquí
    }
    catch(FException t)
    {
        // Captura una excepción de tipo FException
        // Hacer algo
    }
    catch(Exception e)
    {
        // Captura cualquier excepción
        // Hacer algo
    }
    // Resto código
}

```

escucha() es un simple procedimiento en el que se ha declarado que puede lanzar cualquier excepción. Esto se hace en la cabecera del método, con

```
throws Exception
```

Exception es clase base de las excepciones que son tratadas. Debido a la conversión automática entre clase derivada y clase base, se afirma que puede lanzar cualquier excepción. Cuando el método encuentra una condición de error lanza una excepción. Esto se hace con la sentencia:

```
throw new FException ( );
```

El operando de la expresión throw es un objeto. Normalmente los objetos almacenan información sobre el error ocurrido. Java proporciona una jerarquía de clases para el manejo de excepciones; el programador puede declarar clases para el tratamiento de errores en la aplicación

En el método main(), la llamada a escucha() se encierra en un bloque try. Este es un bloque de código encerrado dentro de una sentencia try:

```

try
{
    // ...
}

```

Un bloque catch() captura una excepción del tipo indicado. En el ejemplo anterior se han especificado dos:

```

catch(FException t)
catch(Exception e)

```

Una expresión catch es comparable a un procedimiento con un único argumento.

Diseño de excepciones

La palabra reservada try designa un bloque *try*, que es un área de su programa que detecta excepciones. En el interior de bloques try, normalmente se llaman a métodos que pueden levantar o *lanzar* excepciones. La palabra reservada catch designa un manejador de capturas con una firma que representa un tipo de excepción. Los manejadores de captura siguen inmediatamente a bloques try o a otro manejador catch con un argumento diferente.

Los bloques `try` son importantes ya que sus manejadores de captura asociados determinan cuál es la parte de su programa que maneja una excepción específica. El código que está dentro del manejador de capturas (`catch`) es donde se decide lo que se hace con la excepción lanzada.

Java proporciona un manejador especial, denominado `finally`. Es opcional, de utilizarse debe escribirse después del último `catch()`. En general, la finalidad que tiene es liberar recursos asignados en el bloque `try`, por ejemplo cerrar archivos abiertos en alguna sentencia del bloque `try`. Este manejador tiene la propiedad de que siempre se ejecuta, una vez que ha terminado el bloque, o bien una vez que una excepción ha sido capturada por el correspondiente `catch()`.

Bloques `try`

Un bloque `try` encierra las sentencias que pueden lanzar excepciones. El bloque comienza con la palabra reservada `try` seguida por una secuencia de sentencias de programa encerradas entre llaves. A continuación del bloque `try` hay una lista de manejadores, llamados cláusulas `catch`. Al menos un manejador `catch` debe aparecer inmediatamente después de un bloque `try`, o si no hay manejador `catch` debe especificarse el manejador `finally`. Cuando un tipo de excepción lanzada coincide con el argumento de un `catch`, el control se reanuda dentro del bloque del manejador `catch`. Si ninguna excepción se lanza desde un bloque `try`, una vez que terminan las sentencias del bloque, prosigue la ejecución a continuación del último `catch`. Si hay manejador `finally` (es opcional), la ejecución sigue por sus sentencias; una vez terminadas, continúa la ejecución en la sentencia siguiente.

La sintaxis del bloque `try` es:

1 <code>try</code>	2 <code>try</code>
<code>{</code>	<code>sentencia compuesta</code>
<code> código del bloque try</code>	<code>lista de manejadores</code>
<code>}</code>	
<code>catch (signatura)</code>	
<code>{</code>	
<code> código del bloque catch</code>	
<code>}</code>	

También se puede anidar bloques `try`.

```
void sub(int n) throws Exception
{
    try {
        ...
        // bloque try externo
        try { // bloque try interno
            ...
            if (n == 1)
                return ;
        }
        catch (signatura1) {...} // manejador catch interno
    }
    catch (signatura2) {...} // manejador catch externo
    ...
}
```

Una excepción lanzada en el bloque interior `try` ejecuta el manejador `catch` con `signatura1` si coincide el tipo de excepción. El manejador `catch` con `signatura2` maneja excepciones lanzadas desde el bloque `try` exterior si el tipo de la excepción coincide. El manejador externo de `catch` también captura excepciones lanzadas desde el bloque interior si el tipo de excepción coincide con `signatura2` pero no con `signatura1`. Si los tipos de excepción no coinciden con ninguna `signatura`, la excepción se propaga al llamador de `sub()`.

Normas

`try`

```

{
    sentencias
}
catch (parámetro)
{
    sentencias
}
catch (parámetro)
{
    sentencias
}
etc.

```

1. Cuando una excepción se produce en sentencias del bloque `try` hay un salto al primer manejador (`catch`) cuyo parámetro coincide con el tipo de excepción.
2. Cuando las sentencias en el manejador se han ejecutado, se termina el bloque `try` y la ejecución prosigue en la sentencia siguiente. Nunca se produce un salto hacia atrás, al lugar en que ocurrió la interrupción.
3. Si no hay manejadores para tratar con una excepción, se aborta el bloque `try` y la excepción se re-lanza.
4. Si se utiliza el manejador opcional `finally`, se escribe después del último `catch`. La ejecución del bloque `try`, se lance o no una excepción, siempre termina con las sentencias de `finally`.

Precaución

Se puede transferir el control fuera de bloques `try` con una sentencia `goto`, `return`, `break` o `continue`. Si se ha especificado el manejador `finally`, primero se ejecuta este y después transfiere el control.

Ejemplo 25.2

El método `calcumedia ()` calcula una media de arreglos de tipo `double`; para ello invoca al método `avg ()`. En el caso de ser llamado incorrectamente, `avg ()` lanza excepciones del tipo `MediaException`.

La excepción que se va a lanzar en caso de error, `MediaException`, debe ser declarada como una clase derivada de `Exception`:

```

class MediaException extends Exception
{
    //
}

```

El método `calcumedia ()` define un bloque `try` para tratar excepciones:

```

double calcumedia (int num)
{
    double b[ ] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
    double media;

    try
    {
        media = avg (b, num); // calculo media
        return media;
    }
    catch (MediaException msg)
    {
        System.out.println ("Excepcion captada: " + msg);
        System.out.println ("Calcula Media: uso de longitud del array "
                           + b.length);
    }
}

```

```

        num = b.length;
        return avg (b, num);
    }
}

```

El método `avg ()` tiene que tener en la cabecera la excepción que puede lanzar:

```
double avg (double [ ] p, int n) throws MediaException
```

El método `calcuMedia ()` define un arreglo de tipo `double` y llama a `avg ()` con el nombre del arreglo (`b`) y un argumento de longitud (`num`). Un bloque `try` contiene a `avg ()` para capturar excepciones de tipo `MediaException`.

Si `avg` lanza la excepción `MediaException`, el manejador de `catch` la captura, escribe un mensaje y vuelve a llamar a `avg ()` con un valor predeterminado (la longitud del arreglo `b`).

Lanzamiento de excepciones

La sentencia `throw` levanta una excepción. Cuando se encuentra una excepción la parte del programa que detecta la excepción puede comunicar que la excepción ha ocurrido por levantamiento, o *lanzamiento de una excepción*. El formato de `throw` es:

```
throw objeto
```

El operando de `throw` ha de ser un objeto de una clase derivada de la clase `Exception`.

Una excepción lanzada hace que termine el bloque `try`, las sentencias que siguen no se ejecutan. El objeto que se lanza puede contener información relativa al problema que ha surgido.

El manejador `catch` que captura una excepción realiza un proceso con ella y puede decidir devolver control, `return`, o continuar la ejecución en el mismo método, a continuación del último `catch`. Incluso, la excepción actual se puede relanzar con la misma sentencia: `throw objeto`. Normalmente se utiliza cuando se desea que un segundo manejador sea llamado desde el primero para procesar después la excepción.



Ejemplo 25.3

Se supone tres métodos, el método `main ()` tiene un bloque `try` en el que se llama a `deolit ()`; este método tiene su bloque `try` en el que se llama a `lopart ()`. Este último lanza una excepción que es atrapada por el `try-catch` correspondiente a `deolit ()`, que a su vez relanza la excepción.

```

void lopart ( ) throws NuevaExcepcion
{
    //
    throw new NuevaExcepcion ( );
}
void deolit ( ) throws NuevaExcepcion
{
    int i;
    try
    {
        i = -16;
        lopart ( );
    }
    catch (NuevaExcepcion n)
    {
        System.out.println ("Excepción captada, se relanza");
        throw n;
    }
}
public static void main (String [ ] a)

```

```

{
  try
  {
    deolit ( )
  }
  catch (NuevaExcepcion er)
  {
    System.out.println ("Excepción capturada en main ( )");
  }
}

```

Captura de una excepción: catch

La cláusula de captura (`catch`) constituye el manejador de excepciones. Cuando una excepción se lanza desde alguna sentencia de un bloque `try`, se pone en marcha el mecanismo de captura. La excepción será capturada por un `catch` de la lista de cláusulas `catch` que siguen al bloque `try`.

Una cláusula `catch` consta de tres puntos: la palabra reservada `catch`, la declaración de un único argumento que ha de ser un objeto para manejo de excepciones (*declaración de excepciones*), y un bloque de sentencias. Si la cláusula `catch` se selecciona para manejar una excepción, se ejecuta el bloque de sentencias.

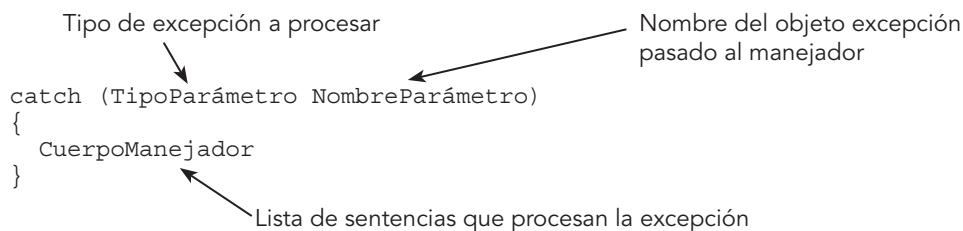
Formato

```

catch (argumento) //captura excepción del tipo del argumento
{
  código del bloque catch
}

```

La especificación del manejador `catch` se asemeja a la definición de un método.



Al contrario que el bloque `try`, el bloque `catch` solo se ejecuta bajo circunstancias especiales. El argumento es la excepción que puede ser capturada, una referencia a un objeto de una clase ya definida, o bien derivada de la clase base `Exception`.

Al igual que con los métodos sobrecargados, el manejador de excepciones se activa solo si el argumento que se pasa (o se *lanza*, “*thrown*”) se corresponde con la declaración del argumento. Hay que tener en cuenta que hay una correspondencia automática entre clase derivada y clase base.

```

catch (Excepcion1 e1) {...}
catch (Excepcion2 e2) {...}
...
catch (Exception e) {...}

```

El último `catch` significa “cualquier excepción” ya que `Exception` es la superclase base de la jerarquía de clases que tratan las anomalías. Se puede utilizar como manejador de excepciones de manera predeterminada ya que captura cualquier excepción.

La sintaxis completa de `try` y `catch` permite escribir cualquier número de manejadores de excepciones al mismo nivel.

```

try
{
    sentencias
}
catch (parámetro1)
{
    sentencias
}
catch (parámetro2)
{
    sentencias
}
...

```

Los puntos suspensivos (...) significa que puede tener cualquier número de bloques catch a continuación del bloque try.

Precaución

La clase base de la cual derivan las excepciones es `Exception`. Por ello el manejador `catch (Exception e)` captura cualquier excepción lanzada. De utilizarse debe situarse el último en el bloque `try-catch`.

Funcionamiento

Cuando ocurre una excepción en una sentencia durante la ejecución del bloque `try`, el programa comproba, por orden, cada bloque `catch` hasta que encuentra un manejador (`catch`) cuyo argumento coincide con el tipo de excepción. Tan pronto como se encuentra una coincidencia, se ejecutan las sentencias del bloque `catch`; cuando se han ejecutado las sentencias del manejador, se termina el bloque `try` y prosigue la ejecución con la siguiente sentencia. Es decir, si el método no ha terminado, la ejecución se reanuda después del final de todos los bloques `catch`.

Si no existen manejadores para tratar con una excepción, esta se relanza para ser tratada en el bloque `try` del método llamador anterior. Si no ocurre ninguna excepción, las sentencias se ejecutan de modo normal y ninguno de los manejadores será invocado.



Ejemplo 25.4

Se escriben diversos `catch` para un bloque `try`. Las clases que aparecen están declaradas en los diversos paquetes de Java, o bien son clases definidas por el usuario.

```

try
{
    // sentencias, llamadas a métodos
}
catch (IOException ent)
{
    System.out.println ("Entrada incorrecta desde el teclado");
    throw ent;
}
catch (SuperaException at)
{
    System.out.println ("Valor supera máximo permitido ");
    return ;
}
catch (ArithmeticException a)
{

```

```

        System.out.println ("Error aritmético, división por cero... ");
    }
    catch (Exception et)
    {
        System.out.println ("Cualquier otra excepción ");
        throw et;
    }
}

```

Cláusula `finally`

En un bloque `try` se ejecutan sentencias de todo tipo, llamadas a métodos, creación de objetos ...; en ciertas aplicaciones se pedirán recursos al sistema para ser utilizados. Las aplicaciones que procesan información abren ficheros para leerlos, ficheros que se asignarán con *uso exclusivo*. Es obvio que todos estos recursos a los que se ha hecho mención tienen que ser liberados cuando el bloque `try` en que se han asignado termina; de igual forma, si no se ejecutan *normalmente* todas las sentencias del bloque porque ha habido alguna excepción, el `catch` que la captura debe preocuparse de liberar los recursos.

Java proporciona la posibilidad de definir un bloque de sentencias que se ejecutarán siempre, ya sea que termine el bloque `try` normalmente, o se produzca una excepción. La cláusula `finally` define un bloque de sentencias con esas características.

La cláusula `finally` es opcional, si está presente debe de situarse después del último `catch` del bloque `try`. Incluso, se permite que un bloque `try` no tenga `catch` asociados si tiene el bloque definido por la cláusula `finally`.

El esquema siguiente indica cómo se especifica un bloque `try-catch-finally`:

```

try
{
    // sentencias
    // acceso a archivos (uso exclusivo ...)
    // peticiones de recursos
}
catch (Excepcion1 e1) {...}
catch (Excepcion2 e2) {...}
finally
{
    // sentencias
    // desbloqueo de archivos
    // liberación de recursos
}

```

Se declara la excepción `RangoException` que será lanzada en el caso de que un valor entero esté comprendido en un intervalo determinado. El bloque `try-catch-finally` se ejecuta; termine de una forma u otra siempre se ejecutan las sentencias del bloque que define `finally`, lo que se pone de manifiesto con una salida por pantalla.

Ejemplo 25.5

```

import java.io.*;
class ConFinally
{
    static void meteo ( ) throws RangoException
    {
        for (int i= 1; i<9; i++)
        {
            int r;
            r = (int)(Math.random( )*77);
            if (r< 3 || r>71)

```

```

        throw new RangoException ("con el valor " + r);
    }
    System.out.println ("fin método meteo");
}
static void gereal ( )
{
    for (int h = 1; h < 9; h++)
    try
    {
        meteo ( );
    }
    catch (RangoException r)
    {
        System.out.println ("Excepción " + r + " capturada");
        System.out.println ("Iteración: " + h);
        break;
    }
    finally
    {
        System.out.println ("Bloque final de try en gereal ( )");
    }
}
public static void main (String [ ] arg)
{
    try
    {
        System.out.println ("Ejecuta main ");
        gereal ( );
    }
    finally
    {
        System.out.println ("Bloque final de try en main ( )");
    }
    System.out.println ("Termina el programa ");
}
}

class RangoException extends Exception
{
    public RangoException (String msg)
    {
        super (msg);
    }
}

```

La ejecución de este programa da lugar a esta salida:

```

Ejecuta main
Excepción: con el valor 2 capturada
Iteración 1
Bloque final de try en gereal ( )
Bloque final de try en main ( )
Termina el programa

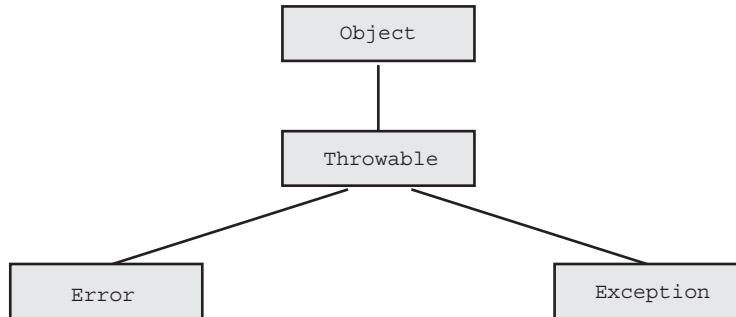
```

Se observa que al generarse el número aleatorio 2, el método `meteo ()` lanza la excepción `RangoException`. Esta es captada por el `catch` que tiene el argumento `RangoException`, ejecuta las sentencias definidas en su bloque y se sale del bucle. A continuación, se ejecutan las sentencias del bloque

finally y devuelve control al método main (); acaba el bloque try de este y de nuevo se ejecuta el bloque finally correspondiente.

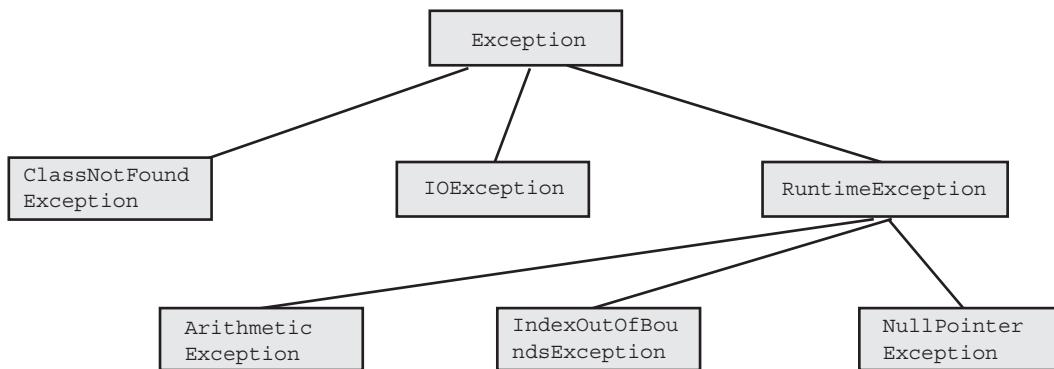
25.3 Clases de excepciones definidas en Java

El lenguaje Java pretende estandarizar el manejo de excepciones y por ello declara un amplio conjunto de clases de excepciones. Estas clases forman una jerarquía de clases, en la que la clase base es Throwable, que deriva directamente de la superclase base Object.



De Throwable derivan dos clases: Error y Exception. Las excepciones del tipo Error son generadas por el sistema, se puede decir que se trata de errores irrecuperables; además es extraño que se produzcan, por ejemplo, “salir de la memoria de la máquina virtual”; por lo tanto, de producirse una excepción Error se propagará hasta salir por el método main (). Los nombres de las subclases que derivan de Error acaban con el sufijo Error, así InternalError o NoClassDefFoundError.

De la clase Exception derivan clases de las que se instancian objetos (excepciones) para ser lanzados (throw), que pueden ser capturados por los correspondientes catch. Las clases derivadas de Exception se encuentran en los diversos paquetes de Java, todas tienen como nombre un identificador que indica su finalidad, terminado en Exception. La jerarquía de excepciones a partir de la base Exception:



RuntimeException

Hay excepciones que se propagan automáticamente por los métodos sin necesidad de especificarlas en la cabecera con throws. Todas las excepciones del tipo RuntimeException (división por cero, índice fuera de rango ...) son de ese tipo. Sería muy tedioso especificar en todos los métodos que se puede propagar una excepción de, por ejemplo, división por cero, y por ello Java las propaga sin tener que especificar ese hecho.

El método histog () de la clase ConHisto, tiene una expresión aritmética en la que puede producirse una excepción debido a una división por cero. La excepción se propaga y se capta en main ().

```

class ConHisto
{
    static int histog ( )
    {
        int k, r, z;
        z = 0;  k = 9;
    }
}
  
```

```

        while (k > 1)
        {
            r = (int)(Math.random() * 13);
            System.out.print("r = " + r);
            z += r + (2 * r) / (r - k);
            System.out.println("z = " + z);
            k--;
        }
        return z;
    }
    static public void main (String [ ] ar)
    {
        try
        {
            System.out.println ("Bloque try. Llamada a histog ()");
            histog ();
        }
        catch (ArithmetricException a)
        {
            System.out.println ("\tCaptura de excepción, " + a);
        }
        catch (RuntimeException r)
        {
            System.out.println ("\tCaptura de excepción, " + r);
        }
    }
}

```

Una ejecución (al depender de los números aleatorios no siempre se genera excepción) da lugar a esta salida:

```

Bloque try. Llamada a histog ()
r = 6 z = 2
r = 11 z = 20
r = 4 z = 22
r = 6 Captura de excepción, java.lang.ArithmetricException: /by zero

```

Al tomar `r` el valor 6 y `k` también 6 el programa detecta una *división por cero* y lanza la excepción. El método `histog ()` no la captura, por lo que se propaga y alcanza `main ()` que sí la captura.

Excepciones comprobadas

Los métodos en los que se lanzan excepciones, bien directamente con `throw`, o bien porque llaman a otro método que propaga una excepción, deben comprobar, verificar, la excepción con el apropiado `try-catch`, si no la excepción se propaga. Esto exige que en la cabecera del método, con la cláusula `throws` se especifiquen los tipos de excepción que permite que sean propagados. Por ejemplo, el método `readLine ()` de la clase `BufferedReader` puede lanzar la excepción del tipo `IOException`, en el caso de tratarla hay que especificar su propagación. Observe este ejemplo:

```

BufferedReader entrada = new BufferedReader(
    new InputStreamReader (System.in) );

int entradaRango ( )  !! error ;;
{
    int d;
    do {
        d = Integer.parseInt(entrada.readLine () )
    }while (d<=0 || d>=10);
    return d;
}

```

El método `entradaRango ()` tiene un error, error en tiempo de compilación. Se debe a que `readLine ()` está implementado con esta cabecera:

```
String readLine ( ) throws IOException
```

Sin embargo `entradaRango ()` no tiene un bloque `try-catch` para tratar la posible excepción, ni tampoco especifica que se puede propagar, lo cual es un error en Java. La forma de evitar el error es especificando la propagación de la excepción:

```
int entradaRango ( ) throws IOException
```

o bien capturarla. La propagación puede llegar al método `main ()`, hay que capturarla. También está la opción de no capturarla y poner: `main () throws IOException`, aunque esto no es una práctica recomendada.

En el ejemplo aparece una llamada al método `parseInt ()`, que puede generar la excepción `NumberFormatException` del tipo `RuntimeException` que no es necesario especificar su propagación. Las excepciones del tipo `RuntimeException` no hay que especificarlas.

Precaución

Es error de compilación no capturar las excepciones que propaga un método, o no especificar que se propagará una excepción.

Java permite que en excepciones de la jerarquía que tiene como base `RuntimeException` no se especifique su propagación.

Métodos que informan de la excepción

La clase `Exception` tiene dos constructores, uno sin argumentos (constructor de manera predeterminada) y el otro con un argumento que se corresponde con una cadena de caracteres.

```
Exception ( );
Exception (String m);
```

Por lo que se puede lanzar la excepción con una cadena informativa:

```
throw new Exception ("Error de lectura de archivo");
```

El método `getMessage ()` devuelve una cadena que contiene la descripción de la excepción, es la cadena pasada como argumento al constructor; su prototipo es:

```
String getMessage ( );
```

A continuación se escribe un ejemplo en el que el método `primero ()` llama a `lotes ()`, que lanza una excepción; es capturada y se imprime el mensaje con el que fue construida.

```
void lotes ( ) throws Exception
{
    //...
    throw new Exception ("Defecto en el producto.");
}
void primero ( )
{
    try
    {
        lotes ( );
    }
    catch (Exception msg)
    {
        System.out.println (msg.getMessage ( ) );
    }
}
```

El método `printStackTrace ()` resulta muy útil para conocer la secuencia de llamadas a métodos realizadas hasta que llega al método donde se ha producido el problema, donde se ha lanzado la excepción. El propio nombre, `printStackTrace ()`, ya indica su finalidad: escribir la cadena con que se inicializa el objeto excepción (si no se ha utilizado el constructor de manera predeterminada) y a continuación la identificación de la excepción junto al nombre de los métodos por donde se ha ido propagando.



Ejemplo 25.6

Se va a generar una excepción de división por cero. La excepción no es capturada por lo que se pierde en el método `main ()` y termina la ejecución. En pantalla se muestra la traza de llamadas a los métodos que ha acabado con una excepción `RuntimeException`.

```
class ExcpArt
{
    static void atime ( )
    {
        int k, r;
        r = (int) Math.random ( );
        System.out.println ("Método atime");
        k = 2/r;
    }
    static void batime ( )
    {
        System.out.println ("Método batime");
        atime ( );
    }
    static void zatime ( )
    {
        System.out.println ("Método zatime");
        batime ( );
    }
    static public void main (String [ ] ar)
    {
        System.out.println ("Entrada al programa, main( )");
        zatime ( );
    }
}
```

La ejecución muestra por pantalla:

```
Entrada al programa, main ( )
Método zatime
Método batime
Método atime
Exception in thread "main" java.lang.ArithmetricException: / by zero
at ExcpArt.atime (ExcpArt.java: 8)
at ExcpArt.batime (ExcpArt.java: 13)
at ExcpArt.zatime (ExcpArt.java: 18)
at ExcpArt.main (ExcpArt.java: 24)
```

Nota

Una excepción que no es capturada por ninguno de los métodos donde se propaga, llegando al método `main ()`, termina la ejecución y hace internamente una llamada a `printStackTrace ()` que visualiza la ruta de llamadas a métodos y los números de línea.

25.4 Nuevas clases de excepciones

Las clases de excepciones que define Java pueden ser ampliadas en las aplicaciones. Se pueden definir nuevas clases de excepciones para que así las aplicaciones tengan su específico control de errores.

La clases que se definen tienen que derivar de la clase base `Exception`, o bien directa o indirectamente. Por ejemplo,

```
public class MiExcepcion extends Exception { ... }
public class ArrayExcepcion extends NegativeArraySizeException { ... }
```

Es habitual que la clase que se define tenga un constructor con un argumento cadena en el que se puede dar información sobre la anomalía generada. En ese caso, a través de `super` se llama al constructor de la clase base `Exception`.

```
public class MiExcepcion extends Exception
{
    public MiExcepcion (String info)
    {
        super(info);
        System.out.println ("Constructor de la clase.");
    }
}
```

Las clases que definen las excepciones pueden tener cualquier tipo de atributo o método que ayude al manejo de la anomalía que representan. La siguiente clase guarda el tamaño con que se ha intentado dar tamaño a un *array*.

```
public class ArrayExcepcion extends NegativeArraySizeException
{
    int n;
    public ArrayExcepcion (String msg, int t)
    {
        super (msg);
        n = t;
    }
    public String Informa ( )
    {
        return getMessage ()+ n;
    }
}
```

Se pueden lanzar excepciones de tipo definido en las aplicaciones, por ejemplo:

```
int tam;
int [ ] v;
tam = entrada.nextInt ( );
if (tam <= 0)
    throw new ArrayExcepcion ("Tamaño incorrecto.",tam);
```

En aplicaciones complejas que lo requieran se puede definir una jerarquía propia de clases de excepciones, siempre la clase base ha de ser `Exception`. El siguiente esquema muestra una jerarquía para la aplicación *Cajero*:

```
public class CajeroException extends Exception {...}
public class CuentaException extends CajeroException{...}
public class LibretaException extends CajeroException{...}
public class CuentaCorrienteException extends CuentaException{...}
```

25.5 Especificación de excepciones

La técnica de manejo de excepciones se basa, como se ha visto antes, en la *captura de excepciones* que ocurren al ejecutar las sentencias. Una pregunta que cabe hacerse es: “¿Cómo se conoce cuál es el tipo de excepción que puede generar un método? Una forma natural de conocer esta cuestión es leer el código de los métodos, pero en la práctica, esto no es posible en métodos que son parte de grandes programas y, además llaman a otros métodos. Otra forma, es leer la documentación disponible sobre la clase y los métodos, y esperar que contenga información sobre los diferentes tipos de excepciones que se pueden generar. Desgraciadamente, tal información no siempre se puede encontrar.

Java ofrece una tercera posibilidad, que además, para evitar errores de sintaxis, obliga a utilizar. Consiste en declarar en la cabecera de los métodos las excepciones que puede generar. Esto se conoce como *especificación de excepciones*. La especificación lista las excepciones que un método puede lanzar. Garantiza que el método no lance ninguna otra excepción, a no ser las excepciones de tipo `RuntimeException` que no es necesario especificar. Los sistemas bien diseñados con manejo de excepciones necesitan definir qué métodos lanzan excepciones y cuáles no. Con especificaciones de excepciones se describen exactamente cuáles excepciones, si existen, lanza el método.

El formato de especificación de excepciones es:

```
tipo nombreMetodo (signatura) throws e1, e2, en
{
    cuerpo del metodo
}
e1, e2, en lista separada por comas de nombres de excepciones (la lista especifica que el método
    puede lanzar directa o indirectamente, incluyendo excepciones derivadas públicamente de
    estos nombres)
```

Java distingue en este aspecto dos tipos de excepciones: aquellas que *se deben comprobar* y aquellas que no es necesario. Estas últimas son las que se producen en expresiones aritméticas, índices de *array* fuera de rango, formato de conversión incorrecto...

En las excepciones que *se deben comprobar*, Java obliga a su captura con un bloque `try-catch`; o bien, a su propagación al método llamador, y para ello es necesario especificar en la cabecera del método que puede propagar la excepción con la cláusula `throws` seguida del nombre de la excepción; en caso contrario se produce un error de sintaxis, de compilación. Todas las excepciones que el programador define *se deben comprobar*. En general, todas las excepciones, excepto `RuntimeException` (y las de tipo `Error`), son las que *se deben comprobar*.

Sintácticamente, una *especificación de excepciones* es parte de una definición de un método. Formato:

```
tipo nombreMetodo(lista arg,s) throws lista excepciones
```

Ejemplo

```
void metodoF(argumentos) throws T1, T2
{
    ...
}
```

↑
especificación de excepciones

Regla

Una especificación de excepciones sigue a la lista de argumentos del método. Se especifica con la palabra reservada `throws` seguida por una lista de tipos de excepciones, separadas por comas.

Regla

En la especificación de excepciones se sigue la regla en cuanto a conversión automática de tipo clase derivada a clase base. De tal forma que para expresar que se puede propagar o lanzar cualquier excepción:

```
tipo nombreMetodo(arg,s) throws Exception
```

Al ser `Exception` la clase base de todas las excepciones a comprobar.

Ejemplo

Métodos de la clase PilaTest con especificación de excepciones.

```
class PilaTest
{
    // ...
    public void quitar (int valor) throws EnPilaVaciaException {...}
    public void meter (int valor) throws EnPilaLlenaException {...}
    // ...
}
```

Ejercicio 25.1

Cálculo de las raíces o soluciones de una ecuación de segundo grado.

Una ecuación de segundo grado $ax^2 + bx + c = 0$ tiene dos raíces:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Los casos de indefinición son: 1) $a = 0$; 2) $b^2 - 4ac < 0$, que no produce raíces reales, sino imaginarias. En consecuencia, se consideran dos excepciones:

NoRaizRealException y CoefAceroException.

La clase EcuacionSegGrado tiene como atributos a, b y c, que son los coeficientes de la ecuación; además, r1, r2, que son las raíces. El método raices () calcula las raíces, su declaración es:

```
void raices( ) throws NoRaizRealException, CoefAceroException
```

El cuerpo del método:

```
void raices( ) throws NoRaizRealException, CoefAceroException
{
    double descr;
    if (b*b < 4*a*c)
        throw new NoRaizRealException ("Discriminante negativo",a,b,c);
    if (a==0)
        throw new CoefAceroException ("No ecuac. segundo grado ",a,b,c);
    descr = Math.sqrt (b*b - 4*a*c);
    r1 = (-b - descr) / (2*a);
    r2 = (-b + descr) / (2*a);
}
```

raices () lanza excepciones si no tiene raíces reales o si el primer coeficiente, a, es cero. El método no captura excepciones sino que se propagan, es necesario la cláusula throws con los tipos NoRaizRealException, CoefAceroException.

El método desde el que se llame a raices () debe tener un bloque try-catch para capturar a las excepciones lanzadas, o bien se propagarán y esto exige de nuevo la cláusula throws.

La siguiente aplicación define las clases con las excepciones mencionadas anteriormente. En el método main () se piden los coeficientes de la ecuación, se crea el objeto, se llama al método de cálculo y de escribir por pantalla.

```
import java.io.*;
import java.util.*;
// representa la excepción: ecuación no tiene solución real
class NoRaizRealException extends Exception
{
    private double a,b,c;
    public NoRaizRealException(String m, double a, double b, double c)
```

```
{  
    super (m);  
    this.a = a;  
    this.b = b;  
    this.c = a;  
}  
public String getMessage ( )  
{  
    return "Para los coeficientes "+(float)a +", " +  
           (float)b + ", " +(float)c +super.getMessage ( );  
}  
}  
// representa la excepción: no es ecuación de segundo grado  
class CoefAceroException extends Exception  
{  
    public CoefAceroException (String m)  
    {  
        super (m);  
    }  
}  
// clase para representar cualquier ecuación de segundo grado  
class RaiceSegGrado  
{  
    private double a,b,c;  
    private double r1,r2;  
    public RaiceSegGrado (double a, double b, double c)  
    {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
    public void raices ( ) throws NoRaizRealException, CoefAceroException  
    {  
        ...  
    }  
    public void escribir ( )  
    {  
        System.out.println ("Raices de la ecuacion; r1 = "  
                           + (float)r1 + " r2 = " + (float)r2);  
    }  
}  
// clase principal  
public class Raices  
{  
    public static void main (String [ ] ar)  
    {  
        RaiceSegGrado rc;  
        double a,b,c;  
        Scanner entrada = new Scanner(System.in);  
        // entrada de coeficientes de la ecuación  
        System.out.println ("Coeficientes de ecuación de segundo grado");  
        System.out.println (" a = ");  
        a = entrada.nextDouble ( );  
        System.out.println (" b = ");  
        b = entrada.nextDouble ( );  
        System.out.print (" c = ");  
        c = entrada.nextDouble ( );  
    }  
}
```

```

// crea objeto ecuación y bloque para captura de excepciones
try
{
    rc = new RaiceSegGrado(a, b, c) ;
    rc.raices ( ) ;
    rc.escribir ( ) ;
}
catch (NoRaizRealException er)
{
    System.out.println(er.getMessage ( ) ) ;
}
catch (CoeficienteAcero er)
{
    System.out.println (er.getMessage ( ) ) ;
}
}

```

Se puede observar que en el constructor de la clase `NoRaizRealException` se hace una llamada al constructor de la clase base, `Exception`, para pasar la cadena. También se ha redefinido el método `getMessage()`, de tal forma que devuelve la cadena con que se inicializa al objeto excepción concatenada con los coeficientes de la ecuación de segundo grado.

25.6 Multitarea

Los sistemas operativos permiten iniciar la ejecución de diversas aplicaciones. Ello implica trabajar con una aplicación sin que termine la ejecución de otra aplicación. Por ejemplo, mientras que se está actualizando la base de datos con los movimientos ocurridos el día anterior, se puede arrancar una aplicación que reproduzca una canción. De la gestión de la ejecución de varios programas (aplicaciones) se encargan los programas del sistema operativo y es lo que se conoce como *multitarea*. Cada programa en ejecución recibe el nombre de *proceso*. Por esa razón, la multitarea puede considerarse como la ejecución simultánea, concurrente, de múltiples procesos.

Gracias a la multitarea se puede estar editando una página Excel, y a la vez escuchar nuestras canciones favoritas en la misma computadora: computadora con un único procesador (CPU).

Java se diseñó con la funcionalidad de multitarea. El lenguaje facilita procesar varias tareas simultáneamente, es decir procesos concurrentes. La base de la multitarea en Java está en los *hilos de ejecución*, implementados a partir de la clase `Thread` (paquete `java.lang`).

Utilización de la multitarea

Un programa puede necesitar realizar más de una tarea al mismo tiempo, de tal forma que el programa arranque distintos *hilos* (*threads*) de ejecución, realizando acciones diferentes. Los navegadores (Explorador, Firefox, Chrome, ...) ejecutan múltiples *hilos* (*procesos*) para llevar a cabo su trabajo, es decir la descarga y visualización de una página web. Existirá un *hilo* encargado de la descarga de la información de la dirección web seleccionada; otro *hilo* gestiona los menús que permiten al usuario realizar otras acciones. Y más *hilos* o *procesos* se activarán según se requieran más actividades.

Cuando se ejecuta un programa basado en componentes gráficos (botones, campos de texto, tablas, ...) se ponen en marcha múltiples *hilos de ejecución*. Uno se encarga de la interacción con el usuario de la aplicación; por ejemplo, cuando se pulsa con el ratón un botón, el evento generado se recibe en una cola de eventos y, cuando salga de la cola, se realizan las acciones programadas para tal evento. Al menos habrá otro hilo que es el correspondiente a nuestro programa principal (*hilo de main* ()).

Siempre que se vayan a realizar aplicaciones con muchas operaciones de entrada/salida (lectura de un archivo, transmisión de datos por la red, etc.), se deben utilizar *hilos (threads)* de ejecución. También, cuando se necesite realizar operaciones que tengan tiempos de espera y que el usuario pueda realizar otras acciones entre tanto.

25.7 Creación de hilos

Un hilo dispone de sus propios recursos, aunque puede compartir recurso con otros hilos, como por ejemplo un archivo. Un hilo no puede ejecutarse fuera del programa en que se encuentra, no es un programa en sí mismo. Un hilo puede ejecutar cualquier tarea, basta con indicarlo en el método `run ()` (clase `Thread`), que es el que determina la actividad principal de los hilos.

Java permite crear hilos de dos formas:

1. La clase que vaya a ejecutar procesos concurrentes debe declarar que implementa la interfaz `Runnable` (`java.lang`). Esta interfaz solo define el método `run ()`. Dentro de este método será donde se pongan las acciones (sentencias) que se ejecutarán de forma concurrente.

```
class ClaseConcurre implements Runnable
{
    public void run ( ) {...}
}
```

Para crear y activar un hilo basado en la clase que implementa `Runnable` se instancia un objeto `ClaseConcurre` y después un objeto de la clase `Thread` al que se le asocia el objeto `ClaseConcurre`:

```
ClaseConcurre objetoTarea = new ClaseConcurre (parámetros);
Thread hilo = new Thread (objetoTarea);
```

Para poner en marcha el hilo:

```
hilo.start ( )
```

2. Declarar la clase con procesos concurrentes derivada de la clase `Thread`. La nueva clase hereda todos los métodos de `Thread`; deberá redefinir el método `run ()`, que será donde se sitúen las acciones a ejecutar.

```
class ClaseConcurre extends Thread
{
    public void run ( ) {...}
}
```

Para crear un hilo basado en la clase que hereda de `Thread` se instancia un objeto de la clase `ClaseConcurre`:

```
ClaseConcurre hilo = new ClaseConcurre (parámetros);
```

O bien,

```
Thread hilo = new ClaseConcurre (parámetros);
```

Para poner en marcha el hilo:

```
hilo.start ( )
```



Ejemplo 25.7

Se crean dos hilos mediante una clase que implementa la interfaz `Runnable`.

```
import java.lang.Thread; // no es necesario, java.lang siempre está presente
public class PruebaDeHilo
{
    public static void main (String [ ] a)
    {
        ClaseHilo h1, h2;
        h1 = new ClaseHilo ("Primer hilo");
        h2 = new ClaseHilo ("Segundo hilo");
    }
}
// clase diseñada para crear hilos
```

```

class ClaseHilo implements Runnable
{
    private Thread hilo;
    private String entrada;
    public ClaseHilo (String m)
    {
        entrada = m;
        hilo = new Thread (this);
        hilo.start ( );
    }
    // método en el que se definen las acciones a realizar por el hilo
    public void run ( )
    {
        System.out.println ("Comienza ejecución de: " + entrada);
        try {
            Thread.currentThread.sleep (1000);
        }
        catch (InterruptedException er)
        {
            System.out.println ("Excepción " + er);
        }
        System.out.println ("Fin de la ejecución de " + entrada);
    }
}

```

La interfaz `Runnable` declara el método `run ()`. Toda clase que implemente esa interfaz debe definir dicho método. En el ejemplo, la clase `ClaseHilo` implementa a `Runnable` y define `run ()` de tal forma que escribe una cadena y “duerme” (detiene la ejecución) durante 1000 milisegundos. El método `sleep ()` propaga la excepción `InterruptedException`, que es necesario *comprobar* en un bloque `try-catch`.

El constructor de `ClaseHilo` se encarga de crear y activar (iniciar) la ejecución del hilo, con las sentencias:

```

hilo = new Thread (this);
hilo.start ( );

```

Observar que en este contexto `this` referencia al objeto que implementa a `Runnable`.

A recordar

La interfaz `Runnable` declara el método abstracto `run ()`. Este no tiene argumentos, no devuelve tipo alguno. En la redefinición de `run ()` se especifican las acciones a realizar por el hilo, o hilos, de ejecución que se crean a partir de la clase que implementa a `Runnable`.

Declaración: `public void run ()`

Se crean dos hilos mediante una clase que deriva de `Thread`. La funcionalidad es la misma que la del ejemplo 25.7.

```

public class PruebaDeHilo
{
    public static void main (String [ ] a)
    {
        ClaseHilo h1, h2;
        h1 = new ClaseHilo ("Primer hilo");

```

Ejemplo 25.8



```

        h2 = new ClaseHilo ("Segundo hilo");
        // se pone en marcha la ejecución de los dos hilos
        h1.start ( );
        h2.start ( );
    }
}
// clase diseñada para crear hilos
class ClaseHilo extends Thread
{
    private String entrada;
    public ClaseHilo (String m)
    {
        entrada = m;
    }
    // método en el que se definen las acciones a realizar por el hilo
    public void run ( )
    {
        System.out.println ("Comienza ejecución de: " + entrada);
        try {
            sleep (1000);
        }
        catch (InterruptedException er)
        {
            System.out.println ("Excepción " + er);
        }
        System.out.println ("Fin de la ejecución de " + entrada);
    }
}

```

En el ejemplo, la clase `ClaseHilo` redefine el método `run ()` heredado de `Thread`. Este escribe la cadena pasada al constructor y “duerme” (detiene la ejecución) durante 1000 milisegundos.

Ahora el método `sleep ()` es heredado por `ClaseHilo`, se puede decir que es un método de la clase, por eso se puede llamar directamente.

Criterios a seguir para elegir cómo crear un hilo

La primera forma de crear un hilo, clase que implementa `Runnable`, se utiliza con más frecuencia. Permite a la clase heredar de otra clase. La creación de *applets* siempre se hace mediante una clase que hereda de `JApplet`, por ello, si en el *applet* se crean hilos se utilizará este método. La declaración de la clase será:

```
public class MiApplet extends JApplet implements Runnable
```

La segunda forma descrita para crear un hilo, derivar de `Thread`, tiene la bondad de la derivación de clases: todos los métodos de `Thread` son heredados por la subclase. El principal inconveniente es que solo se puede derivar de una clase, por consiguiente esta forma cierra la herencia de otras clases.

25.8 Estados de un hilo, ciclo de vida de un hilo

Los hilos se crean como cualquier otro objeto de Java y duran hasta que finaliza el método `run ()`. Todo hilo pasa por cuatro estados: *creado, ejecutable, bloqueado, eliminado*.

- **Creado.** Este es el estado del hilo cuando se llama al constructor de la clase hilo. Todavía no tiene recursos asignados.
- **Ejecutable.** A este estado se accede cuando el hilo creado llama a `start ()`. Este método asigna recursos para la ejecución del hilo, planifica su ejecución y automáticamente llama al método `run ()`, que es donde se escriben las acciones a realizar por el hilo. Puede haber varios hilos ejecutándose; en un momento dado es uno de ellos el que está en ejecución, los demás se encuentran ejecutándose.

- **Bloqueado.** Un hilo en estado bloqueado no realiza acción alguna, no recibe ciclos de CPU. Hasta que no se desbloquea no se le considera en la distribución de tiempos de ejecución. Un hilo pasa a estado bloqueado cuando llama al método `sleep()` para dormirlo n milisegundos; o bien llamadas a otros métodos como `wait()`, `join()` o `yield()`. También, un hilo se bloquea cuando está a la espera de una operación de entrada/salida.
 - **Eliminado.** Una vez que el método `run()` termina de ejecutarse el hilo pasa a estado eliminado. La clase `Thread` dispone del método `stop()` para que un hilo pase a este estado; sin embargo desde Java 2 se desaconseja utilizar este método por el riesgo de que queden objetos y recursos no liberados.

Los hilos, una vez creados, pasan cíclicamente del estado ejecutable a bloqueado. El sistema se encarga de asignar tiempos de CPU a los hilos en estado ejecutable. Cuando están en ejecución, es el método `run()` del hilo el que se ejecuta. Las acciones que bloquean temporalmente un hilo son:

- Llamada al método `sleep ()` de la clase `Thread`. Este método detiene la ejecución del hilo durante un número de milisegundos. La declaración del método es:

void sleep(long mills) throws InterruptedException

- Ejecutar el método `wait()`. Esta acción produce que el hilo se quede a la espera de que suceda un hecho que permita seguir con la ejecución del hilo. Para que el hilo vuelva al estado ejecutable tiene que recibir una notificación, mediante la llamada a `notify()`, o bien `notifyAll()`. Estos dos métodos, y también `wait()`, están definidos en la clase `Object`. Su declaración es:

```
final void notify ( )
final void notifyAll ( )
final void wait ( ) throws InterruptedException
final void wait (long msq) throws InterruptedException
```

- Estar a la espera de operaciones de entrada/salida (leer un archivo, ...).
 - Cuando desde un hilo un objeto llame a un método declarado con el atributo `synchronized` y dicho objeto esté bloqueado por otro hilo.

Normalmente el método `run ()` se diseña con un bucle cuya ejecución depende de una variable booleana, de tal forma que cuando se quiera detener cambie el estado de la variable.

La clase Thread dispone del método `isAlive()` que devuelve `true` si el hilo se encuentra en estado *ejecutable* o *bloqueado*. Un hilo activo no quiere decir que esté en ejecución, sino que no está eliminado y por consiguiente se puede utilizar. La declaración del método `isAlive()` es:

```
boolean isAlive ( );
```

Otro método de interés en el control de los hilos es `join()`. Un hilo espera la ejecución de otro hilo mediante el uso de `join()`. Este método hace que se bloquee la ejecución del hilo hasta que termine la ejecución del otro hilo que llama a `join()` (`otroHilo.join()`).

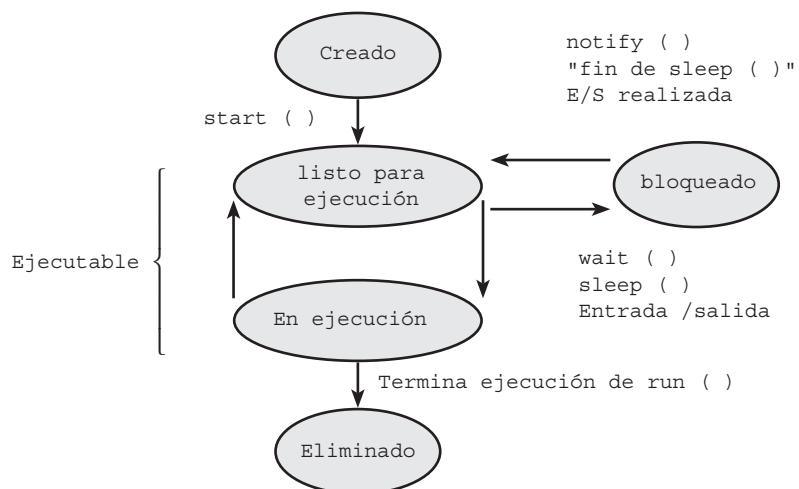


Figura 25.2 Ciclo de vida de un hilo.



Ejemplo 25.9

El programa crea tres hilos desde el hilo principal, es decir, desde `main()`. La ejecución del programa permite activar una llamada a `join()`. Los resultados de la ejecución ponen de manifiesto la funcionalidad de `join()`.

La clase `TareaS` define el hilo, implementa la interfaz `Runnable`. El método `run()` tiene un bucle con un mensaje a pantalla y llamada a `sleep()` con el fin de que el estado del hilo pase a bloqueado y se ejecute otro hilo.

```

import java.util.Scanner;
class TareaS implements Runnable
{
    private String ms;
    Thread tr;
    public TareaS (String m)
    {
        ms = m;
        tr = new Thread(this, ms);
        System.out.println ("Creación hilo:" + tr);
        tr.start (); // ejecutandose
    }
    public void run ()
    {
        try {
            for (int i = 6; i > 0; i-= 2)
            {
                System.out.println ("Hilo: " + tr + " i = " + i);
                tr.sleep (1000); // Hilo "durmiendo", queda bloqueado
            }
        }
        catch (InterruptedException e)
        {
            e.printStackTrace ();
        }
        System.out.println ("Finaliza Hilo " + ms);
    }
}

public class TresHilosJoin
{
    public static void main (String [ ] a)
    {
        Scanner entrada = new Scanner (System.in);
        int opcion;
        do {
            System.out.println ("1. Ejecutar con join ()");
            System.out.println ("2. Ejecutar sin join ()");
            opcion = entrada.nextInt ();
        } while (opcion != 1 && opcion !=2);
        // Se crean tres hilos
        TareaS t1, t2, t3;
        t1 = new TareaS ("Primero");
        t2 = new TareaS ("Segundo");
        t3 = new TareaS ("Tercero");
        System.out.println ("Hilo ejecutandose: " +
                           Thread.currentThread ());
        if (opcion == 1)
        {
    }
}

```

```

        System.out.println ("Cada hilo llamada a join( )");
        try {
            t1.tr.join ( ); //Hilo actual "bloqueado" hasta finalizar t1
            t2.tr.join ( );
            t3.tr.join ( );
        }
        catch (InterruptedException e)
        {
            e.printStackTrace ( );
        }
    }
    else
        System.out.println ("No hay llamada a join ( )");
    //
    System.out.println ("Finaliza hilo main ( )");
}
}
}

```

Al ejecutar el programa con la opción 1 se obtiene:

```

Creación hilo:Thread [Primero,5,main]
Creación hilo:Thread [Segundo,5,main]
Creación hilo:Thread [Tercero,5,main]
Hilo ejecutandose: Thread [main,5,main]
Cada hilo llamada a join ( )
Hilo: Thread [Primero,5,main] i = 6
Hilo: Thread [Segundo,5,main] i = 6
Hilo: Thread [Tercero,5,main] i = 6
Hilo: Thread [Primero,5,main] i = 4
Hilo: Thread [Tercero,5,main] i = 4
Hilo: Thread [Segundo,5,main] i = 4
Hilo: Thread [Primero,5,main] i = 2
Hilo: Thread [Tercero,5,main] i = 2
Hilo: Thread [Segundo,5,main] i = 2
Finaliza Hilo Primero
Finaliza Hilo Tercero
Finaliza Hilo Segundo
Finaliza hilo main ( )

```

Al ejecutar el programa con la opción 2 se obtiene:

```

Creación hilo:Thread [Primero,5,main]
Creación hilo:Thread [Segundo,5,main]
Creación hilo:Thread [Tercero,5,main]
Hilo ejecutandose: Thread [main,5,main]
No hay llamada a join ( )
Finaliza hilo main ( )
Hilo: Thread [Primero,5,main] i = 6
Hilo: Thread [Segundo,5,main] i = 6
Hilo: Thread [Tercero,5,main] i = 6
Hilo: Thread [Primero,5,main] i = 4
Hilo: Thread [Segundo,5,main] i = 4
Hilo: Thread [Tercero,5,main] i = 4
Hilo: Thread [Primero,5,main] i = 2
Hilo: Thread [Segundo,5,main] i = 2
Hilo: Thread [Tercero,5,main] i = 2
Finaliza Hilo Primero
Finaliza Hilo Segundo
Finaliza Hilo Tercero

```

Observe que en esta ejecución termina el hilo del método `main()` antes de los tres hilos creados desde `main()`. Por contra, la ejecución con la opción 1 (llamada a `join()`) `main()` termina en último lugar. Esto se debe a que la llamada a `join()` desde el hilo principal provoca que este se quede a la espera de que termine la ejecución de cada uno de los tres hilos implementados en Tareas.

25.9 Prioridad entre hilos

Los sistemas operativos difieren en la forma de planificar la ejecución de los hilos. Windows gestiona el reparto de los tiempos de ejecución de los hilos (tareas) de forma directamente proporcional a la prioridad asignada a cada hilo. Ello significa la concesión de una cantidad de tiempo de ejecución a un hilo; finalizado ese tiempo el sistema concede la ejecución a otro hilo. Así, de forma circular el sistema va asignando los tiempos de CPU a los hilos, con mayor tiempo a los hilos con mayor prioridad. De esta forma, los hilos con mayor prioridad se ejecutan primero, ya que proporcionalmente se les asigna mayor tiempo de ejecución. Esto no significa que un hilo con mayor prioridad se ejecute hasta terminar, sino que los tiempos de CPU se reparten proporcionalmente según la prioridad que tengan. Hay que tener en cuenta que un hilo, aunque tenga mayor prioridad, pierde el control de la ejecución si efectúa una llamada a `sleep()`; también puede ceder ese control si llama a `yield()`.

Java asigna, por omisión, la prioridad 5 (`Thread.NORM_PRIORITY`) a un hilo cuando este se crea. El rango de valores de la prioridad está determinado por las constantes: `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY`. La prioridad de un hilo se puede modificar con el método de la clase `Thread`:

```
public final void setPriority (int prioridad)
```

Para obtener la prioridad de un hilo se utiliza el método de `Thread`:

```
public final int getPriority ()
```

Por razones de seguridad, no se permite a un *applet* modificar la prioridad de un hilo.

25.10 Hilos daemon

Un hilo *demonio* tiene como propósito realizar servicios a otros hilos de ejecución. Los hilos *demonio* finalizan cuando todos los hilos “*no demonio*” finalizan, o bien cuando termina el método `run()` asociado. El ejemplo típico de hilo demonio es el “recolector de basura” (*garbage collection*), el cual se ejecuta en paralelo con el resto de hilos de la aplicación. El método, de la clase `Thread`, `setDaemon()` aplicado a un hilo, con el argumento `true`, después de su creación lo define como demonio. Por ejemplo:

```
Thread hilo = new ClaseHilo ();
hilo.setDaemon (true);
```

Un “*demonio*” se puede considerar un hilo de servicio. Su existencia es para suministrar algún tipo de servicio a otros hilos, por eso si los hilos a los que dan servicio desaparecen, también desaparecen los *demonio*.

25.11 Sincronización

En las aplicaciones es normal que existan muchos hilos en ejecución. Es posible que varios de ellos deseen modificar el mismo objeto al mismo tiempo; en este caso es necesario sincronizarlos. Por ejemplo, en una aplicación por internet de reservas de entradas para una ópera, dos hilos tratan de reservar simultáneamente el palco 5; se puede producir una situación no deseada, a los dos hilos se les está ofreciendo como libre el palco, si no se sincronizan los dos hilos se llevarían dicho palco. Otra situación que hace necesaria la *sincronización* es cuando un hilo debe esperar datos proporcionados por otro hilo.

Para solucionar este tipo de problemas se sincronizan los procesos. Los bloques de código que acceden a un mismo recurso (un objeto, un archivo,...) desde dos o más hilos constituyen secciones críticas.

Para marcar secciones críticas se utiliza el modificador `synchronized`. Por ejemplo:

```

public synchronized void asignarSeat (int n, char s)
{
    while (condición)
    {
        sentencias;
    }
}

```

De esta forma, durante el tiempo que un hilo ejecuta el método (o bloqueo de código) *sincronizado*, el *recurso* (objeto) queda *bloqueado* (en *lock*). Otros hilos que quieran acceder al mismo recurso se quedan a la espera de una notificación (*notify* ()) de que ha terminado el bloqueo.

Se diseña una clase de gestión de datos. Un método pone los datos en un *buffer* (arreglo), otro método retira datos del *buffer*. Los dos métodos se declaran sincronizados.

Ejemplo 25.10

La clase *GestorDatos* dispone de un arreglo que se crea en el constructor con un tamaño determinado por una constante (*MX*). El método *ponerDato* () asigna valores arbitrarios al *buffer*; el método *retirarDato* () extrae elementos del *buffer*. Estos dos métodos deben declararse sincronizados.

```

class GestorDatos
{
    final static int MX = 50;
    private int [ ] dat;
    private int ultimo;

    public GestorDatos ( )
    {
        dat = new int [MX];
        ultimo = -1;
    }
    public synchronized void ponerDato (int dato)
    {
        while (ultimo < dat.length - 1)
        {
            ++ultimo;
            dat [ultimo] = dato + 2*ultimo+1;
        }
    }
    public synchronized void retirarDato ( )
    {
        int datoRetirado;
        while (ultimo >= 0)
        {
            datoRetirado = dat [ultimo];
            --ultimo;
            System.out.println ("Retirado: " + dato);
        }
    }
}

```

Ahora suponga que la aplicación dispusiera de dos hilos, uno que se encargase de poner datos y el otro de retirar datos:

```

hiloGenera.start ( ); { desde run ( ) llama a ponerDato ( ) }
hiloRetira.start ( ); { desde run ( ) llama a retirarDato ( ) }

```

El primer hilo bloquea el objeto *Dato*, el otro hilo quiere acceder al mismo objeto por lo cual se queda a la espera de que quede libre (“sin *lock*”) el objeto para poder trabajar con él.

Dentro de un método sincronizado no se debe utilizar el método `sleep()` debido a que el objeto se va a quedar bloqueado durante el tiempo de `sleep`. Se recomienda utilizar el método `wait()` de la clase `Object`. El método `wait()` detiene el hilo durante los milisegundos del argumento y lo pone en cola de espera, pero deja de bloquear al objeto. El hilo deja la cola de espera cuando acaba el tiempo, o hasta que otro hilo active el método `notify()` o `notifyAll()` de ese mismo objeto. La declaración de estos métodos se encuentra en la clase `Object`:

```
public final void wait ()
public final void wait (long msgs)
public final void notify ()
public final void notifyAll ()
```

Resumen

- Las excepciones son, normalmente, condiciones (situaciones) de error imprevistas. Estas condiciones terminan el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos son: división por cero, índices fuera de límites en un arreglo, etcétera.
- Java posee un mecanismo para manejar excepciones. Las excepciones son objetos de clases de una jerarquía proporcionada por el lenguaje. El programador puede definir sus clases de excepciones, que han de derivar, directa o indirectamente, de `Exception`.
- El código Java puede levantar (*raise*) una excepción utilizando la expresión `throw`. La excepción se maneja invocando un manejador de excepciones seleccionado de una lista de manejadores que se encuentran al final del bloque `try` del manejador.
- Sintácticamente, `throw` presenta el formato:

```
throw objetoExcepcion ;
```

Lanza una excepción que es un objeto de una clase de manejo de excepciones.

- Sintácticamente, un bloque `try` tiene el formato

```
try
sentencia compuesta
lista de manejadores
```

El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada. El orden en el que están definidos los manejadores determina el orden en el que un manejador de una excepción levantada va a ser atrapada.

- La sintaxis de un manejador es:

```
catch(argumento formal)
sentencia compuesta
```

- El manejador `finally` es opcional; de utilizarse se escribe después del último `catch`. La característica principal es que siempre se ejecuta la sentencia com-

puesta especificada a continuación de `finally`, una vez que termina la última sentencia del bloque `try`, o bien a continuación del `catch` que captura una excepción.

- La especificación de excepciones es parte de la declaración de un método y tiene el formato:

```
cabecera_método throws lista_excepciones
```

- Java define una jerarquía de clases de excepciones. La clase `Throwable` es la superclase base, aunque `Exception` que deriva directamente de la anterior, es la clase base de las excepciones que son *manejadas*.
- Las clases definidas por el programador para el tratamiento de anomalías tienen que derivar directa o indirectamente de la clase `Exception`.

```
class MiException extends Exception {...}
class OtraException extends MiException {...}
```

- Una excepción lanzada en un método debe ser capturada en el método, o bien se propaga al método llamador, en cuyo caso es necesaria la especificación de la excepción en la cabecera del método (`throws`).
- Java se diseñó con la funcionalidad de multitarea. El lenguaje facilita procesar varias tareas simultáneamente, es decir procesos concurrentes.
- La base de la multitarea en Java está en los hilos de ejecución, implementados a partir de la clase `Thread` (paquete `java.lang`).
- El método `run()` (clase `Thread`) es el que determina la actividad principal de los hilos.
- Dos formas de crear hilos: la clase que vaya a ejecutar procesos concurrentes debe declarar que implementa la interfaz `Runnable`. O bien, declarar la clase con procesos concurrentes derivada de la clase `Thread`.
- Todo hilo pasa por cuatro estados: *creado, ejecutable, bloqueado, eliminado*.



Ejercicios

- 25.1 El siguiente programa que maneja un algoritmo de ordenación básico no funciona bien. Sitúe declaraciones en el código del programa de modo que se compruebe si este código funciona correctamente. Escriba el programa correcto.

```
void intercambio (int x, int y)
{
    int aux;
    aux=x;
    x=y;
    y=aux;
}
void ordenar (int [ ] v, int n)
{
    int i, j;
    for (i=0; i< n; ++i)
        for (j=i; j< n; ++j)
            if (v[ j ] < v[ j+1 ] )
                intercambio (v[ j ], v[ j+1 ]);
}
static public void main (String [ ] ar)
{
    int z[ ] = {14,13,8,7,6,12,11,10,9,-5,1,5}
    ordenar (z, 12);
    for (int i=0 ; i<12; ++i)
        System.out.print (z [ i ] + " ");
}
```

- 25.2 Escribir el código de una clase Java que lance excepciones para cuantas condiciones estime convenientes. Utilizar una cláusula `catch` que utilice una sentencia

`switch` para seleccionar un mensaje apropiado y terminar el cálculo.

Nota: Utilice una jerarquía de clase para listar las condiciones de error.

- 25.3 Escribir el código de un método en el cual se defina un bloque `try` y dos manejadores `catch`. En uno de ellos se relanza la excepción. También ha de haber un manejador `finally`, que lanzará una excepción. Determinar qué ocurre con la excepción que relanza el `catch`, para lo cual escriba un sencillo programa en el que se genere la excepción que es captada por el `catch` descrito.

- 25.4 Escribir el código de una clase para tratar el error que se produce cuando el argumento de un logaritmo neperiano es negativo. El constructor de la clase tendrá como argumento una cadena y el valor que ha generado el error.

- 25.5 Escribir un programa Java en el que se genere la excepción del ejercicio 25.4 y se capture.

- 25.6 Definir una clase para tratar los errores en el manejo de cadenas de caracteres. A continuación definir una subclase para tratar el error supuesto de cadenas de longitud mayor de 30 caracteres. Y otra subclase que maneje los errores de cadenas que tienen caracteres no alfabéticos.

- 25.7 Escribir un programa Java en el que se dé entrada a cadenas de caracteres y se capture excepciones del tipo mencionado en el ejercicio 25.6

APÉNDICE A



Estructura de un programa en Java y C/C++. Entornos de desarrollo integrados (Java 7, Java 8 y C++11)

El aprendizaje de la programación requiere no solo el conocimiento de las técnicas y metodologías de programación para realizar el análisis, diseño y construcción de programas (en un lenguaje de programación) sino también una parte práctica de laboratorio: edición, compilación, ejecución y depuración de los programas fuente escritos por los programadores.

En este apéndice se muestra el desarrollo básico de la creación y ejecución de un programa mediante las herramientas básicas (editor, compilador y depurador) o bien con un entorno de desarrollo integrado (IDE, Integrated Development Environment).

A.1 Estructura general de un programa en C/C++ y Java

En los capítulos 3, 16 y 21 se explica la estructura general de un programa escrito en C, C++ o Java y las fases que componen su proceso de desarrollo: edición (creación del programa fuente con un editor de textos), compilación (mediante un programa denominado compilador), ejecución y depuración de errores (modificación y verificación, mediante un programa denominado depurador “debugger”) del programa.

Existen dos métodos tradicionales para realizar el proceso de ejecución de un programa:

1. El método tradicional con un editor de programas fuente y un compilador cuyas órdenes se ejecutan desde la consola de línea de comandos;
2. El entorno de desarrollo integrado (IDE, Integrated Development Environment) que suele ser el elegido en el caso de desarrollos profesionales o cuando ya el programador ha adquirido suficiente habilidad con los programas editor y compilador y sobre todo el conocimiento de todo el proceso.

Si el lector va a trabajar con Java, tal vez la opción más recomendable es trabajar con el kit de desarrollo Java SE Development Kit (JDK) y con la actualización última que ofrece el sitio web de Oracle (antes Sun Microsystems) de la versión con la que piensa trabajar. Otros entornos de desarrollo son los citados anteriormente NetBeans y Eclipse que tienen también versiones para Java. Un entorno de desarrollo específico para Java y bastante eficiente es BlueJ. Si opta por trabajar con el editor y el compilador mediante la consola de línea de comandos, un editor muy acreditado es el ya citado Notepad.

A.2 Proceso de programación en Java

El aprendizaje práctico en Java se puede realizar de diferentes formas y con herramientas muy diversas, como:

- Kit de desarrollo Java (JDK, Java Development Kit)
- Entorno de desarrollo integrado, EDI (IDE, Integrated Development Environment)
- Herramientas de línea de órdenes como un editor de textos y un compilador/depurador

Las tres herramientas nos ayudarán a las tareas prácticas de programación, por lo que el programador deberá conocerlas y luego elegir la que se adapte mejor a sus necesidades. La herramienta más amigable al usuario suele ser un entorno de desarrollo integrado, aunque es un poco más tedioso de utilizar si los programas son sencillos, en cuyo caso puede ser interesante, sobre todo en la fase inicial del aprendizaje, utilizar herramientas clásicas como un editor y un compilador. Analizaremos ahora algunas de estas herramientas y explicaremos las herramientas más populares, comenzando por el kit de desarrollo JDK en la versión 7 u 8, las más utilizadas actualmente (aunque Oracle sigue dando servicio para la versión JDK 6) y los entornos de desarrollo como NetBeans, Eclipse, IntelliJIDEA y BlueJ.

Máquina virtual Java (JVM)

Java es a la vez un lenguaje compilado e interpretado. El código fuente en Java se convierte en instrucciones binarias, al igual que el código máquina de los procesadores ordinarios. Sin embargo, mientras el código fuente de C o C++ se traduce a instrucciones nativas de un procesador específico, el código fuente de Java se compila y se traduce a *bytecodes* (código de bytes), un formato universal que son instrucciones que entiende una máquina virtual; posteriormente el código resultante se interpreta por un intérprete Java que lo ejecuta en dicha máquina virtual.

El intérprete Java se puede implementar en cualquier plataforma (sistema operativo: Windows, Linux, Unix, MacOs, Solaris... y en computadoras PC, Macintosh, servidores...). El intérprete se puede ejecutar como una aplicación independiente o se puede embeber en otra pieza de software, tal como un navegador Web (Explorer, Firefox, Chrome, Safari...). El código Java compilado, en consecuencia, es implícitamente portable. La misma aplicación Java en bytecode, se puede ejecutar en cualquier plataforma que proporcione un entorno de ejecución (Java Runtime Environment, JRE)

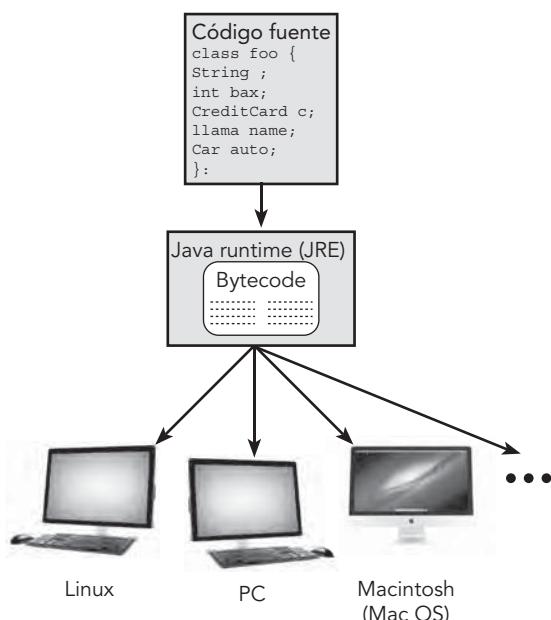


Figura A.1 El entorno de tiempo de ejecución, JRE.

Una vez traducido el programa fuente en Java, no se necesitan producir versiones alternativas de la aplicación para diferentes plataformas, cosa que sí es necesario en otros lenguajes como C y C++ y, por consiguiente no necesita distribuir el código fuente a usuarios finales, y solo el código traducido en *bytecodes*.

Una máquina virtual Java (JVM, Java Virtual Machine) es un software que implementa el sistema en tiempo de ejecución (*Java Runtime*) y ejecuta las aplicaciones Java.

Una plataforma Java SE (edición estándar) incluye el entorno de ejecución en tiempo real JRE (*Java Runtime Environment*) y el kit de desarrollo JDK (Java Development Kit), los cuales a su vez, contienen respectivamente: el lenguaje de programación Java (intérprete, compilador y depurador), otras herramientas y utilidades, la máquina virtual Java (JVM) y la biblioteca Java SE API (figura A.2).

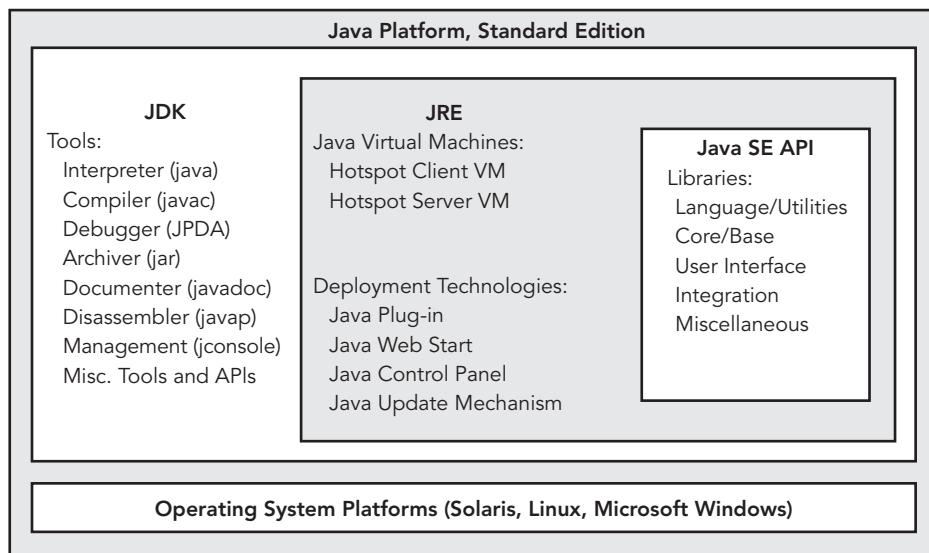


Figura A.2 Plataforma de Java SE (versiones 5, 6, 7 y 8). Fuente: www.oracle.com.

La nueva versión Java 8 incluye Java SE 8 y Java ME Embedded 8. La versión SE permite la ejecución de aplicaciones estándar y la versión ME señala Oracle (en su página oficial de Java que permite desarrollar y desplegar aplicaciones en entornos embebidos y móviles, presentes en tarjetas inteligentes, tarjetas SIM, dispositivos telefónicos (móviles y tabletas), sensores, microcontroladores, aparatos de TV, reproductores de discos Blu-Ray y otras muchas soluciones embebidas, desde cajeros de bancos a lectores de libros electrónicos, *e-book*, etc.

En ambientes profesionales y de empresa, está disponible también en el sitio web de Oracle, la plataforma Java EE (*Enterprise Edition*), aunque en el caso de aprendizaje le recomendamos utilice Java SE.

Prácticas de programación

Si desea hacer prácticas en un computador específico para desarrollar programas en Java, necesitará descargar e instalar un kit de desarrollo JDK; si por el contrario sólo desea utilizar el computador *únicamente* para ejecutar programas en Java que se hubieran compilado en otro lugar, entonces solo necesita el entorno JRE. *El kit de desarrollo JDK incluye JRE y el compilador, intérprete, depurador y otras herramientas de desarrollo adicionales y herramientas API (interfaz de programación de aplicaciones)*.

Para iniciar las prácticas de programación debe descargar una versión del kit de desarrollo JDK SE¹ (le recomendamos la *versión Java 8* o la *versión Java 7*, aunque podrá utilizar las versiones 5 o 6 si desea hacer las prácticas en estas versiones todavía muy utilizadas en laboratorios de prácticas, empresas, etc.) que incluye todas las herramientas necesarias para las diferentes etapas del desarrollo de un programa.

¹ <http://www.oracle.com/technetwork/java/javase/overview/index.html>. Las versiones disponibles en abril de 2014 son Java SE 8 y la última actualización de Java SE 7, la versión 7u51.

Kit de desarrollo Java: JDK y JDK8

JDK es una herramienta o entorno de desarrollo gratuita para construcción de aplicaciones, *applets* y componentes utilizando el lenguaje de programación Java, es decir, escribir programas Java. La herramienta fue creada por Sun Microsystem (hoy propiedad de Oracle) y consta de un conjunto de programas de líneas de órdenes que se utilizan para crear, compilar y ejecutar programas en Java. El JDK incluye herramientas útiles para el desarrollo y pruebas (*testing*) de programas escritos en el lenguaje de programación Java y su ejecución en la plataforma Java.

Oracle anunció en una nota de prensa el lanzamiento oficial de la versión 8 de Java: Java Platform Standard Edition 8 (Java SE 8) y Java Platform Micro Edition (Java ME 8) y los productos relacionados con Java Embedded. Desde el día del anuncio se puede descargar el entorno JDK 8 y las versiones correspondientes de Java. Cada nueva versión de Java viene acompañada de un kit de desarrollo. Las versiones actuales del JDK son JDK 7 y JDK 8, aunque Oracle ofrece versiones gratuitas y soporte de Java 6 y Java 5. La plataforma Java como ya se ha comentado, consta del kit de desarrollo JDK y el entorno de ejecución JRE.

JRE (Java Runtime Environment) es un conjunto de programas de software que permite a una computadora ejecutar una aplicación Java. El software consta de la máquina virtual Java (JVM) que interpreta (traduce) el código de bytes (*bytecode*) en código máquina, bibliotecas API de clases estándar, herramientas de interfaces de usuario (*toolkit*) y una variedad de utilidades.

JDK es un entorno de programación que contiene todo lo necesario para construir aplicaciones Java (compilación, ejecución y depuración) JavaBeans y *applets* de Java. JDK incluye el JRE con la adición del lenguaje de programación y herramientas de desarrollo adicionales y herramientas para API (interfaces de programación de aplicaciones). El JDK de Oracle soporta Mac OS, Solaris, Linux (Oracle, Suse, Red Hat, Ubuntu y Debian) y Windows. Otras herramientas JDK, JVM y JRE para otros sistemas operativos y de propósito especial puede consultar y descargar de modo gratuito en: <http://java-virtual-machine.net/other.html>. Los navegadores soportados son los más populares: Explorer, Firefox, Chrome y Safari.

La tabla A.1 lista las versiones de los JDK proporcionadas por Oracle. En su sitio web oficial se pueden descargar las versiones 7 y 8 de Java, y también versiones antiguas.

Tabla A.1 Versiones de Java Development Kits (JDK).

Kit de desarrollo Java	Nombre	Versión	Paquetes	Clases
Java SE 8 con JDK 1.8.0		2014		
Java SE 7 con JDK 1.7.0	Dolphin	2011	209	4.024
Java SE 6 con JDK 1.6.0	Mustang	2006	203	3.793
Java 2 SE 5.0 con JDK 1.5.0	Tiger	2004	166	3.279
Java 2 SE con SDK 1.4.0	Merlin	2002	135	2.991
Java 2 SE con SDK 1.3	Kestrel	2000	76	1.842
Java 2 con SDK 1.2	Playground	1998	59	1.520
Development Kit 1.1	–	1997	23	504
Development Kit 1.0	Oak	1996	8	212

Direcciones de Oracle

En las direcciones web que se enlistan a continuación se proporciona información para descargas gratuitas y características de Java 8, Java 7 y restantes versiones:

JDK 8

<http://openjdk.java.net/projects/jdk8/>

Descargas de JDK 8 y Java SE 8

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Descargas de JDK 8 y Java SE 8

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Información y descargas del entorno integrado de desarrollo: NetBeans

<https://netbeans.org/features/>

Descarga de NetBeans Java 7

<http://www.oracle.com/technetwork/java/javase/downloads/jdk-7-netbeans-download-432126.html>

Descarga de NetBeans Java 8

<http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html>

Características y mejoras de Java 7

<http://www.oracle.com/technetwork/java/jdk7-relnotes-418459.html>

Características y mejoras de Java 6

<http://www.oracle.com/technetwork/java/javase/features-141434.html>

Novedades de Java 8 en el portal JavaHispano

El portal especializado en Java, JavaHispano.org publica muy buena información profesional para desarrolladores y programadores en Java. En la siguiente página de JavaHispano se publica un artículo con las novedades y nuevas características de Java 8.

<http://www.javahispano.org/portada/2014/3/22/novedades-y-nuevas-caracteristicas-de-java-8.html>

Tabla A.2 Productos de Java de Oracle.

Nombre	Sigla (acrónimo)	Descripción
Java Development Kit	JDK	El software para escribir programas de Java por los programadores
Java Virtual Machine	JVM	Máquina virtual Java
Java Runtime Environment	JRE	Entorno de ejecución en tiempo real
Standard Edition	SE	Plataforma Java para utilizar en escritorios (<i>desktop</i>) y aplicaciones de servidores
Enterprise Edition	EE	Plataforma Java para aplicaciones complejas de servidores
Micro Edition	ME Embedded	Plataforma Java para utilizar en entornos embebidos- tarjetas SIM e inteligentes, teléfonos móviles (celulares), y otros pequeños dispositivos

A.3 Entornos de desarrollo integrados (Java, C y C++)

El sistema más fácil y profesional, hoy día, es un entorno de desarrollo que contiene un editor de texto integrado y menús para compilar y lanzar (ejecutar) un programa junto con un depurador integrado.

Entornos de desarrollo integrados más populares	
BlueJ	(www.blueJ.org)
NetBeans	(www.netbeans.org)
JBuilder	(www.borland.com)
Eclipse	(www.eclipse.org)
JCreator	(www.jcreator.com)
JEdit	(www.jedit.org)
JGrasp	(www.jgrasp.org)
IntelliJIDEA	(www.jetbrains.com/idea/)

Nota: Un EDI contiene un editor para crear o editar el programa, un compilador para traducir el programa y verificar errores de sintaxis, un programa cargador que carga los códigos objetos de los recursos de las bibliotecas utilizadas y un programa para ejecutar el programa específico.

Aunque el JDK es un buen entorno para programación práctica, en la última década han proliferado los entornos de desarrollos integrados y profesionales que se han convertido en herramientas muy potentes y fáciles de utilizar, y están sustituyendo a las herramientas tradicionales (editor, compilador y depurador) o el kit de desarrollo Java, JDK.

Existen numerosos entornos de desarrollo profesionales tanto gratuitos como de pago, aunque las opciones gratuitas son muy variadas y con muy buen rendimiento y prestaciones.

Herramientas de desarrollo

Antes de que comience a practicar y desarrollar programas en su computadora, se debe disponer del software adecuado en la misma de modo que se pueda utilizar para editar, compilar y ejecutar programas en **Java**. El software más adecuado que se puede descargar gratuitamente del sitio de Oracle, es la versión **8** o la versión **7u51** (en el mes de abril de 2014).

Existen diferentes versiones populares de entornos integrados de desarrollo (Integrated Development Environment, IDE) de soporte para Java: **Borland JBuilder**, **IntelliJIDEA**, **Eclipse**, **BlueJ** (uno de los más idóneos para principiantes de programación y NetBeans, para entornos y programadores profesionales). También debe considerar la herramienta de desarrollo Java ya estudiada y más sencilla, JDK, que podrá utilizar independientemente o en unión de los entornos anteriores. El JDK (Java Development Kit) es gratuito y se puede descargar del sitio web de Oracle (www.oracle.com). Siempre que Oracle lanza una nueva versión de Java, también pone disponible en la web, de modo gratuito, el Kit que soporta la versión. Las versiones actuales son **JDK 7**, y **JDK 8**. Normalmente cuando hablamos del lenguaje, emplearemos el nombre de Java y cuando hablamos del Kit utilizaremos JDK.

Las plataformas de Java 7 y 8, Platform Standard Edition (Java SE 7 y Java SE 8) se puede descargar gratis de Internet en [ww.oracle.com/technetwork/java/javase/overview/index.html](http://www.oracle.com/technetwork/java/javase/overview/index.html). También existen empresas reputadas que ofrecen dichas descargas gratuitas con facilidades complementarias, tales como la española *Softonic* o la estadounidense *Download* de Cnet.

NetBeans

NetBeans ([//netbeans.org](http://netbeans.org)) es un entorno de desarrollo integrado de código abierto para Java. Sun Microsystem creó el proyecto de código abierto (*open source*) en junio de 2000 y al día de hoy pertenece el proyecto a la comunidad NetBeans.

NetBeans es un entorno de desarrollo que permite escribir, compilar, depurar y ejecutar programas; está escrito en Java, pero puede servir para cualquier otro lenguaje de programación. Existen dos productos en la actualidad: NetBeans IDE, producto libre y gratuito sin restricciones de uso y NetBeans Platform que es una base modular y extensible usada como estructura de integración para creación de aplicaciones de escritorio.

La gran ventaja de NetBeans es su comunidad con gran número de empresas independientes asociadas, especializadas en desarrollo de software que proporcionan extensiones adicionales que se integran fácilmente en la plataforma y que pueden también utilizarse para desarrollar sus propias herramientas y soluciones. Los dos productos de NetBeans son de código abierto y gratuitos para uso tanto comercial como no comercial y cuyo código fuente está disponible para su reutilización en el sitio web de NetBeans. La versión actual de NetBeans es **NetBeans IDE 8.0** que se ha lanzado de modo simultáneo a la versión 8 de Java.

Eclipse

Eclipse es un EDI de código abierto multiplataforma muy utilizado y popular. Eclipse es una herramienta gratuita disponible en <http://eclipse.org> y está escrito en Java.

Eclipse fue creado inicialmente por IBM como sucesor de la popular familia Visual Age. Hoy Eclipse es desarrollado por la Fundación Eclipse, una organización sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios y servicios que ofrecen herramientas para numerosos lenguajes de programación como C, C++, PHP, etc., además de Java.

Existen dos productos de Eclipse para desarrollo Java: Eclipse IDE for Java EE Developers (para entornos profesionales y de empresas) y Eclipse IDE for Java Developers (para entornos de programación SE de Java).

BlueJ

BlueJ es un EDI que tiene un editor integrado, un compilador, una máquina virtual y un depurador para escritura de programas. También tiene una presentación gráfica para estructuras de clases, soporta edición de texto y gráficos, permite la creación de objetos interactivos, pruebas interactivas y construcción de aplicaciones incrementales.

BlueJ es un entorno de desarrollo muy popular en ambientes académicos debido a que ha sido creado y está soportado por la University of Kent y La Trobe University.

Otros entornos de desarrollo

Existen otros entornos de desarrollo también muy utilizados: Dr.Java (drjava.sourceforge.net), JCreator (www.jcreator.com), es un entorno profesional, propietario y de pago; JBuilder (de Borland, empresa muy prestigiosa de desarrollo de herramientas de software), Code Warrior (Metrowerks) y JGresp (Auburn University).

A.4 Compilación sin entornos de desarrollo: Java y C/C++

Es posible también ejecutar programas utilizando un editor de archivos de texto, como Notepad de Windows o el clásico editor Edit del sistema operativo MS_DOS. Una vez editado el código fuente el programa se compila con el compilador javac del JDK.

Compiladores de Java

El compilador javac lee un archivo fuente .java y crea uno o más archivos .class que pueden ser ejecutados por un intérprete Java. Así si el archivo fuente se llama Demo.java, la compilación del programa se reali-

za escribiendo la orden (instrucción) `javac Demo.java`. Si el compilador no visualiza ningún mensaje de error es que se ha compilado con éxito. Si hay problemas, el compilador informará de cada error. Si el programa se ha compilado sin error, se crea un archivo denominado `Demo.class` en la misma carpeta que contenía `Demo.java`. El archivo de extensión `.class` contiene el bytecode de Java que podrá ser ejecutado por un intérprete Java. El intérprete se llama `java` y se ejecuta también desde la línea de órdenes de la pantalla, con la orden `java Demo` y al ejecutarse, se realizarán las tareas previstas en el código fuente.

Compiladores de C++ (versión C++11 y futura C++14)

El sitio oficial de la Standard C++ Foundation (www.isocpp.org), una fundación muy importante para el “planeta” C++ y uno de cuyos directores es **Bjarne Stroustrup** y una selección de empresas miembros de la Fundación son: Microsoft, Google, Intel, ARM, Oracle, Citrix, IBM... recomienda compiladores gratuitos para C++:

- GCC (GNU Compiler Collection)
- Clang
- Microsoft Virtual C++ 2012
- Intel C++ Compiler
- Oracle Solaris Studio ++ Compiler

La Standard C++ Foundation contiene una página con información sobresaliente de compiladores de C++ y otras herramientas:

- <https://isocpp.org/wiki/faq/compiler-dependencies#free-cpp-compiler>

y dos sitios web donde se recomiendan numerosos compiladores además de los citados anteriormente

- www.compilers.net/Dir/Free/Compilers/CCpp.htm
- www.idiom.com/free-compilers/LANG/C++-1.html

APÉNDICE B



Representación de la información en las computadoras

Una computadora es un sistema para procesar información de modo automático. Un tema vital en el proceso de funcionamiento de una computadora es estudiar la forma de representación de la información en dicha computadora. Es necesario considerar cómo se puede codificar la información en patrones de bits que sean fácilmente almacenables y procesables por los elementos internos de la computadora.

Las formas de información más significativas son: textos, sonidos, imágenes y valores numéricos, y cada una de ellas presenta peculiaridades distintas. Otros temas importantes en el campo de la programación se refieren a los métodos de detección de errores que se puedan producir en la transmisión o almacenamiento de la información y a las técnicas y mecanismos de comprensión de información con el objeto de que esta ocupe el menor espacio en los dispositivos de almacenamiento y sea más rápida su transmisión.

Representación de textos

La información en formato de texto se representa mediante un código en el que cada uno de los distintos símbolos del texto (como letras del alfabeto o signos de puntuación) se asignan a un único patrón de bits. El texto se representa como una cadena larga de bits en la cual los sucesivos patrones representan los sucesivos símbolos del texto original.

En resumen, se puede representar cualquier información escrita (texto) mediante caracteres. Los caracteres que se utilizan en computación suelen agruparse en cinco categorías:

1. **Caracteres alfabéticos** (letras mayúsculas y minúsculas, en una primera versión del abecedario inglés).

A, B, C, D, E, ... X, Y, Z, a, b, c, ... , x, y, z

2. **Caracteres numéricos** (dígitos del sistema de numeración).

0, 1, 2, 3, 4, 5, 6, 7, 8, 9 sistema decimal

3. **Caracteres especiales** (símbolos ortográficos y matemáticos no incluidos en los grupos anteriores).

{ } Ñ ñ ! ? & > # ¢ ...

4. **Caracteres geométricos y gráficos** (símbolos o módulos con los cuales se pueden representar cuadros, figuras geométricas, íconos, etcétera).

|-| ||- -▲ ...

5. **Caracteres de control** (representan órdenes de control como el carácter para pasar a la siguiente línea [NL] o para ir al comienzo de una línea [RC, retorno de carro, *carriage return*, CR], emitir un pitido [BEL], etcétera).

Al introducir un texto en una computadora, a través de un periférico, los caracteres se codifican según un **código de entrada/salida** de modo que a cada carácter se le asocia una determinada combinación de n bits.

Los códigos más utilizados en la actualidad son: **EBCDIC, ASCII y Unicode**:

- **Código EBCDIC** (*Extended Binary Coded Decimal Inter Change Code*). Este código utiliza $n = 8$ bits de forma que se puede codificar hasta $m = 2^8 = 256$ símbolos diferentes. Este fue el primer código utilizado para computadoras, aceptado en principio por IBM.
- **Código ASCII** (*American Standard Code for Information Interchange*). El código ASCII básico utiliza 7 bits y permite representar 128 caracteres (letras mayúsculas y minúsculas del alfabeto inglés, símbolos de puntuación, dígitos 0 a 9 y ciertos controles de información como retorno de carro, salto de línea, tabulaciones, etc.). Este código es el más utilizado en computadoras, aunque el ASCII ampliado con 8 bits permite llegar a 2^8 (256) caracteres distintos, entre ellos ya símbolos y caracteres especiales de otros idiomas como el español.
- **Código Unicode**. Aunque ASCII ha sido y es dominante en la representación de los caracteres, hoy día se requiere la información en muchas otras lenguas, como portugués, español, chino, japonés, árabe, catalán, e. quela, gallego, etc. Este código utiliza un patrón único de 16 bits para representar cada símbolo, que permite 2^{16} bits o sea hasta 65 536 patrones de bits (símbolos) diferentes.

Desde el punto de vista de unidad de almacenamiento de caracteres, se utiliza el archivo (**fichero**). Un **archivo** consta de una secuencia de símbolos de una determinada longitud codificados utilizando ASCII o Unicode y que se denomina **archivo de texto**. Es importante diferenciar entre archivos de texto simples que son manipulados por los programas de utilidad denominados **editores de texto** y los archivos de texto más elaborados que producen los procesadores de texto, tipo Microsoft Word. Ambos constan de caracteres de texto, pero mientras el obtenido con el editor de texto es un archivo de texto puro que codifica carácter a carácter, el archivo de texto producido por un procesador de textos contiene números, códigos que representan cambios de formato, de tipos de fuentes de letra y otros, e incluso pueden utilizar códigos propietarios distintos de ASCII o Unicode.

Representación de valores numéricos

El almacenamiento de información como caracteres codificados es ineficiente cuando la información se registra como numérica pura. Veamos esta situación con la codificación del número 65; si se almacena como caracteres ASCII utilizando un byte por símbolo, se necesita un total de 16 bits, de modo que el número mayor que se podría almacenar en 16 bits (dos bytes) sería 99. Sin embargo, si utilizamos *notación binaria* para almacenar enteros, el rango puede ir de 0 a 65 535 ($2^{16} - 1$) para números de 16 bits. Por consiguiente, la notación binaria (o variantes de ellas) es la más utilizada para el almacenamiento de datos numéricos codificados.

La solución que se adopta para la representación de datos numéricos es la siguiente: al introducir un número en la computadora se codifica y se almacena como un texto o cadena de caracteres, pero dentro del programa a cada dato se le envía un tipo de dato específico y es tarea del programador asociar cada dato al tipo adecuado correspondiente a las tareas y operaciones que se vayan a realizar con dicho dato. El método práctico realizado por la computadora es que una vez definidos los datos numéricos de un programa, una rutina (función interna) de la biblioteca del compilador (traductor) del lenguaje de programación se encarga de transformar la cadena de caracteres que representa el número en su notación binaria.

Existen dos formas de representar los datos numéricos: números enteros o números reales.

Representación de enteros

Los datos de tipo entero se representan en el interior de la computadora en notación binaria. La memoria ocupada por los tipos enteros depende del sistema, pero normalmente son dos bytes (en las versiones de MS-DOS y versiones antiguas de Windows, y cuatro bytes, en los sistemas de 32 bits como Windows o Linux). Por ejemplo, un entero almacenado en 2 bytes (16 bits):

1000 1110 0101 1011

Los enteros se pueden representar con signo (signed, en C++) o sin signo (unsigned, en C++); es decir, números positivos o negativos. Normalmente se utiliza un bit para el signo. Los enteros sin signo pueden contener valores positivos más grandes. Normalmente, si un entero no se especifica “con/sin signo” se suele asignar con signo por defecto u omisión.

El rango de posibles valores de enteros depende del tamaño en bytes ocupado por los números y si se representan con signo o sin signo (la tabla B.1 resume características de tipos estándar en C++).

Representación de números reales

Los números reales son aquellos que contienen una parte decimal como 2,6 y 3,14152. Los reales se representan en *notación científica* o en *coma flotante*; por esta razón en los lenguajes de programación, como C++, se conocen como números en coma flotante.

Existen dos formas de representar los números reales. La primera se utiliza con la notación del punto decimal.

Ejemplos

12.35 99901.32 0.00025 9.0

La segunda forma para representar números en coma flotante es la notación científica o exponencial, conocida también como notación E, es muy útil para representar números muy grandes o muy pequeños.

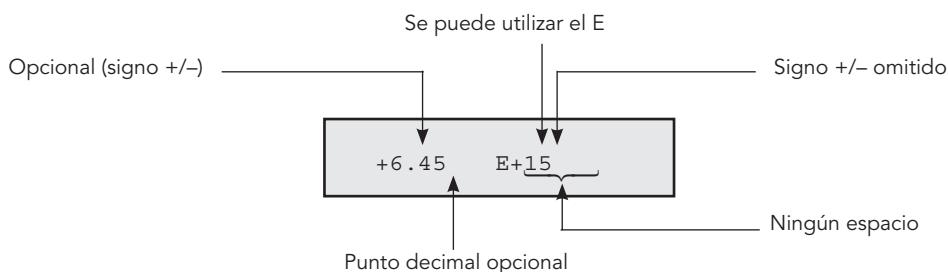
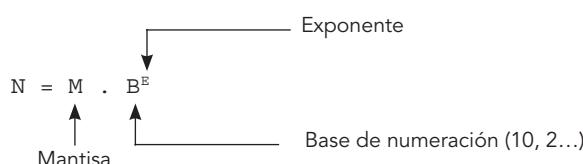


Tabla B.1 Tipos enteros reales, en C++.

Carácter y bool		
Tipo	Tamaño	Rango
short (short int)	2 bytes	-32.738..32.767
int	4 bytes	-2.147.483.648 a 2.147.483.647
long (long int)	4 bytes	-2.147.483.648 a 2.147.483.647
float (real)	4 bytes	10^{-38} a 10^{38} (aproximadamente)
double	8 bytes	10^{-308} a 10^{308} (aproximadamente)
long double	10 bytes	10^{-4932} a 10^{4932} (aproximadamente)
char (carácter)	1 byte	Todos los caracteres ASCII
bool	1 byte	True (verdadero) y false (falso)

Notación exponencial



Ejemplos

Representación de caracteres

Un documento de texto se escribe utilizando un conjunto de caracteres adecuado al tipo de documento. Como ya se ha comentado anteriormente, en los lenguajes de programación se utilizan, principalmente, dos códigos de caracteres. El más común es **ASCII** (American Standard Code for Information Interchange) y algunos lenguajes, como Java, utilizan **Unicode** (www.unicode.org). Ambos códigos se basan en la asignación de un código numérico a cada uno de los tipos de caracteres del código.

En C++, los *caracteres* se procesan normalmente usando el tipo `char`, que asocia cada carácter a un código numérico que se almacena en un byte.

El código ASCII básico que utiliza 7 bits (128 caracteres distintos) y el ASCII ampliado a 8 bits (256 caracteres distintos) son los códigos más utilizados. Así se pueden representar caracteres como 'A', 'B', 'c', '\$', '!', '5', etc. La tabla B.1 recoge los tipos enteros, reales y carácter utilizados en C++, la memoria utilizada (número de bytes ocupados por el dato) y el rango de números.

Representación de imágenes

Las imágenes se adquieren mediante periféricos especializados como escáneres, cámaras digitales de video, cámaras fotográficas, etc. Una imagen, al igual que otros tipos de información, se representa por patrones de bits, generados por el periférico correspondiente. Existen dos métodos básicos para representar imágenes: *mapas de bits* y *mapas de vectores*.

En las técnicas de *mapas de bits*, una imagen se considera como una colección de puntos, cada uno de los cuales se llama *pixel* (abreviatura de “*picture element*”). Una imagen en blanco y negro se representa como una cadena larga de bits que representan las filas de píxeles en la imagen, donde cada bit es bien 1 o bien 0, dependiendo de que el pixel correspondiente sea blanco o negro. En el caso de imágenes en color, cada pixel se representa por una combinación de bits que indican el color de los píxeles. Cuando se utilizan técnicas de mapas de bits, el patrón de bits resultante se llama *mapa de bits*, significando que el patrón de bits resultante que representa la imagen es poco más que un mapa de la imagen.

Muchos de los periféricos de computadora, como cámaras de video, escáneres, etc., convierten imágenes de color en formato de mapa de bits. Los formatos más utilizados en la representación de imágenes se muestran en la tabla B.2.

Mapas de vectores. Otros métodos para representar una imagen se fundamentan en descomponer la imagen en una colección de objetos como líneas, polígonos y textos con sus respectivos atributos o detalles (grosor, color, etcétera).

Tabla B.2 Mapas de bits.

Formato	Formato, origen y descripción
BMP	Microsoft. Formato sencillo con imágenes de gran calidad pero con el inconveniente de ocupar mucho espacio (no útil para la web).
JPEG	Grupo JPEG . Calidad aceptable para imágenes naturales. Incluye compresión. Se utiliza en la web.
GIF	CompuServe. Muy adecuado para imágenes no naturales (logotipos, banderas, dibujos anidados...). Muy usado en la web.

Tabla B.3 Mapas de vectores.

Formato	Descripción
IGES	ASME/ANSI. Estándar para intercambio de datos y modelos de (AutoCAD...).
Pict	Apple Computer. Imágenes vectoriales.
EPS	Adobe Computer.
TrueType	Apple y Microsoft para EPS.

Representación de sonidos

La representación de sonidos ha adquirido una importancia notable debido esencialmente a la infinidad de aplicaciones multimedia tanto autónomas como en la web.

El método más genérico de codificación de la información de audio para almacenamiento y manipulación en computadora es mostrar la amplitud de la onda de sonido en intervalos regulares y registrar las series de valores obtenidos. La señal de sonido se capta mediante micrófonos o dispositivos similares y produce una señal analógica que puede tomar cualquier valor dentro de un intervalo continuo determinado. En un intervalo de tiempo continuo se dispone de infinitos valores de la señal analógica, que es necesario almacenar y procesar, para lo cual se recurre a una *técnica de muestreo*. Las muestras obtenidas se digitalizan con un conversor analógico-digital, de modo que la señal de sonido se representa por secuencias de bits (por ejemplo, 8 o 16) para cada muestra. Esta técnica es similar a la utilizada, de manera histórica, por las comunicaciones telefónicas a larga distancia. Naturalmente, dependiendo de la calidad de sonido que se requiera, se necesitarán más números de bits por muestra, frecuencias de muestreo más altas y lógicamente más muestras por períodos.

Como datos de referencia se puede considerar que para obtener reproducción de calidad de sonido de alta fidelidad para un disco CD de música, se suele utilizar, al menos, una frecuencia de muestreo de 44 000 muestras por segundo.

Los datos obtenidos en cada muestra se codifican en 16 bits (32 bits para grabaciones en estéreo). Cada segundo de música grabada en estéreo requiere más de un millón de bits.

Un sistema de codificación de música muy extendido en sintetizadores musicales es MIDI (*Musical Instruments Digital Interface*) que se encuentra en sintetizadores de música para sonidos de videojuegos, sitios web, teclados electrónicos, etcétera.

APÉNDICE C



Códigos ASCII y UNICODE

C.1 Código ASCII

El código ASCII (*American Standard Code for Information Interchange*: código estándar americano para intercambio de información) es un código que traduce caracteres alfabéticos y caracteres numéricos, así como símbolos e instrucciones de control en un código binario de siete u ocho bits.

Tabla C.1 Código ASCII de la computadora personal PC.

Valor ASCII	Carácter	Valor ASCII	Carácter
0	Nulo	30	Cursor arriba
1	☺	31	Cursor abajo
2	☻	32	Espacio
3	♥	33	!
4	♦	34	"
5	♣	35	#
6	♠	36	\$
7	Sonido (pitido, bip)	37	%
8	☺	38	&
9	Tabulación	39	,
10	Avance de línea	40	(
11	Cursor a inicio	41)
12	Avance de página	42	*
13	Retorno de carro	43	+
14		44	.
15	☼	45	-
16	►	46	.
17	◀	47	/
18	↕	48	0

(continúa)

Tabla C.1 Código ASCII de la computadora personal PC (continuación).

Valor ASCII	Carácter	Valor ASCII	Carácter
19	!!	49	1
20	π	50	2
21	§	51	3
22	-	52	4
23	‡	53	5
24	↑	54	6
25	↓	55	7
26	→	56	8
27	←	57	9
28	Cursor a la derecha	58	:
29	Cursor a la izquierda	59	;
60	<	102	ƒ
61	=	103	g
62	>	104	h
63	?	105	i
64	@	106	j
65	A	107	k
66	B	108	l
67	C	109	m
68	D	110	n
69	E	111	o
70	F	112	p
71	G	113	q
72	H	114	r
73	I	115	s
74	J	116	t
75	K	117	u
76	L	118	v
77	M	119	w
78	N	120	x
79	O	121	y
80	P	122	z
81	Q	123	{
82	R	124	
83	S	125	}
84	T	126	~
85	U	127	△
86	V	128	Q
87	W	129	ü
88	X	130	é

(continúa)

Tabla C.1 Código ASCII de la computadora personal PC (*continuación*).

Valor ASCII	Carácter	Valor ASCII	Carácter
89	Ý	131	â
90	Ž	132	ä
91	[133	à
92	\	134	å
93]	135	ç
94	^	136	ê
95	-	137	ë
96	'	138	è
97	a	139	ï
98	b	140	î
99	c	141	ã
100	d	142	Ä
101	e	143	Å
144	É	186	
145	æ	187	
146	Æ	188	
147	ô	189	
148	ö	190	
149	ò	191	-
150	û	192	¬
151	ù	193	¬
152	ÿ	194	¬
153	Ö	195	¬
154	Ü	196	-
155	¢	197	+
156	£	198	£
157	¥	199	₪
158	₱	200	₱
159	ƒ	201	ර
160	á	202	ළ
161	í	203	ශ
162	ó	204	්
163	ú	205	=
164	ñ	206	්
165	Ñ	207	්
166	a	208	්
167	o	209	්
168	ɛ	210	්
169	ѓ	211	්
170	ѓ	212	්

(continúa)

Tabla C.1 Código ASCII de la computadora personal PC (continuación).

Valor ASCII	Carácter	Valor ASCII	Carácter
171	½	213	ƒ
172	¼	214	∏
173	ı	215	‡
174	«	216	†
175	»	217	„
176	⋮⋮	218	Γ
177	⊗	219	■
178	✖	220	■
179	—	221	—
180	—	222	—
181	—	223	—
182	—	224	α
183	π	225	β
184	ℓ	226	Γ
185	—	227	∏
228	Σ	242	≥
229	σ	243	≤
230	μ	244	∫
231	τ	245	∫
232	φ	246	÷
233	Θ	247	≈
234	Ω	248	◦
235	δ	249	•
236	∞	250	·
237	∅	251	√
238	ε	252	ₙ
239	∩	253	₂
240	≡	254	■
241	±	255	(blanco 'FF')

C.2 Código Unicode

Existen numerosos sistemas de codificación que asignan un número a cada carácter (letras, números, signos...). Ninguna codificación (el código ASCII es un ejemplo elocuente) específica puede contener caracteres suficientes. Por ejemplo, la Unión Europea, por sí sola, necesita varios sistemas de codificación distintos para cubrir todos sus idiomas. También presentan problemas de incompatibilidad entre los diferentes sistemas de codificación. Por esta razón se creó Unicode.

El consorcio Unicode es una organización sin ánimo de lucro que se creó para desarrollar, difundir y promover el uso de la norma Unicode que especifica la representación del texto en productos y estándares de software modernos. El consorcio está integrado por una amplia gama de corporaciones y organizaciones de la industria de la computación y del procesamiento de la información (como Apple, HP, IBM, Sun, Oracle, Microsoft,... o estándares modernos como XML, Java, CORBA, etcétera).

Formalmente, el estándar Unicode está definido en la última versión impresa del libro *The Unicode Standard* que edita el consorcio y que también se puede “bajar” de su sitio web.

En el momento de escribir este apéndice, la versión estándar ofrecida por el consorcio era la versión 6.1.0: 2012, que se puede descargar de la red en las direcciones que aparecen abajo. La última versión en beta y revisión es la versión 7.0.0 presentada en febrero de 2014 y cuyo lanzamiento está previsto para junio de 2014.

Unicode está llamado a reemplazar al código ASCII y algunos de los códigos restantes más populares, como Latin-1, en unos pocos años y a todos los niveles. Permite no solo manejar texto en prácticamente cualquier lenguaje utilizado en el planeta, sino que también proporciona un conjunto completo y comprensible de símbolos matemáticos y técnicos que simplificará el intercambio de información científica.

Recomendamos al lector que visite los sitios web que incluimos en esta página para ampliar la información que necesite en sus tareas de programación actuales o futuras. El código sigue evolucionando y dada la masiva cantidad de información que incluye, el mejor consejo es visitar estas páginas u otras similares, y si ya se ha convertido en un experto programador y necesita el código para efectos profesionales, le recomendamos que descargue de la Red todo el código completo o adquiera en su defecto el libro que le indicamos a continuación que contiene toda la información oficial de Unicode.

Referencias web

Página oficial del consorcio Unicode.

www.unicode.org

Información de Unicode en español.

www.unicode.org/standard/translations/spanish.html

Unicode para lenguajes de programación.

www.unicode.org/resources/programs.html

Recursos Unicode.

www.unicode.org/resources/index.html

unicode 6.0.0 (publicado en línea, el código)

www.unicode.org/versions/unicode_6.0.0

unicode 6.1.0 (especificaciones)

www.unicode.org/unicode_6.1.0

Unicode 7.0.0 Beta (presentado en febrero de 2014)

www.unicode.org/version/unicode_6.1.0

Bibliografía

The Unicode Consortium: The Unicode Standard, Versión 3.0. Reading, MA, Addison-Wesley, 2000.

APÉNDICE D



Palabras reservadas de Java 5 a 8, C y C++11

D.1 Java

La tabla D.1 contiene las palabras clave (reservadas) de Java. Dos de ellas son reservadas pero no se utilizan por el lenguaje Java: `const` y `goto`. Estas palabras pertenecen a otros lenguajes de programación como C y C++ y se incluyen en Java con el objeto de generar mensajes de error mejores si se utilizan en un programa Java. Java 5 introdujo una nueva palabra clave: `enum`.

Tabla D.1 Palabras clave (*keywords*) de Java.

<code>abstract</code>	<code>double</code>	<code>int</code>	<code>super</code>
<code>assert</code>	<code>else</code>	<code>interface</code>	<code>switch</code>
<code>boolean</code>	<code>enum</code>	<code>long</code>	<code>synchronized</code>
<code>break</code>	<code>extends</code>	<code>native</code>	<code>this</code>
<code>byte</code>	<code>final</code>	<code>new</code>	<code>throw</code>
<code>case</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>catch</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>char</code>	<code>for</code>	<code>protected</code>	<code>try</code>
<code>class</code>	<code>goto</code>	<code>public</code>	<code>void</code>
<code>const</code>	<code>if</code>	<code>return</code>	<code>volatile</code>
<code>continue</code>	<code>implements</code>	<code>short</code>	<code>while</code>
<code>default</code>	<code>import</code>	<code>static</code>	
<code>do</code>	<code>instanceof</code>	<code>strictfp</code>	

Consejos

1. *Las palabras clave no se pueden utilizar como identificadores en un programa Java.*
2. `true`, `false` y `null`, literales, se confunden, a veces, con palabras clave. Estas palabras no son claves, sino literales reservados.
3. Las palabras `const` y `goto` no se utilizan actualmente en el lenguaje Java.

Palabras reservadas de Java con significado especial

Algunas palabras tienen un significado especial en Java. Las más utilizadas en la creación de programas se resumen en la tabla D.2.

Tabla D.2 Palabras reservadas especiales.

Tipo C	Rango de valores
class	Indica el comienzo de la definición de una clase.
double	Indica que las variables enumeradas a continuación son números de coma flotante de doble precisión.
extends	Indica que una clase (subclase) se extiende de otras clases (clase base o superclase).
new	Crea una nueva instancia de un objeto.
private	La accesibilidad está restringida.
return	Devuelve el resultado que sigue al método llamador.
static	Hay exactamente un método o campo de datos de su identificador de la clase.
void	No devuelve ningún valor.

Palabras reservadas (*keywords*) de C/C++

Las palabras clave o reservadas (*keywords*) no se pueden utilizar para otros propósitos, como nombres de variables. La tabla D.3 muestra las palabras reservadas de C y C++. Las palabras en negrita son palabras reservadas de ANSI C 99. La palabra reservada `auto` tiene un nuevo significado en C++11.

Tabla D.3 Palabras reservadas de C y C++.

asm	delete	goto	return	typedef
auto	do	if	short	typeid
bad cast	double	inline	signed	typename
bad typeid	dynamic cast	int	sizeof	union
bool	else	long	static	unsigned
break	enum	mutable	static_cast	using
case	except	namespace	struct	virtual
catch	explicit	new	switch	void
char	extern	operador	template	volatile
class	false	private	this	while
const	finally	protected	throw	xor
const cast	float	public	true	xor ed
continue	for	register	try	
default	friend	reinterpret_ cast	type info	

Nota: En la página web

http://en.cppreference.com/w/Main_Page

puede encontrar el lector un excelente sitio con guías de referencia de sintaxis, así como bibliotecas de funciones y de clases de las diferentes versiones estándares de C++ (C++98, C++03, C++11) y de ANSI C (C89, C99, C11).

Palabras nuevas de C++11

```
alignas
alignof
char16_t
char32_t
constexpr
decltype
noexcept
nullptr
static_assert
thread_local
```

APÉNDICE E



Prioridad de operadores C/C++ y Java

Los operadores se muestran en orden decreciente de prioridad de arriba a abajo. Los operadores del mismo grupo tienen la misma prioridad (precedencia) y se ejecutan de izquierda a derecha o de derecha a izquierda según asociatividad. El número que precede al operador es el orden de prioridad.

Operador	Tipo	Asociatividad
1	()	Paréntesis.
1	()	Llamada a función.
1	[]	Subíndice.
1	.	Acceso a miembros de un objeto.
2	++	Prefijo incremento.
2	--	Prefijo decremento.
2	+	Más unitario.
2	-	Menos unitario.
2	!	Negación lógica unitaria.
2	~	Complemento bit a bit unitario.
2	(tipo)	Modelado unitario.
2	new	Creación de objetos.
3	*	Producto.
3	/	División.
3	%	Resto entero.
4	+	Suma.
4	-	Resta.
5	<<	Desplazamiento bit a bit a la izquierda.
5	>>	Desplazamiento bit a bit a la derecha con extensión de signo.
5	>>>	Desplazamiento bit a bit a la derecha rellenando con ceros.

Operador		Tipo	Asociatividad
6	<	Menor que.	Izquierda-Derecha
6	<=	Menor o igual que.	Izquierda-Derecha
6	>	Mayor que.	Izquierda-Derecha
6	>=	Mayor o igual que.	Izquierda-Derecha
6	instanceof	Verificación tipo de objeto.	Izquierda-Derecha
7	==	Igualdad.	Izquierda-Derecha
7	!=	Desigualdad.	Izquierda-Derecha
8	&	AND bit a bit.	Izquierda-Derecha
9	^	OR exclusive bit a bit.	Izquierda-Derecha
10		OR inclusive bit a bit.	Izquierda-Derecha
11	&&	AND lógico.	Izquierda-Derecha
12		OR lógico.	Izquierda-Derecha
13	? :	Condicional ternario.	Derecha-Izquierda
14	=	Asignación.	Derecha-Izquierda
14	+=	Asignación de suma.	Derecha-Izquierda
14	-=	Asignación de resta.	Derecha-Izquierda
14	*=	Asignación de producto.	Derecha-Izquierda
14	/=	Asignación de división.	Derecha-Izquierda
14	%=	Asignación de módulo.	Derecha-Izquierda
14	&=	Asignación AND bit a bit.	Derecha-Izquierda
14	^=	Asignación OR exclusive bit a bit.	Derecha-Izquierda
14	=	Asignación or inclusive bit a bit.	Derecha-Izquierda
14	<<=	Asignación de desplazamiento a izquierda bit a bit.	Derecha-Izquierda
14	>>=	Desplazamiento derecho bit a bit con asignación de extensión de signo.	Derecha-Izquierda
14	>>>=	Desplazamiento derecho bit a bit con asignación de extensión a cero.	Derecha-Izquierda

Índice analítico

- A**
- Accesibilidad y visibilidad en la herencia, 433
 - Acceso a elementos mediante bucles, 231
 - Acceso a los elementos
 - de los arreglos bidimensionales, 230
 - de un arreglo, 622
 - Acceso a miembros de la clase:
 - encapsulamiento, 401
 - Acceso a una estructura de datos mediante
 - el operador apuntador, 286
 - el operador punto, 285
 - Advertencias (*warning*), 80
 - Agotamiento de la entrada, 167
 - Agregación, 421
 - Algoritmo
 - de la burbuja, 262
 - de ordenación por inserción, 266
 - de selección, 266
 - y codificación de la búsqueda binaria, 273
 - Algoritmo: concepto y propiedades, 42
 - Algunos componentes de la jerarquía de colecciones, 691
 - Almacén libre (*free store*), 322
 - Almacenamiento
 - de información en estructuras, 285
 - en memoria de los arreglos, 224
 - Ámbito
 - (alcance) de una variable, 194
 - de archivo fuente, 195
 - de bloque, 195
 - de programa, 194
 - de una función, 195
 - Ánalisis
 - de la búsqueda binaria, 275
 - del algoritmo de la burbuja, 264
 - del algoritmo *quicksort*, 272
 - del problema, 35
 - ANSI, 337
 - Apertura de un archivo, 339
 - Aplicación
 - del tipo abstracto de dato conjunto, 658
 - de apuntadores: conversión de caracteres, 312
 - Aplicaciones prácticas de manejo de excepciones, 589, 591, 723
 - Applets*, 626
 - Apuntador a *File*, 338
 - Apuntadores
 - a apuntadores, 307
 - a cadenas, 310
 - a constantes, 314
 - a estructuras, 321
 - a funciones, 317
 - como argumentos de funciones, 316
 - constants, 313
 - constants a constantes, 315
 - constants frente a apuntadores a constantes, 313
- versus* arreglos, 310
- y arreglos, 308
- y verificación de tipos, 305
- Archivo, 748
 - de texto, 748
 - fuente, 745
- Archivos
 - de cabecera, 83, 447
 - de cabecera y de clases, 492
- Argumentos de la función: paso por valor y por referencia, 468
- Argumentos de plantilla, 564
- Aritmética de apuntadores, 311
- Arrays* (arreglos), 452
- Arreglo, 222
 - de más de dos dimensiones, 232
 - almacenado en memoria, 308
 - de 10 apuntadores a enteros, 310
- Arreglos (*arrays*), 621
 - como miembros, 292
 - de apuntadores, 309
 - irregulares o triangulares, 624
 - listas y tablas. Cadenas, 221
 - multidimensionales, 623
 - y objetos, 622
- Asignación
 - booleana, 114
 - de un elemento, 694
 - dinámica de la memoria, 322
- Asociación, 411
 - cualificada, 415
 - entre clases, 412
- Asociaciones
 - cualificadas, 418
 - reflexivas, 415, 419
 - ternarias, 417
- Asociatividad, 107, 119
- Atributos con valores por defecto (en forma predeterminada), 388
- B**
- Banderas de estado, 151
 - Biblioteca
 - del compilador, 748
 - estándar, 742
 - Big data*. Los grandes volúmenes de datos, 27
 - Bloques
 - de construcción (componentes)
 - de UML 2.5, 377
 - de sentencias, 615
 - try*, 581, 712
 - BlueJ*, 745
 - Bucle
 - (ciclo), 145
 - controlado por bandera-indicador, 151
 - for*, 154
 - for* de C, 154
- infinito, 147
- interno, 169
- pretest*, 146
- while*, 145
- Bucle (lazos), 618
 - anidados, 168
 - controlados por centinelas, 150
 - controlados por contador, 154
 - controlados por indicadores (banderas), 151
 - for* vacíos, 161, 167
 - infinitos, 160
 - while (true)*, 153
 - while* con cero iteraciones, 150
- Búsqueda
 - binaria, 273
 - binaria de un elemento, 274
 - de una clave, 694
 - en listas, 273
 - secuencial y binaria, 273
- Bytecode* de Java, 746
- C**
- Cadena, 238
 - Cadenas, 453, 607
 - inmutables, 607
 - Captura de una excepción: *catch*, 583, 715
 - Caracteres, 750
 - (char)*, 86
 - alfabéticos, 747
 - de control, 748
 - especiales, 747
 - geométricos y gráficos, 747
 - numéricos, 747
 - Características
 - de la herencia múltiple, 535
 - de los algoritmos, 43
 - de los objetos, 386
 - de un algoritmo, 35
 - más sobresalientes de la resolución de problemas, 35
 - típicas de E/S para archivos en C, 337
 - Categorías de modelado, 374
 - Ciclo de vida de un
 - Applet*, 629, 631, 633
 - hilo, 731
 - Cierre de archivos, 341
 - Clase, 395, 474
 - Arrays*, 692
 - Collections*, 695
 - concreta*, 440
 - Graphics*, 633
 - Object*, 655
 - Clases
 - abstractas, 426, 438, 678
 - compuestas, 499
 - contenedoras, 549
 - de almacenamiento, 196

- de asociación, 416
 de excepciones, 578
 de excepciones definidas en Java, 719
 de utilidades: *Arrays* y *Collections*, 692
 en UML, 396
 públicas, 645
 y los objetos, 473, 637
 y objetos, 473
Cláusula finally, 717
Cloud Computing (computación en la nube), 23
Code Warrior, 762
Codificación, 28
 de un programa, 39
 del algoritmo de inserción, 267
 del algoritmo de la burbuja, 263
 del algoritmo *quicksort*, 270
Código
 de entrada/salida, 748
 EBCDIC, 748
 ejecutable, 745
 fuente, 28, 745
 generado por una función fuera de línea, 192
 máquina, 28
 objeto, 745, 746
 Unicode, 750, 755
Código ASCII (American Standard Code for Information Interchange), 18, 748, 755, 756
Códigos de caracteres ASCII y Unicode, 611
Colección, 689
Colecciones, 689
 en Java, 689
 parametrizadas, 705
Comentarios, 74, 83
 en C++, 446
Comparable, 697
Comparación de
 bucles *while*, *for* y *do-while*, 165
 objetos: Comparable y Comparator, 697
Comparador, 697
Compilación, 745
 separada, 208
 y ejecución de un programa, 40
 y sus fases, 29
Compilador, 739
 javac, 745
Compiladores e intérpretes, 29
Complejidad de la búsqueda secuencial, 275
Componentes principales de una computadora, 8
Comprobación alfábética y de dígitos, 199
Computadora, 742
 personal, 4
Computadoras
 en perspectiva, 3
 modernas: una breve taxonomía, 5
 o dispositivos de mano (*handheld*), 5
 personales y estaciones de trabajo, 5
Concatenación de cadenas, 607
Concepto
 de apuntador (puntero), 301
 y uso de funciones de biblioteca, 198
Condición de terminación de la recursión, 213
Condiciones de error en programas, 574
Conjuntos, listas y mapas, 691
Consejo de programación, 153
Consideraciones de diseño, 524
Consola de línea de comandos, 62
Constantes, 88, 603
 caracteres, 89
 de cadena, 90
 declaradas *const* y *volatile*, 91
 definidas (símbólicas), 91
 enteras, 89
 enumeradas, 91
 literales, 88, 450
 reales, 89
Constructor, 493
 alternativo, 495
 de copia, 496
 por defecto, 494, 648
Constructores
 en herencia, 672
 sobrecargados, 495, 648
 inicializadores en herencia, 530
Conversión
 de tipos, 120, 458
 en expresiones, 459
 en pasos de argumentos, 459
 entre objetos y tipos básicos, 509
 explícita, 118, 459
 implícita, 118
Creación de
 macros con argumentos, 193
 un programa, 75
Criterios a seguir para elegir cómo crear un hilo, 730
Cuatro formas diferentes de representar una clase, 396
Cuerpo del bucle, 145
Cuerpo de un bucle while, 150
- D**
Dato miembro, 483
Datos
 globales, 59
 locales, 59
De C a C++, 444
Declaración, 93
 de apuntadores, 302, 451
 de métodos, 403
 de objetos de clases, 400, 401, 427, 429, 431
 de un tipo parametrizado, 705
 de una clase, 397, 638
 de una clase derivada, 432, 522
 de una estructura, 281
 de variables, 84
 de variables de cadena, 239
 o definición, 95
Declaraciones globales, 71
Definición
 de la clase, 475, 538
 de las funciones miembro, 565
 de plantilla de funciones, 553
 de una función, 467
 de una plantilla de clase, 560
 de un objeto, 385, 474
 de variables de estructuras, 282
 típica de una clase, 485
Dependencia, 410
Dependencias, generalizaciones-especializaciones y asociaciones, 410
Depuración, 739
 del programa, 739
Depurador (debugger), 80, 739
Desarrollo de
 programas web, 23
 software orientado a objetos con UML, 377
Desbordamiento aritmético, 752
Destructor, 498
Destructores, 532
Desventaja de const frente a #define, 92
Diagrama de
 objetos, 415
 secuencia, 392
Diagramas
 de clases, 383
 de flujo, 48
 de Nassi-Schneiderman (N-S), 57
 de UML 2.5, 374
 estructurados o estructurales, 375
Dibujar imágenes en un applet, 631
Diferencias
 entre *const* y *#define*, 92
 entre paso de variables por valor y por referencia, 189
 entre *while* y *do-while*, 164
Diferentes usos de bucles for, 158
Direcciones en memoria, 300
Directivas
 del preprocesador, 70
Directrices en la toma de decisión iteración/ recursión, 214
Diseño
 de algoritmos, 37, 44
 de bucles, 165
 de excepciones, 580, 711
 de formato libre, 600
 del problema, 36
 eficiente de bucles, 149
 orientado a objetos (OO), 61
 y representación gráfica de clases en UML, 394
Dispositivos
 de almacenamiento secundario, 12
 de comunicación, 13
 de entrada y salida, 12
Documentación interna, 40
Documento HTML para applet, 629
Dominación o prioridad, 536
Dr.Java, 745
- E**
Eclipse, 745
Editor
 compilador y depurador de errores, 61
 de textos, 61, 748

- Ejemplo de abstracción, 364
estructuras anidadas, 290
función plantilla, 557
objeto derivado, 676
- Ejemplos de herencia múltiple, 436
objetos, 385
- Elementos de agrupación, 379
de comportamiento, 378
de notación (notas), 379
de un programa en C, 81
estructurales, 377
- Encapsulamiento, 384, 483
- Ensamblador, 5
- Enteros (*int*), 84
- Entorno de desarrollo integrado (EDI), 63, 739, 740
- Entrada, 99, 613
de cadenas de caracteres, 100
de números y cadenas, 255
y salida, 97, 446, 612
- Enumeraciones, 170
Enumeration, 700
- Errores de sintaxis, 79, 80, 744
en tiempo de ejecución, 80
fatales, 80
lógicos, 79
- Escritura de algoritmos, 45
valores lógicos, 88
- Espacios de nombres, 447
- Especificación de excepciones, 586
una clase, 398
- Especificaciones de UML, 379
públicas, privadas y protegidas, 642
- Estado, comportamiento e identidad, 386
- Estados de un hilo, ciclo de vida de un hilo, 730
- Estructura de *break* y *continue*, 465
de un programa en C, 739
general de un programa en C, 68
- Estructuras de control, 461
de control iterativas o repetitivas, 145
o registros, 365
- Etapas de creación de un programa, 76
- Evolución de los lenguajes de programación, 31
un objeto, 389
una clase, 389
- Excepciones imprevistas, 588
- F**
- Fases del diseño de un algoritmo, 45
- Final de un bucle, 166
de control, 615, 617, 619
(*stream*), 337
- Formas de información, 763
- Formatos nemáticos (nemotécnicos), 31
- Función, 466
amiga, 500
atof, 252
main(), 72
miembro, 474
strlen(), 249
strncmp(), 251
- Funciones aleatorias, 205
de biblioteca, 466, 758
de carácter, 199
de conversión de caracteres, 201
de entrada/salida, 337
de entrada y de salida de archivos, 342
de prueba de caracteres especiales, 201
de utilidad, 206
definidas por el usuario, 72
en línea y fuera de línea, 489
logarítmicas y exponenciales, 204
matemáticas, 606
matemáticas de carácter general, 202
miembro de una clase, 487
numéricas, 202
recursivas, 210
strcat() y *strncat()*, 249
trigonométricas, 204
- G**
- Generaciones de computadoras, 3
- Generalización múltiple, 426
- Genericidad: plantillas (*templates*), 548
- Geolocalización y realidad aumentada, 26
- Grandes computadoras (*mainframes*), 5
- H**
- Hardware*, 7, 9, 11
- Herencia, 520
clases derivadas, 427, 429, 431
múltiple, 428, 435, 533
privada, 434, 528
protegida, 434, 528
pública, 434, 525
simple, 428, 533
- Herramientas de línea de órdenes, 740
de programación, 61, 63
gráficas y alfanuméricas, 37
- Hilos *daemon*, 734
de ejecución, 727
- Historia de UML, 379
del lenguaje Java: de Java 1.0 a Java 7, 596
- I**
- Identidad, 367, 391
- Identificadores, 82
- Implementación de clases, 491, 644
de una interface (interfaz), 683
del TAD Conjunto, 656
- Indirección de apuntadores, 304
- Información en texto, 747
- Inicialización de arreglos multidimensionales, 229
arreglos y arreglos anónimos, 623
la clase base, 537
un apuntador a una función, 318
un arreglo de apuntadores a cadenas, 310
una declaración de estructuras, 283
una variable en una declaración, 602
variables, 94, 602
variables de cadena, 240
- Instanciación, 480
de una plantilla de clases, 562
- Interfaces, 682, 683, 685
- Interfaz, 474, 638
- Internet de las cosas, 26
y la web, 21
- Iteración del bucle, 145
- Iteradores de una colección, 700
- Iterator*, 701
- J**
- JBuilder*, 744
- JCreator*, 745
- JDK, 742, 745
- Jerarquía de clases, 423, 424
colecciones *List*, 703
- Jerarquías de generalización/especialización, 425
- JGresp*, 745
- L**
- Lanzamiento de excepciones, 582, 714
- Las 11 propiedades sobresalientes (*buzzwords*) de Java, 598
- Lectura de información de una estructura, 286
y escritura de elementos de arreglos bidimensionales, 230
- Lenguaje C: elementos básicos, 34, 68, 176, 280, 444, 689
de modelado, 372
de programación Java, 597, 598
máquina, 4, 28
Smalltalk, 29
- Lenguaje Unificado de Modelado (UML, *Unified Model Language*), 372
- Lenguajes de alto nivel, 5
de programación, 27
de programación ensambladores y de alto nivel, 18
declarativos, 32
imperativos (procedimentales), 32
orientados a objetos, 33
- Ligadura, 539
dinámica, 539
dinámica frente a ligadura
estática, 544
dinámica mediante métodos abstractos, 679
estática, 539

Longitud y comparación de cadenas, 608
 Los Social Media, 22
 Llamada a una función, 181

M

Manejador de excepción, 576
 Manejo de excepciones en
 C++, 576
 Java, 708
 Mantenimiento y documentación, 42
 Mapa de bits, 750
 Mapas de vectores, 750
 Máquina Virtual Java, 740
 Máxima de Dijkstra, 753
 Mecanismo de manejo de excepciones,
 577, 709

Memoria
 central RAM (*Random Access Memory*),
 10

de acceso aleatorio, RAM, 10
 de la computadora, 9
 de sólo lectura, ROM, 10

Mensaje, 638
 de error, 80, 746
 Mensajes entre objetos, 390

Método

finalize, 650

Metodología de la programación, 42, 58

Métodos

abstractos, 677
 de la clase *String*, 609
 de una clase, 642
 que informan de la excepción, 721
static, 654

Miembros, 281

static de una clase, 652

Modelado e identificación de objetos,
 367

Modelo de

compilación de inclusión, 570
 compilación separada, 570
 manejo de excepciones, 578, 579, 581, 583,
 585, 710, 719, 721

Modos de

abrir un archivo binario, 340
 apertura de un archivo, 340

Movilidad: tecnologías, redes e internet
 móvil, 24, 25

Múltiples

instancias de un objeto, 388
 interfaces, 684

Multiplicidad, 413

Multitarea, 727

N

NetBeans, 745

Niveles de herencia, 429

Nombre de una función, 179

Nombres de arreglos como apuntadores,
 308

Normas de las clases abstractas, 679

Notación

 binaria, 748
 científica o exponencial, 749

Números en coma flotante, 749

O

Objeto
 con sus atributos, 387
 de la clase, 479
 estado y comportamiento, 384
 que envía el mensaje: *this*, 651
 Objetos, 364
 Opciones de compilación, 493
 Operaciones
 abstractas, 439
 no válidas con apuntadores, 312

Operador
 coma, 116, 120, 458
 condicional, 115, 457
 de asignación, 104, 456
 delete, 461
 instanceof, 656
 new, 459
 sizeof, 117, 457

Operadores, 454, 455
 aritméticos, 105, 112, 120, 454, 604
 booleanos, 112
 de asignación, 120
 de decrementación, 108
 de decremento, 148, 456
 de incrementación, 108
 de incremento, 148, 456
 de manipulación de bits, 605
 de postincremento, 110
 de preincremento, 110
 de prioridad, 107
 especiales, 120
 lógicos, 113
 matemáticos, 112
 relacionales, 111, 112, 120
 relacionales y lógicos, 455, 605
 y expresiones, 103, 604

Ordenación, 260
 o clasificación de datos, 260
 por inserción, 266
 rápida, 261
 Organización típica de un programa
 orientado a objetos, 364
 Otros ejemplos de definición/
 declaración, 282

P

Packages, 638
 Palabras reservadas, 82
 Paquetes, 645
 Paradigmas de programación, 32
 Parámetros
 const de una función, 191
 de la función, 187, 189
 en un *applet*, 633
 Partes de la manipulación de excepciones,
 578
 Pascalina, 3
 Paso de parámetros por valor, 188
 Patrones estructurales, 150
 Periféricos de salida, 12
 Pixel, 766
 Plantilla, 550
 de funciones, 551
 para manejo de pilas de datos, 564

Plantillas

 de clase, 559
 de función ordenar y buscar, 557
 o tipos parametrizados, 548

Polimorfismo, 371, 542, 681
 con ligadura dinámica, 543
 sin ligadura dinámica, 543

Posiciones de la cadena, 608
 Precedencia de operadores, 606

Preguntar antes de la iteración, 166

Preprocesador, 446

Primer programa Java, 599

Principio de
 encapsulamiento, 642
 ocultación de la información, 642

Prioridad, 119

 entre hilos, 734

Problemas en las funciones plantilla, 559

Proceso

 completo de depuración de un programa,
 78
 de compilación/ejecución de un programa,
 62
 de edición de un archivo fuente,
 77
 de ejecución de un programa en C, 76
 de programación, 20
 de un arreglo de tres dimensiones, 233

Programa

 Java, 756
 o archivo ejecutable, 62
 vacío, 741

Programación

 estructurada, 58
 modular, 58, 77
 orientada a objetos en Java. Clases y
 objetos, 637
 orientada a objetos (POO), 33, 60, 598

Programadores

Propiedades de Java, 598
 con un número no especificado de
 parámetros, 186
 de las funciones, 184, 185

Prueba (*testing*), 81, 752

Pruebas

Pseudocódigo, 37

Q

 ¿Qué es la realidad aumentada?, 26
 ¿Qué son clases?, 638
 ¿Qué son objetos?, 638

R

Recolección de objetos, 649, 675, 701
 Recuperación de información de una
 estructura, 287

Recursión

 infinita, 215
 versus iteración, 213

Recursividad indirecta: funciones
 mutuamente recursivas, 211

Redes

 inalámbricas, 24
 móviles, 25

Reescritura de código reusable, 371

- Referencia a la clase base: `super`, 674
 Referencias, 452
 Refinamiento de un algoritmo, 44
 Reglas
 básicas de formación de identificadores, 82
 funcionamiento de la asignación
 dinámica, 332
 visibilidad, 398
 Relaciones entre clases, 409
 Repetición o iteración, 145
 Repetición
 bucle `do-while`, 163
 bucle `for`, 154
 Representación
 de caracteres, 750
 de enteros, 748
 de imágenes, 750
 de la información en las computadoras
 (códigos de caracteres), 18
 de números reales, 749
 de sonidos, 751
 de valores numéricos, 748
 gráfica de los algoritmos, 47
 gráfica de una clase, 395
 gráfica en UML, 385
 Resolución de problemas con computadoras:
 fases, 34
 Responsabilidad y restricciones, 393
 Restricciones en
 asociaciones, 414, 419
 las agregaciones, 422
 Reutilización
 de código, 370
 o reusabilidad, 370
 Revolución Web 2.0, 22
- S**
 Salida, 97
 a la consola, 612
 de cadenas de caracteres, 100
 formateada con `printf`, 612
 Sangría en las sentencias `if` anidadas, 130
 Secciones pública y privada de una clase, 401
 Secuencia y sentencias compuestas, 461
 Secuencias de escape, 87
 Selección y repetición, 462
 Sentencia `break` en los bucles, 152
 de control `switch`, 132
 `do-while`, 464
 `for`, 464
 `if`, 462
 `package`, 645
 `return`, 469
 `return 0`, 741
 `switch`, 463
 `using`, 469
 `while`, 145, 463
- Sentencias, 753
 `break`, 162
 `break y continue`, 465
 de asignación múltiples, 603
 `if-else` anidadas, 129
 nulas en bucles `for`, 161
 Servidores, 5
 Signos de puntuación y separadores, 83
 Sincronización, 734
 Sintaxis de
 la implementación de una función
 miembro, 532
 un constructor, 531
 Sistemas
 embebidos, 5
 operativos móviles, 25
 Sobrecarga
 de funciones miembro, 499
 de operadores, 502
 de operadores unitarios, 504
 Software
 como servicio (SaaS), 24
 conceptos básicos y clasificación, 13
 de aplicaciones, 14
 de sistema, 14
Stack, 699
struct, 282
 Sub cadenas de la clase *String*, 608
 Subíndices de un arreglo, 223
 Superclase, 425
 Supercomputadoras, 5
 Symbols (iniciación) de apuntadores, 303
- T**
 Tablas de verdad, 113
 Tamaño de
 la secuencia de entrada, 166
 los arreglos, 225
 una estructura, 284
 Taxonomía de diagramas (notaciones) UML
 2.5, 375
 Técnica de muestreo, 751
 Técnicas de programación estructurada, 59
 Terminaciones anormales de un bucle, 149
 Tipo
 abstracto de datos, 689
 de dato de retorno, 180
 de dato lógico, 87
 Tipos
 abstractos de datos en Java, 656
 abstractos de datos: Clases, 365
 apuntadores, 450
 carácter, 453
 constantes, 451
 colecciones, 690
 de coma flotante (*float/double*), 85
 de coma flotantes (reales), 449
- de datos, 601
 de datos básicos/primitivos, 448
 de datos en C, 83
 de datos nativos, 448
 de mensajes, 393
 de métodos, 405
 de programas Java, 599
 enumeración, 452
 parametrizados o tipos genéricos, 548
 Tokens (elementos léxicos de los
 programas), 82
 Traductores, 28
 Tratamiento de los códigos de error, 575, 709,
 711, 713, 715, 717
 Traza de llamadas de funciones, 182
- U**
 Unidad central de proceso, 8
 Unidades de medida de memoria, 11
 Uso
 de estructuras en asignaciones, 283
 de las plantillas de funciones con clases, 569
 de `malloc()` para arreglos
 multidimensionales, 326
 de paréntesis, 107
 de sentencias `switch` en menús, 137
 del polimorfismo, 681
 Utilización de
 la multitarea, 727
 una clase plantilla, 565
 una plantilla de clase, 562
- V**
 Valor centinela, 150, 166
 Variable
 bandera, 151
 de control del bucle, 147
 registro, 197
 Variables
 automáticas, 196
 constantes y asignaciones, 601, 603
 estáticas, 197
 externas, 196
 globales, 96
 interface, 685
 locales, 95, 196
 `static`, 652
 Ventajas
 de `const` sobre `#define`, 92
 de los apuntadores, 308
 del polimorfismo, 681
 Verificación y depuración de un programa, 40
 Visibilidad de una función, 208
- W**
 Web Semántica y la Web 3.0, 23
 World Wide Web (WWW), 21

