

ANÁLISIS y DISEÑO de ALGORITMOS

Implementaciones en C y Pascal

Gustavo López - Ismael Jeder - Augusto Vega



ANÁLISIS y DISEÑO de
ALGORITMOS

Implementaciones en C y Pascal

ANÁLISIS y DISEÑO de ALGORITMOS

Implementaciones en C y Pascal

Gustavo López - Ismael Jeder - Augusto Vega

López, Gustavo
Análisis y diseño de algoritmos : implementaciones en C y Pascal /
Gustavo López ; Ismael Jeder ; Augusto Vega. - 1a ed. - Buenos Aires :
Alfaomega Grupo Editor Argentino, 2009.
336 p. ; 24x21 cm. - (3B)
ISBN 978-987-23113-9-1
1. Informática. 2. Programación. I. Jeder, Ismael II. Vega, Augusto III.
Título
CDD 005.1

Queda prohibida la reproducción total o parcial de esta obra, su tratamiento informático y/o la transmisión por cualquier otra forma o medio sin autorización escrita de Alfaomega Grupo Editor Argentino S.A.

Edición: Damián Fernández

Corrección: Paula Smulevich y Romina Aza

Coordinadora: Yamila Trujillo

Diseño de interiores: Juan Sosa

Diagramación de interiores: Diego Linares

Corrección de armado: Silvia Mellino

Revisión técnica: Pablo Sznajdleder

Diseño de tapa: Diego Linares

Internet: <http://www.alfaomega.com.mx>

Todos los derechos reservados © 2009, por Alfaomega Grupo Editor Argentino S.A.
Paraguay 1307, PB, oficina 11

ISBN 978-987-23113-9-1

Queda hecho el depósito que prevé la ley 11.723

NOTA IMPORTANTE: La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. Alfaomega Grupo Editor Argentino S.A. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Los nombres comerciales que aparecen en este libro son marcas registradas de sus propietarios y se mencionan únicamente con fines didácticos, por lo que Alfaomega Grupo Editor Argentino S.A. no asume ninguna responsabilidad por el uso que se dé a esta información, ya que no infringe ningún derecho de registro de marca. Los datos de los ejemplos y pantallas son ficticios, a no ser que se especifique lo contrario.

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino, S.A.

Paraguay 1307 P.B. "11", Buenos Aires, Argentina, C.P. 1057

Tel.: (54-11) 4811-7183 / 8352

E-mail: ventas@alfaomegaelitor.com.ar

México: Alfaomega Grupo Editor, S.A. de C.V.

Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100

Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396

E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A.

Carrera 15 No. 64 A 29, Bogotá, Colombia

PBX (57-1) 2100122 - Fax: (57-1) 6068648

E-mail: sciente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A.

General del Canto 370-Providencia, Santiago, Chile

Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786

E-mail: agechile@alfaomega.cl

A mi familia; y en especial a mis padres, que me acompañan en todo momento.
Vaya un agradecimiento para los miembros del Laboratorio de Informática de Gestión
de la Facultad de Ingeniería de la Universidad de Buenos Aires, al que dirijo.

Gustavo López

A mis seres queridos, que son el motor de mi vida... Y a mis sobrinos, que me llenan de felicidad.
Especialmente a Gladys, quien nos acompaña en cada instante de nuestras vidas.

Ismael Jeder

Gracias a todas las personas que me enseñaron a soñar. A Miryam y a Carlos. A María Inés.
A mis abuelos, Mabel, Zulema, Heli y Victoriano. Y a Dios, sobre todas las cosas.

Augusto Vega

Mensaje del Editor

Los conocimientos son esenciales en el desempeño profesional. Sin ellos es imposible lograr las habilidades para competir laboralmente. La universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una vida profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecerles a estudiantes y profesionales conocimientos actualizados dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Alfaomega hace uso de los medios impresos tradicionales en combinación con las tecnologías de la información y las comunicaciones (IT) para facilitar el aprendizaje. Libros como éste tienen su complemento en una página Web, en donde el alumno y su profesor encontrarán materiales adicionales, información actualizada, pruebas (test) de autoevaluación, diapositivas y vínculos con otros sitios Web relacionados.

Esta obra contiene numerosos gráficos, cuadros y otros recursos para despertar el interés del estudiante, y facilitarle la comprensión y apropiación del conocimiento.

Cada capítulo se desarrolla con argumentos presentados en forma sencilla y estructurada claramente hacia los objetivos y metas propuestas. Cada capítulo concluye con diversas actividades pedagógicas para asegurar la asimilación del conocimiento y su extensión y actualización futuras.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje, y pueden ser usados como textos guía en diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

**Lic. Gustavo López**

Es Master en Administración de Negocios (UTN). Especialista en Ingeniería Gerencial (UTN). Es Licenciado en Análisis de Sistemas egresado de la Universidad de Buenos Aires. Doctorando por la Universidad de Granada en el Doctorado en Enseñanza de las Ciencias y la Tecnología. Es docente en Algoritmos y Programación I y director del Departamento de Computación de la Facultad de Ingeniería de la Universidad de Buenos Aires. Publicó los libros *Algoritmia, Arquitectura de Datos y Programación Estructurada* (2003) y *Elementos de Diseño y Programación con ejemplos en C* (2006), ambos en Nueva Librería.

**Lic. Ismael Jeder**

Es Licenciado en Análisis de Sistemas egresado de la Universidad de Buenos Aires. Es Jefe de Trabajos Prácticos Interino de la materia Taller de Desarrollo de Proyectos III y Prosecretario del Departamento de Computación de la Facultad de Ingeniería de la Universidad de Buenos Aires.

**Ing. Augusto Vega**

Es Ingeniero en Informática egresado de la Universidad de Buenos Aires. Obtuvo la maestría en Arquitectura de Computadores, Redes y Sistemas del Departamento de Arquitectura de Computadores de la Universidad Politécnica de Cataluña. Es doctorando en la Universidad Politécnica de Cataluña, en el campo de Arquitectura y Tecnología de Computadores. Fue docente en materias relacionadas con programación y arquitectura de computadoras en la Universidad de Buenos Aires. Actualmente trabaja como investigador en el BSC-CNS (Barcelona Supercomputing Center – Centro Nacional de Supercomputación).

Revisor técnico: Ing. Pablo Augusto Szajdleder

Es Ingeniero en Sistemas de Información egresado de la Universidad Tecnológica Nacional. Tiene las certificaciones internacionales Sun Certified Java Programmer (SCJP, 1997) y Sun Certified Java Developer (SCJD, 1998) y está certificado como Instructor Oficial Java para Sun Microsystems (1997). Trabaja como instructor Java para Sun Microsystems, Oracle, Informix y Borland. Es profesor de la cátedra de Algoritmos y Estructuras de Datos en la Universidad Tecnológica Nacional (FRBA) y Profesor de Algoritmos I (Java) en la Universidad Nacional de San Martín.

Contenido	
Capítulo 1	
Introducción a la programación estructurada	1
1.1 Introducción.....	2
1.2 La computadora electrónica.....	2
1.3 Los lenguajes de programación	3
1.4 ¿Qué es un algoritmo?	4
1.5 Paradigma de programación estructurada	4
1.6 El lenguaje C.....	6
1.7 El lenguaje Pascal	6
1.8 Etapas del desarrollo de software	7
1.8.1 Análisis del problema.....	7
1.8.2 Diseño del algoritmo.....	8
1.8.3 Codificación	8
1.8.4 Compilación y ejecución.....	9
1.8.5 Verificación y depuración	9
1.8.6 Documentación.....	10
1.9 Estructura de un programa en C.....	11
1.9.1 Directivas al preprocesador.....	11
1.9.2 Prototipos de funciones.....	11
1.9.3 La función <code>main()</code>	12
1.9.4 Declaración de constantes.....	12
1.9.5 Declaración de tipos y variables.....	12
1.9.6 Lógica de la función principal.....	14
1.10 Poniendo todo junto	17
1.11 Estructura de un programa en Pascal	19
1.11.1 El programa principal	19
1.12 Ahora, integrando	22
1.13 Resumen	23
1.14 Contenido de la página Web de apoyo	24
Capítulo 2	
Datos y sentencias simples. Operaciones de entrada/salida	25
2.1 Introducción	26
2.2 Tipos de datos simples	26
2.3 Little endian vs. big endian.....	28
2.4 Modificadores de tipos en C.....	28
2.5 Palabra reservada <code>void</code>	29
2.6 Otros modificadores	29
2.7 Tipos de datos definidos por el usuario	30
2.8 Construcción de sentencias básicas	31
2.9 Operadores	32
2.9.1 Operadores aritméticos en C.....	32
2.9.2 Operadores aritméticos en Pascal	33
2.9.3 Operadores relacionales y lógicos en C.....	34
2.9.4 Operadores relacionales y lógicos en Pascal.....	34
2.9.5 Operadores de manejo de bits en C.....	35
2.9.6 Operadores de manejo de bits en Pascal	36
2.9.7 Otros operadores	37
2.10 Operaciones de entrada/salida.....	37
2.10.1 Función <code>printf()</code>	37
2.10.2 Vulnerabilidades de <code>printf()</code>	39
2.10.3 Función <code>scanf()</code>	39
2.10.4 Vulnerabilidades de <code>scanf()</code>	40
2.10.5 Entrada y salida en Pascal.....	41
2.11 Resumen	41
2.12 Problemas propuestos	42
2.13 Problemas resueltos.....	43
2.14 Contenido de la página Web de apoyo	55
Capítulo 3	
Subrutinas.....	57
3.1 Introducción	58
3.2 Funciones	59
3.3 Ámbito de las declaraciones.....	62
3.4 Parámetros	65
3.5 Argumentos por línea de comandos	67
3.6 Mapa de memoria	69
3.7 Consideraciones de desempeño.....	72

3.8 Resumen.....	74	6.2 Declaración y uso de registros	150				
3.9 Problemas propuestos	75	6.3 Registros como parámetros de funciones.....	152				
3.10 Problemas resueltos.....	76	6.4 Registros jerárquicos	154				
3.11 Contenido de la página Web de apoyo.....	88	6.5 Uniones	155				
Capítulo 4							
Tipos estructurados homogéneos. Vectores y matrices....	89	6.6 Tablas	156				
4.1 Introducción	90	6.7 Resumen.....	159				
4.2 Arreglos lineales.....	90	6.8 Problemas propuestos	159				
4.3 Declaración y uso de arreglos lineales	91	6.9 Problemas resueltos.....	160				
4.4 Arreglos multidimensionales	94	6.10 Contenido de la página Web de apoyo.....	177				
4.5 Arreglos como parámetros de subprogramas	96	Capítulo 7					
4.6 Cadenas de caracteres.....	100	Archivos	179				
4.7 Enumeraciones.....	102	7.1 Introducción.....	180				
4.8 Resumen.....	103	7.2 Tratamiento de archivos en lenguaje C	181				
4.9 Problemas propuestos	104	7.2.1 Apertura de un archivo.....	182				
4.10 Problemas resueltos.....	105	7.2.2 Cierre de un archivo	183				
4.11 Contenido de la página Web de apoyo.....	124	7.2.3 Funciones para manipulación de archivos.....	183				
Capítulo 5							
Complejidad algorítmica.		7.2.4 Archivos como parámetros de funciones	186				
Métodos de ordenamiento y búsqueda.....	125	7.3 Tratamiento de archivos en lenguaje Pascal	187				
5.1 Introducción	126	7.3.1 Apertura de un archivo.....	187				
5.2 Complejidad computacional.....	126	7.3.2 Cierre de un archivo	188				
5.2.1 Cota superior asintótica - O	127	7.3.3 Funciones para manipulación de archivos.....	188				
5.2.2 Cota inferior asintótica - Ω	128	7.3.4 Archivos como parámetros de procedimientos					
5.2.3 Cota ajustada asintótica - Θ	129	188				
5.3 Métodos de búsqueda.....	129	7.4 Archivos de acceso directo	189				
5.3.1 Búsqueda secuencial	129	7.4.1 Archivos de acceso directo en lenguaje C	189				
5.3.2 Búsqueda binaria	130	7.4.2 Archivos de acceso directo en lenguaje Pascal	190				
5.4 Métodos de ordenamiento.....	132	7.5 Operaciones entre archivos	191				
5.4.1 Ordenamiento por burbujeo.....	132	7.5.1 Apareo	191				
5.4.2 Ordenamiento por selección	133	7.5.2 Mezcla.....	192				
5.4.3 Ordenamiento por inserción	133	7.5.3 Intersección	193				
5.5 Mezcla de arreglos.....	134	7.5.4 Unión	193				
5.6 Resumen.....	136	7.6 Resumen	193				
5.7 Problemas propuestos	137	7.7 Problemas propuestos	194				
5.8 Problemas resueltos.....	137	7.8 Problemas resueltos	196				
5.9 Contenido de la página Web de apoyo.....	147	7.9 Contenido de la página Web de apoyo	213				
Capítulo 6							
Estructuras y tablas.....	149	Capítulo 8					
6.1 Introducción	150	Claves e índices.....	215				
			8.1 Introducción	216			
			8.2 Claves	216			
			8.3 Índices	218			
			8.4 Índices y archivos	219			
			8.4.1 Índices primarios y secundarios	221			

8.4.2 Eliminación y agregado de registros.....	221	10.10 Problemas resueltos	277
8.5 Resumen.....	221	10.11 Contenido de la página Web de apoyo	290
8.6 Problemas propuestos	222	Capítulo 11	
8.7 Problemas resueltos.....	223	El proceso de compilación	291
8.8 Contenido de la página Web de apoyo.....	245	11.1 Introducción.....	292
Capítulo 9			
Recurrencia.....	247	11.2 El proceso de compilación	292
9.1 Introducción	248	11.3 Preprocesamiento.....	292
9.2 Algoritmos recursivos	248	11.3.1 Directivas #define #undef.....	293
9.3 Tipos de recursividad	252	11.3.2 Directiva #error.....	293
9.4 Resumen.....	253	11.3.3 Directiva #include.....	293
9.5 Problemas propuestos	254	11.3.4 Directivas #if #ifdef #ifndef #else #endif	293
9.6 Problemas resueltos.....	254	11.3.5 Directiva #pragma	294
9.7 Contenido de la página Web de apoyo.....	262	11.3.6 Directivas {\$define} {\$undef}	294
Capítulo 10			
Memoria dinámica y manejo de punteros.....	263	11.3.7 Directivas {\$ifdef} {\$else} {\$endif}	294
10.1 Introducción.....	264	11.3.8 Directiva \${I}	295
10.2 Administración de memoria dinámica.....	265	11.4 Compilación	296
10.3 Punteros.....	265	11.5 Enlace	297
10.3.1 Punteros a memoria dinámica	267	11.6 Automatización del proceso de compilación	298
10.4 Punteros sin tipo.....	271	11.6.1 Herramienta make	300
10.5 Aritmética de punteros	273	11.6.2 Estructura del archivo makefile	300
10.6 Punteros y arreglos.....	274	11.6.3 Bloques de descripción.....	300
10.7 Punteros a funciones	275	11.6.4 Comandos	300
10.8 Resumen	275	11.6.5 Macros	300
10.9 Problemas propuestos.....	277	11.6.6 Reglas de inferencia	301



Información del contenido de la página Web.

El material marcado con asterisco (*) sólo está disponible para docentes.

Prefacio.

Video explicativo (02:03 minutos aprox.).

Capítulo 1. Introducción a la programación estructurada

Mapa conceptual.

Autoevaluación.

Video explicativo (02:25 minutos aprox.).

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 2. Datos y sentencias simples. Operaciones de entrada/salida

Mapa conceptual.

Autoevaluación.

Video explicativo (02:19 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 3. Subrutinas

Mapa conceptual.

Simulación:

- Función y procedimiento.

Autoevaluación.

Video explicativo (04:39 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 4. Tipos estructurados homogéneos. Vectores y matrices

Mapa conceptual.

Simulación:

- Operaciones entre vectores.
- Operaciones entre matrices.
- Trasposición de una matriz.
- Máximo y mínimo de un vector.

Autoevaluación.

Video explicativo (03:45 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 5. Complejidad algorítmica. Métodos de ordenamiento y búsqueda

Mapa conceptual.

Simulación:

- Búsqueda secuencial.
- Búsqueda binaria.
- Ordenamiento por burbujeo.
- Ordenamiento por selección.
- Ordenamiento por inserción.

Autoevaluación.

Video explicativo (03:02 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 6. Estructuras y tablas

Mapa conceptual.

Simulación:

- Uso de registros.
- Unión de registros.

Autoevaluación.

Video explicativo (02:20 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 7. Archivos

Mapa conceptual.

Simulación:

- Archivos de acceso secuencial.
- Archivos de acceso directo.

Lecturas adicionales:

- *Archivos* (versión preliminar de 20 páginas), de Martín Silva, será parte del libro *Sistemas Operativos* que publicará Alfaomega Grupo Editor.

Agradecemos a su autor por permitir que su escrito sea parte de las lecturas complementarias de esta obra.

Autoevaluación.

Video explicativo (04:21 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas.*

Presentaciones.*

Capítulo 8. Claves e índices

Mapa conceptual.

Simulación:

- Organización de archivo por un índice.

Respuesta y solución de problemas seleccionados.

Autoevaluación.

Video explicativo (02:38 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas. *

Presentaciones. *

Capítulo 9. Recurrencia

Mapa conceptual.

Simulador:

- Recursividad directa.
- Recursividad indirecta.

Autoevaluación.

Video explicativo (02:24 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas. *

Presentaciones. *

Capítulo 10. Memoria dinámica y manejo de punteros

Mapa conceptual.

Lecturas adicionales:

- *Memorias*, de Patricia Quiroga, es parte del libro *Arquitectura de Computadoras* de Alfaomega Grupo Editor (60 páginas). Agradecemos a su autora por permitir que su escrito sea parte de las lecturas complementarias de esta obra.

Autoevaluación.

Video explicativo (03:50 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas. *

Presentaciones. *

Capítulo 11. El proceso de compilación

Mapa conceptual.

Simulador:

- El proceso de compilación.

Autoevaluación.

Video explicativo (02:21 minutos aprox.).

Código fuente de los ejercicios resueltos.

Evaluaciones propuestas. *

Presentaciones. *

Lecturas adicionales:

- *Introducción a los diagramas de flujo* de Pablo Augusto Sznajdleder (20 páginas).
- *Programmer's guide*, Guía del programador de FreePascal (160 páginas en inglés)
- *User's guide* - Guía del usuario de FreePascal (167 páginas en inglés)
- *The GNU Pascal Manual* (552 páginas en inglés)
- *Language reference guide* (136 páginas en inglés).
- *The New C Standard: An economic and cultural commentary* (1615 páginas en inglés).
- *International Standard ©ISO/IEC ISO/IEC 9899:TC3* (552 páginas en inglés).

Software para descargar:

- FreePascal (compilador gratuito de Pascal).
- GNU Pascal (compilador gratuito de Pascal).
- MinGW (compilador gratuito de C).
- DJGPP (compilador gratuito de C).
- EditPad Pro (Este editor trabaja en conjunto con los compiladores FreePascal y MinGW, dándole una interface cómoda para programar).
- Dia (Herramienta para hacer, entre otros, diagramas de flujo).

Vínculos a páginas especialmente seleccionadas sobre algoritmos.

Glosario

Registro en la Web de apoyo.

Para tener acceso al material de la página Web de apoyo del libro:

1. Ir a la página <http://virtual.alfaomega.com.mx>
2. Registrarse como usuario del sitio y propietario del libro.
3. Ingresar al apartado de inscripción de libros y registrar la siguiente clave de acceso

4. Para navegar en la plataforma virtual de recursos del libro, usar los nombres de Usuario y Contraseña definidos en el punto número dos. El acceso a estos recursos es limitado. Si quiere un número extra de accesos envíe un correo electrónico a webmaster@alfaomega.com.mx

Estimado profesor: Si desea acceder a los contenidos exclusivos para docentes por favor contacte al representante de la editorial que lo suele visitar o envíenos un correo electrónico a webmaster@alfaomega.com.mx

Convenciones utilizadas en el texto.

El material marcado con asterisco (*) sólo está disponible para docentes.

	Conceptos para recordar: Bajo este ícono se encuentran definiciones importantes que refuerzan lo explicado en la página.	Libro	Web de apoyo
	Comentarios o información extra: Este ícono ayuda a comprender mejor o ampliar el texto principal.	Libro	Web de apoyo
	Diagrama de flujo: Representaciones gráficas sobre la estructura de cada capítulo.	Libro	Web de apoyo
	Simulación: Contenido interactivo para facilita el conocimiento.	Libro	Web de apoyo
	Software para descargar: Indica que está disponible el programa que se menciona.	Libro	Web de apoyo
	Videos explicativos: Comentarios del autor sobre los distintos capítulos.	Libro	Web de apoyo
	Autoevaluaciones: Un sistema de preguntas con respuestas múltiples.	Libro	Web de apoyo
	Lecturas complementarias: Escritos que profundizan sobre las temáticas tratadas.	Libro	Web de apoyo
	Vínculos de interés: Guías directas a temas relacionados.	Libro	Web de apoyo
	Glosario: Definiciones de términos que no se utilizan en forma frecuente o que son de gran relevancia para el objeto de estudio.	Libro	Web de apoyo
	Código fuente: El código de los ejercicios resueltos disponible para su uso. En el libro se utiliza otra tipografía para los listados de código o cuando se refiere a un comando en el texto.	Libro	Web de apoyo
	Evaluaciones propuestas*: Bajo este rótulo se encuentran exámenes realizados por el autor para sus cursos.	Libro	Web de apoyo
	Foro*: Un espacio para dudas, críticas o sugerencias, el mejor medio para comunicarse con el autor del libro.	Libro	Web de apoyo
	Presentaciones*: Se encuentran aquí los archivos necesarios para proyectar en una pantalla las diapositivas de los capítulos de cada clase. Facilita la comprensión por parte del estudiante.	Libro	Web de apoyo
	Ejercicios resueltos*: Un conjunto de consignas para desarrollar con su correspondiente respuesta.	Libro	Web de apoyo

Prólogo

De un buen libro sobre fundamentos de programación debemos esperar dos cualidades: Simpleza y profundidad.

Simpleza para que el lector pueda comprender de forma cabal lo que se desea comunicar, y profundidad para que con el mayor cuidado y rigor se acceda al conocimiento.

Estos atributos, que a primera vista parecen contradictorios, se alcanzan en Análisis y diseño de algoritmos, implementaciones en C y Pascal.

Los autores, que cuentan con varios años de experiencia al frente de cursos de programación, han sido capaces de abordar, desde un punto de vista simple y práctico, todos los tópicos de la programación de base. Es una obra adecuada y recomendable para el ámbito académico, ya que está pensada para facilitar el proceso de aprendizaje y es, también, un libro de consulta que no debe faltar en la biblioteca de todo programador.

Mateo Valero

Director del Barcelona Supercomputing Center - Centro Nacional de
Supercomputación (BSC-CNS)
Barcelona, España

Prefacio.

El presente libro corresponde a la condensación de muchos años de docencia y trabajo profesional de los autores. En este sentido, recopila la experiencia recabada durante todo este período. Presenta una introducción al estudio de la algoritmia y su implementación en lenguajes Pascal y C. Las principales técnicas de diseño de algoritmos se muestran mediante una explicación detallada y la implementación de estructuras de datos, con el fin de arribar a soluciones óptimas. Estas explicaciones se aportan de la manera más sencilla posible, pero sin perder el detenimiento ni el rigor científico-conceptual.

El libro está concebido para utilizarse como texto en asignaturas que traten sobre algoritmos y estructuras de datos. No obstante, como se presentan muchas implementaciones prácticas de algoritmos, también es útil para aquellos profesionales que desarrollen aplicaciones informáticas. En este contexto, los autores de este libro –profesionales de la enseñanza universitaria– hemos considerado que, a pesar de la excelente bibliografía existente, quedaba un nicho sin cubrir, que abarcaba un enfoque eminentemente práctico con implementaciones en ambos lenguajes y con el soporte teórico de rigor imprescindible. Los algoritmos se elaboran mediante refinamientos sucesivos. Por último, los ejemplos se desarrollan en Pascal y C.

Contenido.

El libro presenta las técnicas principales utilizadas en la implementación de algoritmos y estructuras de datos.

Esta obra está organizada en once capítulos, cada uno esencial en el eje conceptual que envuelve.

En el capítulo 1 se analiza el estilo y las buenas prácticas de la programación, y se facilita un conjunto de herramientas, nociones y conceptos básicos utilizados en programación.

El capítulo 2 se centra en estructuras de datos y el uso de sentencias simples para su manipulación, el puntapié inicial de cualquier curso de algoritmia o programación. En este capítulo se brinda un panorama detallado de estructuras y sentencias básicas. Los algoritmos y las estructuras de datos más robustas y complejas en general devienen de estos bloques constructivos, por lo que resulta esencial aprehender cabalmente estos conceptos.

En el capítulo 3 se acerca al lector a la técnica de “divide y vencerás”; esto es, dividir un problema en otros más sencillos o pequeños con el fin de abordarlos en forma individual y que, en su conjunto, resuelvan el problema original. Expresado en otras palabras, esta buena práctica de programación permite descomponer un aplicativo en módulos más chicos, cada uno más pequeño y más fácil de programar y mantener.

En el capítulo 4 se presentan conjuntos de datos implementados por los lenguajes Pascal y C. Se brindan ejemplos de cómo estos lenguajes de programación soportan no tan sólo va-



En la página Web de apoyo encontrará un breve comentario del autor sobre el libro.

riables simples, como en los capítulos anteriores, sino conjuntos de datos como vectores (arreglos lineales) y matrices (arreglos multidimensionales).

En el capítulo 5, Complejidad algorítmica - Métodos de ordenamiento y búsqueda, se desarrolla el uso más eficiente de las operaciones de búsqueda, ordenación y mezcla, en las que estadísticamente toda computadora emplea la mayor parte de su tiempo de procesamiento. En este capítulo exclusivo se analizan los algoritmos y los métodos de más eficiente implementación. Por su gran trascendencia, se incluyó una amplia selección de ejercicios y problemas resueltos.

El capítulo 6 abarca el uso de estructuras y tablas. Se describen registros y conjuntos acompañados de numerosos ejercicios y problemas resueltos que faciliten un uso eficiente.

En el capítulo 7 se trata el uso de archivos. Su concepto, diseño, estructura e implementación son, sin duda alguna, el soporte fundamental para la resolución y la implantación de cualquier aplicativo que procese información. Sin un tratamiento oportuno y adecuado de esa información y su correspondiente almacenamiento en el soporte que se considere conveniente, el mundo no podría resolver sus problemas de manera fidedigna.

En el capítulo 8 se desarrollan los conceptos de claves e índices. Aquí se instala la idea de que a la porción de información que empleamos para referirnos en forma específica a un bloque particular de datos se la denomina clave. Con el fin de manipular cantidades importantes de datos, debería pensarse en la posibilidad de construir un índice de modo que al brindar cierta información relacionada con un registro, podamos acceder a él en forma directa. En la segunda mitad de este capítulo se introducirá el concepto de índice y se tratarán sus nociones fundamentales por medio del desarrollo de numerosos ejercicios.

En el capítulo 9 se desarrolla la propiedad de la recursividad, propiedad de una rutina, un subprograma o una entidad de poder llamarse o invocarse a sí mismo. Se trata de uno de los temas más difíciles y, a la vez, más utilizados por la programación.

En el capítulo 10 se presenta el tratamiento de la memoria en forma dinámica. Todas las variables y las estructuras de datos declaradas en el ámbito de un programa residen en memoria. Este espacio en memoria se solicita en el instante en que arranca la ejecución del programa, y se libera cuando termina. En este caso, las variables en cuestión se denominan estáticas, y son las que hemos estado utilizando hasta este momento. Otra alternativa para asignar memoria a variables y estructuras de datos es hacerlo de manera dinámica: La memoria se solicita y se libera en el punto donde se hará uso de ella (durante la ejecución del programa).

En el capítulo 11 se abordan temas relacionados con el ambiente de desarrollo para la implementación y la compilación de algoritmos. Se analizan por separado las etapas más importantes que comprenden el proceso de compilación (preprocesamiento, generación de código y enlace), y también se presentará una forma de automatizar ese proceso (mediante la herramienta `make`). Esto último es relevante cuando los proyectos son de gran envergadura y en especial cuando involucran varios archivos fuente, de cabecera o bibliotecas.

Para el profesor.

El material que se presenta está especialmente pensado para utilizarlo como libro de texto en cursos de grado. También es adecuado en la formación de programadores en el área de Algoritmia. Está diseñado para aprender desde el inicio un lenguaje de alto nivel, como Pascal o C, así como proveer conocimientos mínimos sobre el manejo de sistemas de computación.

En la Web de apoyo hay material complementario que incluye diapositivas para el desarrollo de todos los temas de cada capítulo.

Para el estudiante.

Analizar la eficiencia en el diseño de los algoritmos es de vital importancia. La técnica presentada en este libro es “dividir para vencer”, de refinamientos sucesivos o metodología *top-down*. Le hemos puesto un acento especial a los principales paradigmas de diseño de algoritmos.

Nuestro trabajo está pensado para nivel inicial, no obstante, profundiza tanto en la faz teórica como en la práctica, y brinda una cantidad abundante de ejercicios con el fin de fijar los conceptos desarrollados.

Se encuentran en la Web de apoyo valiosas simulaciones de conceptos desarrollados en el libro, autoevaluaciones sobre cada capítulo, el software gratuito que se menciona en el libro y el código de los problemas resueltos.

Para el profesional.

Nuestro trabajo también puede utilizarse profesionalmente como manual para programadores que ya estén familiarizados con el tema; en ese sentido, se presentan prototipos de implementaciones de algoritmos eficientes. Los algoritmos desarrollados en este texto tienen una gran aplicación práctica y, a lo largo del libro, se abordan diferentes aspectos relacionados con su implementación más eficiente. Los algoritmos propuestos se implementaron en C y Pascal.

Objetivos.

- Que el docente cuente con herramientas didácticas para facilitar el proceso enseñanza-aprendizaje.
- Que el estudiante adquiera las habilidades y las destrezas propias del manejo de un método algorítmico disciplinado, logradas por medio del uso de estructuras abstractas, de control, selectivas, secuenciales y repetitivas.
- Que el estudiante utilice técnicas de modularización, con el fin de hacer una abstracción más sencilla de los problemas que se le planteen.
- Que el estudiante adquiera habilidades y destrezas para utilizar buenas prácticas en las etapas de diseño, codificación, depuración, pruebas y documentación de sus aplicativos.
- Que el estudiante adquiera habilidades y destrezas para la implementación de soluciones en lenguajes C y Pascal.

1

Introducción a la programación estructurada

Contenido

1.1 Introducción	2
1.2 La computadora electrónica	2
1.3 Los lenguajes de programación.....	3
1.4 ¿Qué es un algoritmo?.....	4
1.5 Paradigma de programación estructurada.....	4
1.6 El lenguaje C	6
1.7 El lenguaje Pascal.....	6
1.8 Etapas del desarrollo de software.....	7
1.9 Estructura de un programa en C	11
1.10 Poniendo todo junto	17
1.11 Estructura de un programa en Pascal.....	19
1.12 Ahora, integrando.....	22
1.13 Resumen.....	23
1.14 Contenido de la página Web de apoyo.....	24

Objetivos

- Introducir algunas definiciones básicas de informática.
- Abstraer un problema del mundo real y convertirlo en un modelo computable, vale decir, que pueda ser introducido en una computadora u ordenador, con el fin de ser resuelto.
- Señalar el estilo y las buenas prácticas de la programación.
- Dominar el conjunto de herramientas, nociones y conceptos básicos utilizados en programación.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.



Máquina diferencial de Babbage

Charles Babbage diseño la máquina en 1821, con la intención de que evalué funciones polinómicas. Para realizar el cálculo de dichas funciones se basó en el método de diferencias finitas que se limita a una operatoria de suma de los términos del polinomio, de esta forma realizaba los cálculos con menor complejidad.

1.1 Introducción.

En este capítulo se brinda al lector un conjunto de nociones y conceptos básicos del ámbito de la informática y, en particular, de lo que se denomina programación. Estas ideas fundamentales permitirán que la persona que esté dando sus “primeros pasos” comprenda el marco en el que se desenvuelven otros aspectos más complejos de la informática, que se desarrollarán a lo largo de este libro.

Lo cierto es que en la vida cotidiana el “informático” (espero que se nos permita denominar de esta manera a aquella persona que realiza alguna de las tantas disciplinas que abarca la informática) rara vez se pregunta sobre el origen del término “computar” y sus derivados (computadora, computación, etc.), y sobre la necesidad que llevó al ser humano a desarrollar máquinas cada vez más complejas y poderosas para realizar mayor cantidad de cálculos en menos tiempo. Es algo esperable. Crecemos, desde pequeños, realizando cálculos y “cuentas” más o menos complejos. Al principio, los dedos de las manos son nuestra “computadora” más eficaz. Luego, una calculadora electrónica de bolsillo y, más adelante, podría ser una computadora personal o una consola de videojuegos, entre tantos dispositivos electrónicos que en el presente quedan comprendidos por el término “computadora”.

Computar, como verbo, proviene del latín *computare*, que significa contar o calcular. Por su parte, *computare* se forma con otras dos palabras del latín: *com* (que significa “con”) y *putare* (que significa “pensar”). O sea, que en el término “computadora” encontramos encerrados varios conceptos: el de “contar” (aspecto mecánico) y el de “pensar” (aspecto lógico). Así, y de una forma genérica, se puede definir a la computadora como un instrumento que, con parte de mecánica y parte de lógica, permite realizar cálculos y cuentas.

Sin embargo, cuando en la actualidad utilizamos el término “computadora”, está claro a qué nos referimos. Es que estos dispositivos se volvieron un elemento tan común en nuestras vidas como lo son la televisión o el teléfono. Ahora bien, en este caso, es necesario aclarar que la computadora como la conocemos es un dispositivo electrónico y que también pueden clasificarse como tales un ábaco, la máquina diferencial de Babbage o un “cuenta ganado”.

1.2 La computadora electrónica.

Una computadora electrónica es una máquina, compuesta por un conjunto de circuitos electrónicos, algunos componentes mecánicos e interfaces para interactuar con el exterior (usuarios u otros dispositivos), y cuya finalidad es procesar información para obtener resultados. Los datos que constituyen la entrada (*input*) a la computadora se procesan mediante una lógica (programa) para producir una salida (*output*), como se observa en la figura siguiente:

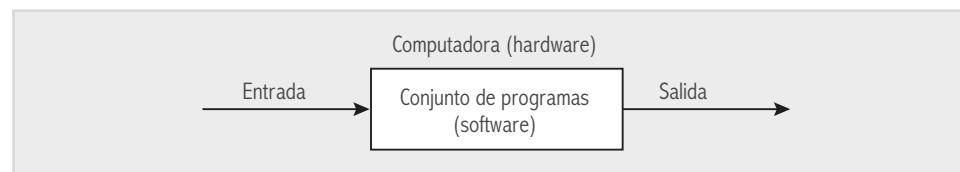


Fig. 1-1. Modo en que la computadora procesa la información.

Como se indicó antes, en una computadora se pueden distinguir un aspecto físico y otro lógico. En una computadora electrónica el aspecto mecánico suele denominarse hardware, y lo conforman los circuitos electrónicos (microprocesador, memorias, conexiones), las partes mecánicas y los dispositivos periféricos. Asimismo, el aspecto lógico es el software o programas, esto es, instrucciones y acciones que, con una lógica definida, determinan qué procesamiento realizar sobre los datos de entrada.

En este escenario el “actor” más importante es el programador, que es la persona responsable de indicar a la computadora la lógica de procesamiento que debe aplicarse sobre las entradas, expresada en forma de programa. La tarea del programador no es simple, principalmente porque en la práctica es más que un mero tipeador de instrucciones, ya que suele abocarse también al análisis y diseño previos, y a la toma de decisiones para ese fin.

En cuanto a los programas que conviven en una computadora, podemos clasificarlos en:

- Software del sistema.
- Software de aplicación.

El software del sistema es el conjunto de programas indispensables para que la máquina funcione. Aquí, el más importante es el sistema operativo, que es el encargado de administrar los recursos disponibles (por ejemplo, el almacenamiento de datos en archivos, el acceso a dispositivos periféricos como una impresora, etc.) y también de gestionar la interacción con el usuario. Además del sistema operativo, suelen considerarse software del sistema los compiladores y las herramientas disponibles para la creación de programas nuevos.

Con el término software de aplicación nos referimos a programas de propósito general para la realización de tareas concretas, como procesos contables, aplicaciones científicas, hojas de cálculo, juegos, etc. Un ejemplo concreto son los programas que el lector podrá construir en breve. El software de aplicación, por su parte, requiere al software del sistema para el acceso a los recursos o la interacción con el “mundo exterior” (periféricos, usuario, computadoras remotas).

1.3 Los lenguajes de programación.

Las computadoras son máquinas, eso está claro, y como tales carecen de inteligencia y capacidad de reflexión, por lo que sus acciones están completamente controladas y dirigidas por pulsos electrónicos que “marcan el ritmo”. Nada está librado al azar.

El ritmo de estas acciones está gobernado por el programa en ejecución y, por su parte, el programa está diseñado y construido por un programador. O sea que el programador controla el destino de los procesos dentro de una computadora por medio de los programas.

No obstante, ¿cómo se construye un programa y cómo se lo pone a “correr”? Un programa, casi como una obra de arte, requiere dos cosas fundamentales: creatividad y disciplina. Lo primero, la creatividad, es una cualidad subjetiva y puede darse en mayor o menor medida en diferentes programadores. En cambio, la disciplina se adquiere con aprendizaje y práctica. El programador no puede lanzarse a crear un programa sin haber definido, entre otras cosas, cuál es el paradigma al que se va a ajustar. Luego, las acciones que conforman el procesamiento de los datos se expresan como sentencias, y se escriben utilizando un lenguaje de programación, que facilita la construcción de programas sin necesidad de que el programador deba manipular pulsos electrónicos para que la computadora funcione. Estos lenguajes se clasifican en tres grandes categorías:

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

Los lenguajes de máquina son aquellos cuyas instrucciones entiende directamente la computadora, y no necesitan traducción posterior para que el procesador pueda comprender y ejecutar el programa. Las instrucciones en lenguaje de máquina se expresan en términos de la unidad de memoria más pequeña, el bit (dígito binario, 0 o 1).

Los lenguajes de bajo nivel o ensambladores fueron diseñados para simplificar el proceso de la programación en lenguaje de máquina, que resulta difícil y tedioso. Estos lenguajes son



Tres tipos de lenguajes:

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

dependientes de la máquina, o sea que dependen del conjunto de instrucciones específicas de la computadora. En este lenguaje las instrucciones se escriben en códigos alfabéticos conocidos como mnemónicos y que son abreviaturas en inglés: ADD para la suma, DIV para dividir, etc. De este modo, los mnemónicos son mucho más fáciles de recordar que las secuencias de dígitos 0 y 1.

Los lenguajes de programación de alto nivel son aquellos como el Cobol o el Pascal en los que las instrucciones o sentencias se escriben con palabras similares a los lenguajes humanos, lo que facilita la escritura y la comprensión del programador. Además, también propician la portabilidad del software, esto es, la posibilidad de que un programa escrito en una computadora determinada pueda ejecutarse en otra diferente.



Algoritmo de compresión.

Diariamente utilizamos algoritmos de compresión, por ejemplo, cuando vemos una imagen en Internet o escuchamos un MP3. Este algoritmo realiza una estadística de datos que se repiten y reformula la manera en que los datos se representan a nivel binario.

1.4 ¿Qué es un algoritmo?

El Diccionario de la Real Academia Española indica que un algoritmo es un “conjunto ordenado y finito de operaciones que permite hallar la solución de un problema”. En términos sencillos, un algoritmo es una “receta”, o sea, un conjunto de pasos que, ejecutados de la manera correcta, permite obtener un resultado (en un tiempo acotado).

A menudo, los algoritmos están asociados con las computadoras y los lenguajes de programación. Sin embargo, se los encuentra en la vida cotidiana, cuando cada uno de nosotros (racionalmente o por instinto) ejecuta acciones para lograr fines determinados. Por ejemplo, una taza de café con leche podría prepararse mediante el siguiente algoritmo:

1. Encender una hornalla.
2. Colocar una jarra con leche sobre la hornalla.
3. Esperar a que la leche hierva.
4. Colocar café en una taza.
5. Verter un poco de leche en la taza y batir.
6. Verter más leche en la taza hasta colmarla.
7. Agregar azúcar a gusto.



Fortran.

Es un lenguaje de programación de alto nivel y propósito general. La comunidad científica lo utiliza para realizar aplicaciones de cálculo. Este lenguaje involucra la aritmética de números complejos, que incrementa la cantidad de aplicaciones para las que se utiliza.

Si se nos permite enriquecer la definición expuesta, también es necesario que las operaciones que forman parte del algoritmo estén perfectamente definidas y que no haya ambigüedad al momento de ejecutarlas. En el segundo paso del ejemplo uno podría preguntarse en cuál de las hornallas colocar la jarra. Parece algo muy obvio para nosotros que somos seres humanos, pero no para una computadora, que no es capaz de razonar ni de sacar conclusiones por cuenta propia.



Cobol.

Desarrollado en 1960 para homogeneizar, con un lenguaje de programación universal, los distintos tipos de computadoras. Se pensó para manejar la información de negocios.

Un algoritmo es unívoco, lo que implica que si se ejecuta varias veces el mismo algoritmo sobre el mismo conjunto de datos de entrada, siempre se obtienen los mismos datos de salida. Además, el resultado debe generarse en un tiempo finito. Los métodos que utilizan algoritmos se denominan métodos algorítmicos, en oposición a los 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 habían sido convertidos con facilidad en ellas. En los últimos años, sin embargo, las técnicas de inteligencia artificial hicieron posible la implementación del proceso heurístico en computadoras.

1.5 Paradigma de programación estructurada.

Los albores de la programación de computadoras se remontan a la segunda mitad de la década de 1950, con el nacimiento de los lenguajes Fortran y Cobol (*COmmon Business-Oriented Language*), entre otros. Por entonces, no estaban claras las pautas a seguir en el proceso de

construcción de programas, y muchas veces numerosas decisiones del diseño quedaban sujetadas a criterios personales del programador. Faltaba mucho por hacer.

En 1966 los matemáticos Corrado Böhm y Giuseppe Jacopini demostraron con el “teorema de estructura” que un programa propio puede escribirse utilizando sólo tres tipos de estructuras de control:

- Secuenciales.
- Selectivas.
- Repetitivas.

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

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

El “teorema de estructura” sentó las bases de la programación estructurada, estableciendo normas pragmáticas para el desarrollo de programas. Asimismo, aunque con frecuencia los créditos son otorgados a Böhm y Jacopini, lo cierto es que las mayores contribuciones las realizó Edsger Dijkstra con ulterioridad. Sus aportes principales fueron el concepto de “modularización” (particionar el problema en módulos más pequeños y manejables) y la abolición de la sentencia GOTO.

Así, la programación estructurada presenta las siguientes características:

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente (partiendo de lo más “general”, y descendiendo hacia lo más “particular”).
- Cada módulo se codifica utilizando las tres estructuras de control básicas (secuenciales, selectivas y repetitivas, con ausencia total de sentencias GOTO).

La sentencia GOTO es una instrucción presente en muchos lenguajes de programación, y “rompe” la estructura de un programa, permitiendo un comportamiento más laxo del flujo de ejecución. Cuando se ejecuta esa instrucción, el flujo de ejecución no continúa con la siguiente, sino con la que esté indicada en la propia sentencia GOTO. Esto suele conducir al desarrollo de programas confusos y de alto costo de mantenimiento (incluso por el propio programador), que son una fuente habitual de errores. Sin embargo, es preciso aclarar que en lenguajes de bajo nivel o ensamblador, el GOTO es indispensable para implementar estructuras selectivas o repetitivas no presentes a ese nivel.

De los lenguajes contemporáneos, C y Pascal poseen características que los hacen atractivos para la programación estructurada, en especial en un ámbito académico:

- Lenguajes de propósito general.
- Lenguajes imperativos (orientados a órdenes).
- Lenguajes estructurados (soportan las estructuras básicas secuenciales, selectivas y repetitivas; no necesitan la sentencia GOTO).
- Lenguajes recursivos (soportan la recursividad: propiedad de llamarse a sí mismos una función o un procedimiento).
- Tipos de datos simples y estructurados, así como datos definidos por el usuario.
- Generación de programas ejecutables rápidos y eficientes mediante el uso de compiladores.



Vínculo a la página Web personal de Corrado Böhm disponible en la Web de apoyo.

- Facilidad para realizar programación modular debido a la posibilidad de diseñar subprogramas o módulos.
- Posibilidad de ser tratados como lenguajes de alto nivel.
- Posibilidad de programación a nivel de sistema y de dispositivos físicos (en el caso del lenguaje C).
- Buen manejo de archivos físicos.

1.6 El lenguaje C.



Vínculo a la página Web personal de Dennis Ritchie disponible en la Web de apoyo

A finales de la década de 1960 y principios de la de 1970, Dennis Ritchie (un científico norteamericano, que por entonces trabajaba para los Laboratorios Bell) creó un nuevo lenguaje de programación de propósito general, imperativo y estructurado, que llamó "C", y que en poco tiempo se convirtió en el lenguaje más popular para el desarrollo del sistema operativo UNIX (que, hasta entonces, era programado en lenguaje ensamblador). Su nombre se debe a que muchas características fueron tomadas de su antecesor, el lenguaje de programación "B", que por su parte se basa en un lenguaje de 1966 llamado "BCPL" (*Basic Combined Programming Language*).

El lenguaje de programación C es de propósito general, lo que significa que puede utilizarse para implementar algoritmos que son solución de problemas de diversa naturaleza, como matemática, ingeniería, física, enseñanza de idiomas, diseño industrial, biología, desarrollo de software de base, economía, contabilidad, gestión de bases de datos, telecomunicaciones, redes, programación de dispositivos electrónicos, etc. En pocos años se convirtió en el lenguaje estándar para estudiantes de programación en muchas universidades.

Un aspecto destacable del lenguaje C es que resulta sensible a mayúsculas y minúsculas, por lo que no es lo mismo escribir `int` (que es una palabra reservada para indicar el tipo de dato entero) que `INT`.

El desarrollo de programas en un lenguaje requiere "palabras reservadas" (*keywords*) que conforman el "vocabulario" capaz de ser entendido por ese lenguaje. Son "reservadas" en el sentido de que el programador no puede asignarles otros significados (por ejemplo, en C la palabra reservada `float` indica un tipo de dato de coma flotante –real– y sólo se la puede emplear con esa connotación). Por su parte, C tiene pocas palabras reservadas: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `restrict`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

1.7 El lenguaje Pascal.



En la Web de apoyo encontrará el vínculo a la página de Niklaus Wirth.

Otro lenguaje de uso extendido en el ámbito académico para la enseñanza de la programación estructurada es Pascal. Este lenguaje, que es contemporáneo a C, surge en 1969, y su creador es Niklaus Wirth, un científico de origen suizo.

Pascal, como el caso de C, también es un lenguaje de propósito general, imperativo y estructurado. Su característica más sobresaliente es que está diseñado con el propósito de incentivar las buenas prácticas de la programación estructurada, o, como el propio Wirth afirmó, "que un lenguaje usado en la enseñanza debe mostrar estilo, elegancia, consistencia, a la vez que refleje las necesidades (pero no siempre los malos hábitos) de la práctica".

A diferencia del lenguaje C, Pascal no es sensible a mayúsculas y minúsculas. Así, para utilizar un tipo de dato entero, es indistinto emplear las palabras reservadas `integer`, `Integer` o `INTEGER`.

1.8 Etapas del desarrollo de software.

La razón principal para que las personas aprendan lenguajes de programación es utilizar la computadora como una herramienta para la resolución de problemas. En este sentido, pueden identificarse dos etapas en el proceso de resolución de problemas asistida por computadora:

1. Resolución del problema en sí.
2. Implementación (realización) de la solución en la computadora.

El resultado de la primera etapa es el diseño de un algoritmo para resolver el problema. Como se indicó antes, un algoritmo es un conjunto finito de instrucciones que conducen a la solución del problema. En la comunidad de habla castellana es frecuente la descripción del algoritmo en lenguaje español, pese a que todos los compiladores e intérpretes de lenguajes de programación tengan sus palabras reservadas en inglés.

El algoritmo se expresa en un lenguaje de programación que la computadora pueda comprender (por ejemplo, C o Pascal, entre tantos otros). Ese algoritmo, expresado en cualquier lenguaje de programación de computadoras, se denomina programa. El programa en ejecución en la computadora se denomina proceso.

El diseño de programas es una tarea difícil y un proceso creativo. No hay un conjunto completo de reglas para indicar cómo escribir programas. Sin embargo, se debe considerar el uso de lineamientos o conductas para la obtención de una solución computable (o sea que la solución se exprese mediante un programa de computadora), a saber:

- Análisis del problema.
- Diseño del algoritmo.
- Codificación.
- Compilación y ejecución.
- Verificación y depuración.
- Documentación.

Las dos primeras tareas conducen a un diseño detallado escrito en forma de algoritmo. Durante la codificación se implementa el algoritmo en un código escrito en un lenguaje de programación, que refleja las ideas desarrolladas en las fases de análisis y diseño.

La compilación tiene como resultado el código de máquina, que es la traducción del código fuente, mediante el empleo de intérpretes o compiladores. A continuación el programa compilado es alimentado en la computadora para que ésta lo ejecute (indicando el punto de partida y los datos de entrada).

La verificación y la depuración son necesarias para encontrar errores y eliminarlos. Podrá comprobar que mientras más tiempo gaste en el análisis y el diseño, menos tiempo invertirá en verificación y depuración. Además, las etapas de codificación, compilación, ejecución, verificación y depuración se llevan a cabo en forma iterativa, lo que significa que al momento de verificar y depurar el programa, es necesario volver a codificar partes de él, compilar y ejecutar.

Además de todo esto, se debe realizar la documentación del programa, con el objeto de que cualquier persona ajena a él pueda entender qué hace y cómo lo hace.

1.8.1 Análisis del problema.

El primer paso para encontrar la solución a un problema mediante una computadora es analizarlo. Se requiere definir el problema de la manera más precisa posible, tarea que, por lo general, requiere el máximo de imaginación y creatividad por parte del programador.



Proceso de resolución de problemas

1. Resolución del problema en sí.
2. Implementación (realización) de la solución en la computadora.



Conductas que hay que seguir para la obtención de una solución computable.

- Análisis del problema.
- Diseño del algoritmo.
- Codificación.
- Compilación y ejecución.
- Verificación y depuración.
- Documentación.

Dado que se busca una solución, se debe examinar el problema cuidadosamente con el fin de identificar qué tipo de información se necesita producir. A continuación, el programador debe observar los elementos de información dados en el problema que puedan ser útiles para obtener la solución. Por último, debe generarse un procedimiento para producir los resultados deseados a partir de los datos, y que será el algoritmo.

La etapa de análisis es de gran importancia y aquí es necesario dedicar tiempo y esfuerzo considerables para no tener sorpresas en las etapas posteriores.

1.8.2 Diseño del algoritmo.

Durante el análisis del problema se determina qué hace el programa. Con el diseño se reconoce cómo el programa realiza la tarea solicitada. La transición desde el análisis hasta el diseño implica pasar desde un escenario ideal a uno realista, que considera las limitaciones del ámbito donde el programa será ejecutado.

La forma habitual de abordar la resolución de un problema complejo es dividirlo en subproblemas y a continuación dividir éstos en otros de nivel más bajo, hasta que pueda implementarse una solución en la computadora. Este método se conoce como diseño descendente o *top-down*. El proceso de dividir el problema en cada etapa y expresar cada paso en forma más detallada se denomina refinamiento sucesivo.

Cada subproblema se resuelve mediante un módulo (subprograma) que tiene un solo punto de entrada y otro de salida, de acuerdo con el paradigma de programación estructurada. Así, el resultado tendrá la forma de un módulo principal (el de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que, por su parte, pueden llamar a otros subprogramas. El diseño de los programas estructurados de esta forma se denomina modular, y el método de fragmentar el programa en módulos más pequeños se llama programación modular. Los módulos pueden planearse, codificarse, comprobarse y depurarse en forma independiente (incluso entre diferentes programadores) y, a continuación, combinarse entre sí.

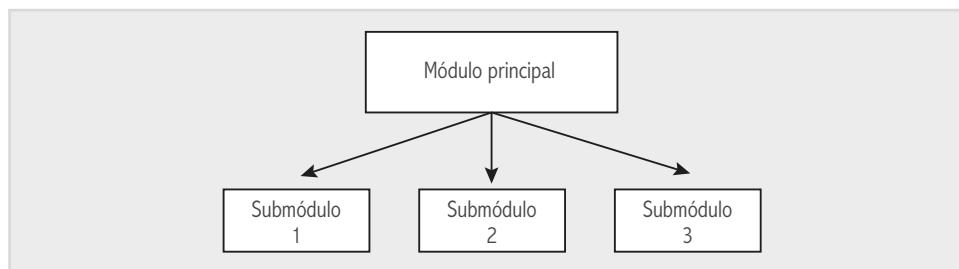


Fig. 1-2. Esquema de diseño modular y programación modular.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar con ulterioridad, aunque en la práctica suele ser necesario considerar también las características (ventajas y limitaciones) de cierto número de lenguajes “candidatos”.

1.8.3 Codificación.

Codificar es escribir, en lenguaje de programación de alto nivel, la representación del algoritmo desarrollado en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código debería poder escribirse con igual facilidad en un lenguaje u otro.



Diseño descendente o *top down*:

Resolución de un problema mediante la división de éste en subproblemas menores.

1.8.4 Compilación y ejecución.

Una vez que el algoritmo se convirtió en programa mediante el proceso de codificación, es preciso traducirlo a código o lenguaje de máquina, el único que la computadora es capaz de entender y ejecutar. El encargado de realizar esta función es un programa traductor (compilador o intérprete). Si tras la compilación se presentan errores (conocidos como "errores de compilación") es preciso, entonces, modificar el código del programa de forma que éste se adapte a las reglas de sintaxis del lenguaje elegido, con el fin de corregir los errores. Este proceso continúa hasta que ya no se producen errores, con lo que el compilador proporciona uno o más programas "objeto", aún no ejecutables directamente. A continuación, se procede al enlace (*link*) de los diferentes objetos producidos por el compilador para generar, finalmente, el programa ejecutable. Una vez ejecutado, y suponiendo que no hay errores durante su ejecución (llamados errores de "tiempo de ejecución"), se obtendrá la salida de los resultados del programa.

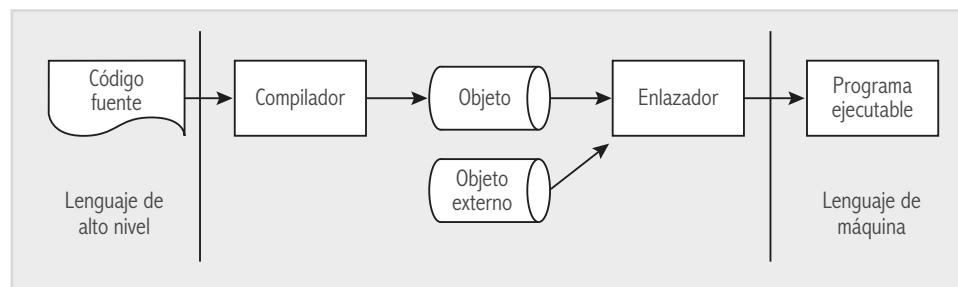


Fig. 1-3. Aplicación del programa traductor (compilador o intérprete).

1.8.5 Verificación y depuración

La verificación y la depuración son procesos sucesivos mediante los que se comprueba un programa con una amplia variedad de datos de entrada ("datos de prueba"), para determinar si hay errores, identificarlos y corregirlos. Para realizar la verificación y la depuración se debe desarrollar una amplia gama de datos de prueba: valores normales de entrada, valores extremos, valores con salida o resultado conocido y que comprueban aspectos esenciales del programa. Suele ser de gran utilidad realizar un seguimiento de los cambios de estados en los sucesivos pasos de ejecución del programa utilizando alguna herramienta auxiliar (por ejemplo, un depurador).

Todo depende de la complejidad del programa en cuestión. En algunos casos es suficiente llevar a cabo pruebas de "caja negra"; esto es, establecer valores de entrada y comprobar que las salidas sean las esperadas, sin considerar el comportamiento interno del algoritmo.

Los errores más difíciles de detectar y, por lo tanto, los más peligrosos, son los de la lógica del programa, debidos a un mal diseño de ésta. No saltan a la vista como errores de compilación ni de ejecución y sólo pueden advertirse por la obtención de resultados incorrectos –cuando se conoce el resultado exacto de los cálculos o se tiene idea del orden de magnitud de ellos–, por lo que la importancia de esta fase en la vida de un programa resulta crucial, antes de pasarlo de manera definitiva a una fase de producción.

Una vez detectados estos errores hay que volver a rediseñar el algoritmo, codificar y compilarlo, para obtener el código ejecutable correcto.

En 1998 la sonda espacial *Mars Climate Orbiter* fue lanzada al espacio con el propósito de estudiar las características climáticas del planeta Marte. Sin embargo, al entrar en órbita, una falla en la navegación produjo que la sonda alcanzara una altitud peligrosamente baja, lo que la destruyó contra la atmósfera marciana. El error: parte del software se construyó usando el sistema de medidas norteamericano y otra parte, utilizando el sistema inglés. Éste es un buen testimonio de una verificación inadecuada.

Errores de tiempo de ejecución:
Intento de división por cero, raíces cuadradas de números negativos, intentos de apertura de archivos inexistentes, dispositivos periféricos no conectados, etc.

1.8.6 Documentación.

La documentación de un programa se clasifica en interna y externa. La primera se incluye en el código del programa mediante comentarios que ayudan a comprenderlo. Para funcionar el programa no necesita la existencia de comentarios; más aún, los comentarios no generan código ejecutable cuando se traducen a código de máquina. Su única finalidad es hacer que los programas sean más fáciles de leer y comprender, sobre todo para aquellas personas ajenas a ellos, e incluso para los propios programadores. En lenguaje C, los comentarios son multilínea y se delimitan por /* y */. Por ejemplo:

```
/* Este es un comentario en C */
```

En Pascal los comentarios también son multilínea y se delimitan con los caracteres { y }. Por ejemplo:

```
{ Este es un comentario en Pascal }
```

El objetivo del programador es escribir códigos sencillos y limpios. La documentación interna también se complementa con la utilización de identificadores, que indiquen o se adecuen mejor a la finalidad de los distintos componentes dentro del programa.

La importancia de una documentación interna adecuada se manifiesta, todavía en mayor grado, cuando los programas son complejos y tienen cientos o miles de líneas y es muy difícil su seguimiento.

La documentación es vital cuando se desean corregir posibles errores o bien modificar el programa. Después de cada cambio la documentación debe actualizarse para facilitar cambios posteriores. Es práctica corriente numerar las versiones sucesivas del programa, así como indicar el nombre de los sucesivos programadores que intervinieron tanto en la concepción del programa inicial como en las distintas modificaciones posteriores.

Asimismo, es conveniente una presentación y una indentación adecuadas en las distintas líneas del programa, así como líneas en blanco que separan los distintos módulos, de forma que éste sea más legible.

Hay que tener en cuenta que para el programa traductor el código fuente es un archivo de texto, esto es, una sucesión de caracteres, y que la traducción a código ejecutable es independiente de su presentación y formato, pero no así para la persona que tiene que leer el código o modificarlo. Así, por ejemplo, el código podría ser procesado por el compilador sin inconvenientes:

```
int main()
{
    int x; float y; printf("ingrese un número: \n");
    scanf("%d", &x); y = 4.1888*x*x*x; printf("\nResultado: %d", y);
}
```

Sin embargo, es difícil de leer y provoca mayor fatiga en su seguimiento y comprensión.

La documentación externa, por su parte, debe incluir:

1. Listado del programa fuente, en el que se incluyan mapas de memoria, referencias cruzadas y cualquier otra cosa que se obtenga en el proceso de compilación.
2. Explicación de cualquier fórmula o cálculos y expresiones complejas en el programa.
3. Especificación de datos y formatos de pantalla para su entrada y salida, así como cuantas consideraciones se estimen oportunas para mejorar la eficiencia del programa.



Un identificador es una combinación de caracteres alfabéticos y dígitos que se utiliza para poner nombre a los distintos componentes del programa (variables, constantes, funciones, tipos de datos, etc.).



Con indentación nos referimos a la sangría o margen que dejamos al codificar el programa y que permite que éste sea más inteligible.

En general, la documentación externa se compone de un manual de instalación, un manual de usuario y un manual de mantenimiento, sobre todo en grandes aplicaciones. En un proyecto grande cada uno de estos manuales abarca varios volúmenes que es necesario leer para conocer su origen, el desarrollo y la evolución de sus distintos componentes.

1.9 Estructura de un programa en C.

Un programa sencillo escrito en C puede presentar, entre otras, las siguientes partes:

- Directivas al preprocesador.
- Prototipos de funciones.
- La función `main()`.
- Las definiciones de las demás funciones elaboradas por el programador.

1.9.1 Directivas al preprocesador.

El preprocesamiento es una etapa previa a la compilación, en la que se toma el código escrito por el programador y se lo transforma en un nuevo código, más conveniente para el compilador. Esta traducción se realiza de acuerdo con palabras especiales que el programador inserta en su código, denominadas directivas al preprocesador. Éstas tienen la siguiente forma:

```
#<nombre de la directiva>
```

Por ejemplo:

```
#include <stdio.h>
#define MAX 30
```

La primera indica al preprocesador que debe incluir el archivo de cabecera (*header*) `stdio.h` mientras que la segunda directiva indica al preprocesador que toda vez que encuentre la palabra `MAX` en el código fuente, la reemplace por el número 30.

1.9.2 Prototipos de funciones.

Cuando se escribe un programa, si éste no es demasiado sencillo, en general se lo divide en una función principal y otras auxiliares. Durante el proceso de compilación, se requiere que cada función esté definida **antes** de cualquier llamada a ella, ya sea con su implementación completa o sólo con su nombre, parámetros recibidos y tipo de dato returnedo, lo que se conoce como el prototipo de la función. Como ejemplo, tenemos:

```
float f1(int, char);           /* prototipo 1*/
char f2(float, int, float);   /* prototipo 2*/
void f3();                     /* prototipo 3*/
int main()                     /* programa principal */
{
    ...
}
```

Acá se observa un programa C con su función `main()` y, antes, la declaración de tres prototipos de funciones, `f1()`, `f2()` y `f3()`. Cada declaración comienza indicando el tipo de dato returnedo por la función, luego su nombre y, entre paréntesis, la lista de tipos de parámetros



En la actualidad hay herramientas denominadas controladores de versiones que permiten gestionar el código fuente de los programas junto con su documentación, de manera que sea más disciplinado el trabajo que apliquen distintos programadores sobre el software. Además, facilitan el proceso de documentación interna, por ejemplo, agregando un encabezado a cada archivo de código con una descripción de él, y un detalle de las modificaciones aplicadas y el nombre de los autores de éstas. Algunos ejemplos de estas herramientas son CVS (*Concurrent Versions System*) y SVN (*Subversion*).

recibidos. Por ahora no avanzaremos sobre el conocimiento de las funciones –salvo la función `main()`–, sino hasta unos capítulos más adelante.

1.9.3 La función `main()`.

Esta función es un tanto “especial”, en cuanto que es el “punto de partida” por el cual comienza la ejecución del programa. A partir de aquí podrán invocarse otras funciones. Además, como parte del cuerpo de la función `main()`, se pueden encontrar:

- Declaración de constantes (por lo general estas declaraciones se hacen fuera del cuerpo del `main()` para que sean accesibles a otras funciones).
- Declaración de tipos (estas declaraciones suelen hacerse fuera del cuerpo del `main()` para que sean accesibles a otras funciones).
- Declaración de variables (son las variables locales del `main()`).
- Sentencias (instrucciones) que conforman la lógica de la función `main()`.

No siempre es necesario realizar todas las declaraciones.



La función `main()` comprende:

- Declaración de constantes.
- Declaración de tipos.
- Declaración de variables.
- Sentencias (instrucciones).

1.9.4 Declaración de constantes.

Muchas veces puede ser útil emplear en varias oportunidades dentro de un programa un valor numérico o un texto determinado. Para evitar tener que definirlo en cada punto en que se utiliza, emplearemos una referencia única en todo el programa, a la que llamaremos constante. Las constantes son inalterables (de ahí su nombre), en general se las nombra usando mayúsculas y se declaran como se muestra a continuación:

```
const <tipo de dato> <nombre de la constante> = <valor>;
```

Por ejemplo:

```
const float PI = 3.14;
const char *MENSAJE = "Buenos días!";
```

También podríamos declarar una constante utilizando un tipo de directivas al procesador:

```
#define PI 3.14
```

En este caso, cada vez que el preprocesador reconozca la palabra `PI` en el código, la reemplazará por el número 3,14.

1.9.5 Declaración de tipos y variables.

Los diferentes objetos de información con los que trabaja un programa C se conocen en conjunto, como datos. Todos los datos tienen un tipo asociado. Un dato puede ser un simple carácter, un valor entero o un número decimal, entre otros.

Lenguajes como C o Pascal son conocidos como fuertemente tipados (*strongly-typed*), en cuanto a que es obligatorio para el programador asignar un tipo determinado a cada dato procesado. La asignación de tipos a los datos tiene dos objetivos principales:

- Detectar errores de operaciones en programas (por ejemplo, si una operación matemática requiere números enteros, el uso de números decimales sería un error que, eventualmente, podría detectar el compilador).
- Determinar cómo ejecutar las operaciones (por ejemplo, en lenguajes como Pascal el operador “+” significa sumar si los operandos son numéricos; en cambio, permite concatenar si los operandos son cadenas de caracteres).

Los datos son almacenados en la memoria de la computadora, en una posición determinada de ella, representada por una dirección de memoria. A nivel del programa (y del programador), en esta posición de memoria puede escribirse o leerse un dato de acuerdo con el tipo especificado (por ejemplo, podríamos indicar que la posición 1 000 de la memoria almacena un byte que representa un carácter). Sin embargo, debido a la incomodidad de utilizar direcciones para acceder a los datos en memoria, en lenguajes de alto nivel (como C o Pascal) se utilizan nombres (o “etiquetas”) que están asociados con esas posiciones. A estas etiquetas, que permiten acceder a una posición determinada de memoria, tanto para escritura como para lectura, se las denomina variables. En C, una variable se declara indicando el tipo de dato que alojará y su nombre:

```
<tipo de dato> <nombre de la variable>;
```

Por ejemplo:

```
int numero_empleado; /* número de empleado */
float horas;           /* horas trabajadas */
int edad;              /* edad del empleado */
char apellidos[30];   /* apellidos del empleado */
```

Todas las variables de un programa C deben declararse antes de utilizarse, y es una buena práctica de programación utilizar nombres representativos para ese fin que sugieran lo que las variables representan (esto hace que el programa sea más legible y fácil de comprender). También es buena práctica incluir comentarios breves que indiquen cómo se utiliza la variable.

El tipo de un dato determina la naturaleza del conjunto de valores que puede tomar una variable. Dentro de la computadora, los datos (de acuerdo con el tipo) se representan de alguna manera en particular. Por ejemplo, un dato de tipo carácter (asumiendo que se utiliza el estándar ASCII de representación de caracteres), se representa como una secuencia de 8 bits (un byte), un número entero con 32 bits (4 bytes), un número decimal de precisión simple también con 4 bytes, entre otros. Cada bit puede tomar los valores 0 o 1, y la cantidad de bits que se requieren para representar un dato implica cuánta memoria de computadora se necesita para ese fin.

Los tipos de datos básicos del lenguaje C son los siguientes:

Tabla 1-1 - Tipos de datos básicos del lenguaje C

Tipo	Tamaño* (bytes)	Descripción del tipo
char	1	Carácter o entero de un byte
int	4	Entero (con signo)
signed int	4	Entero (con signo)
unsigned int	4	Entero (sin signo)
short	2	Entero corto (con signo)
signed short	2	Entero corto (con signo)
unsigned short	2	Entero corto (sin signo)
long	8	Entero largo (con signo)
signed long	8	Entero largo (con signo)
unsigned long	8	Entero largo (sin signo)
float	4	Decimal de simple precisión
double	8	Decimal de doble precisión

* Los tamaños de los tipos de datos dependen de la arquitectura (procesador). En este caso se considera una arquitectura de 32 bits.



ASCII

American Standard Code for Information Interchange. Estándar para codificación de caracteres, creado en 1963, de uso muy frecuente en el manejo de texto en computadoras y redes.



La función `main()` es el punto de partida por donde el sistema operativo busca su primera instrucción ejecutable.

1.9.6 Lógica de la función principal.

Todo programa ejecutable en lenguaje C debe tener una función llamada `main()`, que es el punto de partida por donde el sistema operativo busca su primera instrucción ejecutable. Esta función puede ser un único programa, o bien un programa puede tener también otras funciones (módulos) que colaboran con `main()`. De acuerdo con el paradigma de programación estructurada, en toda función C –incluida la función `main()`– pueden encontrarse, entre otras cosas, sentencias que indiquen la realización de tres tipos de acciones: secuencia, selección o iteración.

1.9.6.1 Secuenciales.

Son las asignaciones y las invocaciones a funciones. La asignación se hace de la siguiente forma:

```
variable = expresión;
```

Por ejemplo:

```
num1 = 15;
num2 = 5;
resultado = suma(num1,num2); /* llamada a función */
resultado = resultado / 2;
...
```

Las llamadas a funciones se verán en capítulos posteriores.



Las estructuras selectivas se utilizan para tomar decisiones lógicas.

1.9.6.2 Selección.

Las estructuras selectivas se utilizan para tomar decisiones lógicas y existen en dos “sabores”: la sentencia `if-then-else` y la sentencia `switch`.

La sentencia `if-then-else` se considera de alternativa doble (si se cumple cierta condición, entonces..., en caso contrario...), y tiene la siguiente estructura:

```
if (condición)
    acción 1;
else
    acción 2;
```

En lenguaje C, cuando alguna de las alternativas tiene más de una instrucción, esa opción deberá llevar una llave de apertura ‘{’ y otra de cierre ‘}’. Por ejemplo:

```
if (condición)
{
    acción 1;
    acción 2;
    acción 3;
}
else
{
    acción 4;
    acción 5;
}
```

La sentencia `switch`, en cambio, se considera de selección múltiple, ya que el flujo de ejecución puede continuar por una cantidad N de alternativas posibles, según el valor de la expresión que se evalúa al principio:

```

switch (expresión)
{
    case Opción1: < acción 1>;
                    break;

    case Opción2: < acción 2>;
                    break;

    ...

    case Opción N: < acción N>;
                     break;

    default:       < acción M>;
}

```

1.9.6.3 Iteración o repetición.

Las estructuras repetitivas (llamadas también bucles o ciclos) se utilizan para realizar varias veces el mismo conjunto de operaciones. Entre ellas se encuentran aquellas donde la cantidad de repeticiones se conoce *a priori* y aquellas en las que las repeticiones se realizan hasta que se cumple una condición lógica dada. En lenguajes como C y Pascal hay tres tipos de estructuras iterativas: `for`, `while` y `do-while`.

La estructura `for` permite definir un bucle controlado por un contador, denominado variable de control o de inducción. La sintaxis es:

```

for ( ... ; ... ; ... )
{
    < acción 1>
    < acción 2>
    ...
}

```

El encabezado de un bucle `for` tiene tres partes (separadas por ";"). En la primera se inicializa la variable de control y sólo se ejecuta una vez, antes de la primera iteración. La segunda es la condición lógica que debe cumplirse para que la próxima iteración se ejecute; esta condición se evalúa antes de cada iteración y, cuando deja de satisfacerse, el bucle `for` termina. La tercera parte del encabezado es la actualización de la variable de control y se ejecuta después de cada iteración. Para que esto quede más claro, a continuación se presenta un ejemplo simple de un bucle `for` que acumula en una variable la suma de los primeros 10 números naturales (siendo `i` la variable de control):

```

acum = 0;
for (i = 1; i <= 10; i = i+1 )
{
    acum = acum + i;
}

```

En C hay formas más sofisticadas de utilizar el ciclo `for`, que podremos ver a medida que avancemos a lo largo del libro.



Las estructuras repetitivas se utilizan para realizar varias veces el mismo conjunto de operaciones.

La estructura de control `while`, por su parte, evalúa la condición lógica antes de comenzar cada iteración. Si ésta es verdadera (esto es, se satisface), entonces se ejecuta el cuerpo de la estructura `while`. En caso contrario, el bucle `while` termina. La sintaxis es la siguiente:

```
while (condición)
{
    < acción 1>
    < acción 2>
    ...
}
```

Por ejemplo, la acumulación de los primeros 10 números naturales puede llevarse a cabo usando una construcción `while`, como se muestra a continuación:

```
acum = 0;
i = 1;
while (i <= 10)
{
    acum = acum + i;
    i++;
}
```

La sentencia `i++` en el ejemplo anterior es una forma abreviada, provista por el lenguaje C, de realizar el incremento `i = i+1`.

Por último, un bucle `do-while` se utiliza cuando se quiere asegurar que el ciclo se ejecuta al menos una vez, puesto que la evaluación de la condición lógica se hace al final de éste. Tiene la forma siguiente:

```
do
{
    < acción 1>
    < acción 2>
    ...
}
while (condición);
```

El mismo ejemplo anterior pero utilizando una construcción `do-while`, podría escribirse como se muestra a continuación:

```
acum = 0;
i = 1;
do
{
    acum = acum + i;
    i++; /* i = i+1 */
}
while (i <= 10);
```

1.9.6.4 Sentencias de salto.

En C hay sentencias para evitar la ejecución estrictamente secuencial del programa. En general (sentencias `goto`, `break` y `continue`) no deben utilizarse para elaborar programas estructurados, aunque pueden ser útiles si se justifica en forma apropiada su aplicación en un código.

```
return <expresión>;
```

Se usa para devolver el control del flujo de ejecución desde una función, siendo `<expresión>` el valor (dato) retornado por ella.

```
goto <etiqueta>;
...
<etiqueta>: <acción>
```

La sentencia `goto` permite efectuar un salto incondicional hasta otro punto del programa, indicado por una etiqueta. Este recurso que ofrece C no debería utilizarse en programación de alto nivel ni cuando se pretenda construir código reusable y de acuerdo con el paradigma de programación estructurada. Para una justificación clara y contundente al respecto, le recomendamos al lector (una vez avanzado y familiarizado con la programación estructurada) leer el artículo "*Go To Considered Harmful*" publicado por Edsger Dijkstra (*Letter to Communications of the ACM*, 1968).

```
break;
```

Ya se vio su uso en la construcción `switch`. También se puede usar en un bloque repetitivo. El efecto es la terminación del ciclo (`switch`, `for`, `while`, `do-while`) y la próxima sentencia a ejecutar es la que sigue al bloque.

```
continue;
```

Se puede usar en un bloque repetitivo (`for`, `while`, `do-while`) para forzar una nueva iteración del ciclo, ignorando las sentencias que están a partir de `continue` y hasta el fin del ciclo.

1.10 Poniendo todo junto.

A continuación se presenta un ejemplo simple escrito en lenguaje C. Este programa tiene como única funcionalidad mostrar en pantalla un menú de tres opciones. Si el usuario selecciona la opción "1", se muestra la leyenda "Este es un mensaje!"; si selecciona la opción "2", se muestra la leyenda "Este es otro mensaje!"; y si selecciona la opción "3", el programa finaliza su ejecución.

```
#include <stdio.h>

/* Constantes */
#define OPCION_1          1
#define OPCION_2          2
#define OPCION_SALIR      3

/* Prototipos de funciones */
void mostrar_menu();
int leer_opcion();
void ejecutar(int);
/* Programa Principal */
int main()
{
```



La sentencia `goto` permite efectuar un salto incondicional hasta otro punto del programa. No debería utilizarse en programación de alto nivel ni cuando se pretenda construir código reusable y de acuerdo con el paradigma de programación estructurada.



En la Web de apoyo, encontrará el vínculo a la página Web personal de Edsger Dijkstra.

```

/* Declaración de la variable 'opcion' */
int opcion;

do
{
    mostrar_menu();
    opcion = leer_opcion();
    ejecutar(opcion);
}
while (opcion != OPCION_SALIR);

return 0;
}

/* Implementación de funciones */

void mostrar_menu()
{
    printf("Elija una opción: \n");
    printf(" 1 - Mostrar un mensaje \n");
    printf(" 2 - Mostrar otro mensaje \n");
    printf(" 3 - Salir \n");
}

int leer_opcion()
{
    int opcion;
    /* Se lee un número entero por teclado */
    scanf("%d", &opcion);
    return opcion;
}

void ejecutar(int opcion)
{
    switch (opcion)
    {
        case OPCION_1: printf("Este es un mensaje! \n");
                        break;
        case OPCION_2: printf("Este es otro mensaje! \n");
                        break;
        case OPCION_SALIR: printf("Saliendo... \n");
                            break;
        default: printf("Opción incorrecta! \n");
    }
}

```

 La palabra `void` no es un tipo de dato en sí y sirve para indicar, en este caso, las funciones `mostrar_menu()` y `ejecutar()`, que no retornan valor alguno.

En primer lugar, mediante la directiva al preprocesador `#include`, se incluye el archivo de cabecera `stdio.h` (provisto con el compilador de C) que permite, entre otras cosas, tener acceso al teclado y a la pantalla (periféricos de entrada y salida, respectivamente). Luego, usando la directiva `#define`, se definen tres constantes.

En la declaración de prototipos de funciones, es interesante observar la palabra reservada `void`. Ésta no es un tipo de dato en sí y sirve para indicar que, en este caso, las funciones `mostrar_menu()` y `ejecutar()` no retornan valor alguno.

En el programa principal se declara una variable local de tipo entero, llamada `opcion` (la que almacenará la opción ingresada a través del teclado por el usuario). La parte más interesante de la función `main()` es una estructura repetitiva `do-while` que, como puede observarse, hace tres cosas bien definidas: muestra un menú, lee la opción ingresada por el usuario desde el teclado y ejecuta una acción según la opción (todo esto mientras la variable `opcion` no tome el valor definido en `OPCION_SALIR`).

Por último, luego del programa principal se implementan las funciones auxiliares. Una de ellas, la función `ejecutar()`, recibe un parámetro de tipo entero llamado `opcion`, que controla el comportamiento de la estructura `switch`.

Además de las llamadas a las funciones auxiliares, también pueden encontrarse llamadas a otras funciones que no están implementadas en este programa (o, al menos, eso parece). Se trata de las funciones `printf()` y `scanf()`, provistas con el compilador (razón por la cual se incluye el archivo de cabecera `stdio.h`).

1.11 Estructura de un programa en Pascal.

En el caso de Pascal, la estructura de un programa es, en gran medida, similar a la de uno escrito en C y, principalmente, puede contener:

- Prototipos de funciones.
- El programa principal (equivalente a la función `main()` de C).
- Las definiciones de las demás funciones elaboradas por el programador.

1.11.1 El programa principal.

El programa principal en Pascal equivale a la función `main()` del lenguaje C, y es el punto a partir del que comienza a ejecutarse. Sin embargo, a diferencia de C, no toma la forma de una función. Las constantes y variables utilizadas por el programa principal se declaran fuera de él (a nivel global). Por lo general busca evitarse esta práctica (por razones que quedarán más claras a lo largo del libro), las constantes y las variables se declaran en forma local a cada función o procedimiento que las utilice.

En el caso de los tipos de datos, en Pascal encontramos los siguientes:

Tabla 1-2 - Tipos de datos en lenguaje Pascal

Tipo	Tamaño* (bytes)	Descripción del tipo
<code>char</code>	1	Carácter
<code>integer</code>	2 o 4	Entero (con signo)
<code>word</code>	2	Entero (sin signo)
<code>shortint</code>	1	Entero corto (con signo)
<code>byte</code>	1	Entero corto (sin signo)
<code>longint</code>	4	Entero largo (con signo)
<code>longword</code>	4	Entero largo (sin signo)
<code>single</code>	4	Decimal de simple precisión
<code>real</code>	8	Decimal de doble precisión
<code>double</code>	8	Decimal de doble precisión
<code>boolean</code>	1	Valor booleano (verdadero o falso)
<code>string</code>	máximo 255	Cadena de caracteres

* Se considera una arquitectura de 32 bits. También pueden observarse diferencias de acuerdo con el compilador usado.



La estructura de un programa en Pascal puede contener:

- Prototipos de funciones.
- El programa principal.
- Las definiciones de las demás funciones elaboradas por el programador.



El programa principal también puede tener variables locales.

En Pascal, la declaración de variables es muy similar a la de C, y es necesario indicar el nombre y el tipo de dato de ella, anteponiendo la palabra reservada var (para indicar el comienzo de la sección de declaración de variables):

```
var <nombre de la variable>: <tipo de dato>;
```

Por ejemplo:

```
var numero_empleado: integer;      { número de empleado }
    horas: real;                  { horas trabajadas }
    edad: integer;                { edad del empleado }
    apellidos: string[30];        { apellidos del empleado }
```



Un procedimiento es un módulo similar a una función, pero que no retorna valor alguno como resultado de su ejecución. Equivale al uso de funciones con tipo void en lenguaje C.

Por su parte, las constantes se declaran utilizando la palabra reservada const, como se muestra a continuación:

```
const PI = 3.14;
```

Con respecto a la modularización de un programa Pascal, esto puede llevarse a cabo de las siguientes maneras:

- Mediante la construcción de funciones, como en el caso de C.
- Mediante la construcción de procedimientos.

Un procedimiento es un módulo similar a una función, pero que no retorna valor alguno como resultado de su ejecución. Básicamente, equivale al uso de funciones con tipo void en lenguaje C.

En cuanto a las estructuras de control, como Pascal es un lenguaje orientado a la programación estructurada, brinda soporte para las construcciones secuenciales, selectivas y repetitivas, como en el caso de C. Las diferencias son sólo sintácticas, como se mostrará a continuación.

1.11.1.1 Secuenciales.

Básicamente, nos referimos a una secuencia de instrucciones donde no se observa estructura de control alguna que pudiese modificar el sentido o la dirección del flujo de ejecución, como se muestra en el ejemplo siguiente:

```
num1 := 15;
num2 := 5;
resultado := suma(num1,num2);      { llamada a función }
resultado := resultado div 2;
...
```

1.11.1.2 Selección.

La sentencia if-then-else, en el caso de Pascal, toma la forma que sigue:

```
if (condición) then
begin
    acción 1;
    acción 2;
    acción 3
end
else
```

```
begin
    acción 4;
    acción 5;
end;
```

La construcción de selección múltiple switch se denomina **case-of** en el caso de Pascal, aunque no hay diferencias conceptuales entre ambas. La sintaxis de la sentencia case-of es la siguiente:

```
case (expresión) of
{
    Opción1: < acción 1>;
    Opción2: < acción 2>;
    ...
    else      < acción M>;
}
```

1.11.1.3 Iteración o repetición.

El bucle **for** en Pascal es de construcción más simple (y, por lo general, más clara) dado que no permite tantos “grados de libertad” como sucede en C. También está controlado por una variable de inducción que puede tomar un rango de valores ordinales, de manera ascendente o descendente. En el ejemplo que sigue se muestran los números enteros desde 1 hasta 10; **i** es la variable de control:

```
acum := 0;
for i:=1 to 10 do
begin
    acum := acum + i;
end;
```

Es importante notar que la variable de inducción es manejada en forma automática por la sentencia **for**. En el ejemplo anterior, se incrementa en una unidad luego de cada iteración.

La sentencia **while** de Pascal se comporta de la misma manera que la correspondiente al lenguaje C.

```
acum := 0;
i := 1;
while (i <= 10) do
begin
    acum := acum + i;
    i := i+1;
end;
```

Por último, lo que en C se conoce como **do-while**, en Pascal se denomina **repeat-until**, aunque no se observen grandes diferencias entre ambas. Lo más importante para destacar, en este caso, es que la estructura **repeat-until** de Pascal itera hasta que la condición se cumpla



Lo que en C se conoce como do-while, en Pascal se denomina repeat-until.

(mientras que el do-while de C itera mientras la condición lógica se cumpla). Veamos el ejemplo siguiente:

```

acum := 0;
i := 1;
repeat
begin
    acum := acum + i;
    i := i+1;
end
until (i > 10);

```

1.12 Ahora, integrando.....

A continuación se reproduce el mismo ejemplo integrador expuesto antes para el lenguaje C pero, en esta oportunidad, escrito en lenguaje Pascal, utilizando las estructuras que se acaban de mencionar.

```

program ejemplo_integrador;

uses crt; {Equivale a stdio.h de C}

{Constantes}
const OPCION_1 =      1;
const OPCION_2 =      2;
const OPCION_SALIR =  3;

{Prototipos}
procedure mostrar_menu; forward;
function leer_opcion: integer; forward;
procedure ejecutar(opcion: integer); forward;

{ Implementación de funciones }

procedure mostrar_menu;
begin
    writeln('Elija una opción:');
    writeln(' 1 - Mostrar un mensaje');
    writeln(' 2 - Mostrar otro mensaje');
    writeln(' 3 - Salir');
end;

function leer_opcion: integer;
{Variable local a la función}
var opcion: integer;
begin
    { Se lee un número entero por teclado }
    readln(opcion);
    leer_opcion := opcion;
end;

```

```

procedure ejecutar(opcion: integer);
begin
  case (opcion) of
    OPCION_1:      writeln('Este es un mensaje!');
    OPCION_2:      writeln('Este es otro mensaje!');
    OPCION_SALIR: writeln('Saliendo... ');
    else           writeln('Opción incorrecta!');
  end;
end;

{Variables del programa principal}
var opcion: integer;

{Programa Principal}
begin
  repeat
    begin
      mostrar_menu();
      opcion := leer_opcion();
      ejecutar(opcion);
    end
  until (opcion = OPCION_SALIR);
end.

```

En este punto, es un buen ejercicio para el lector hacer una comparación entre este programa y el expuesto antes y sacar la mayor cantidad de conclusiones posibles, como las que siguen:

- Que un programa Pascal empieza con un encabezado de la forma:
`program <nombre del programa>.`
- Que las implementaciones de procedimientos y funciones deben encontrarse luego de los prototipos pero antes de que se los llame.
- Que la entrada de datos desde teclado se lleva a cabo con la función `readln()`, y la salida por pantalla mediante el procedimiento `writeln()` (ambos pertenecientes a `crt`, que se incluye al principio del programa).
- Que la asignación de un valor en una variable se realiza usando el operador `:=`.

Aquí finaliza el capítulo introductorio. Como tal, no se pretende cargar de nueva información al lector sino, simplemente, echar un vistazo amplio y general sobre las cuestiones que se desarrollarán más en detalle a continuación.

1.13 Resumen

Una computadora electrónica es una máquina cuya finalidad es procesar información para obtener resultados. Los datos que constituyen la entrada (*input*) se procesan mediante una lógica (programa) para producir una salida (*output*).

Un programa de computadora es la materialización de un algoritmo y está conformado por un conjunto de pasos que, ejecutados de la manera correcta, permiten obtener un resultado en un tiempo acotado. El programa se implementa mediante un lenguaje de programación (por

ejemplo, Pascal, C, Cobol, Java) y debe ajustarse a un paradigma de programación, entre los que se pueden destacar el estructurado y el orientado a objetos.

La programación estructurada, cuyas bases se sentaron a mediados de la década de 1960, predica que un programa de computadora puede escribirse utilizando sólo tres tipos de estructuras de control: secuenciales, selectivas y repetitivas. Para que esto pueda aplicarse, el programa debe ser propio, lo que significa que posee un solo punto de entrada y un solo punto de salida, que hay caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas las partes del programa, y que todas las instrucciones son ejecutables y no existen lazos o bucles infinitos.

Con ulterioridad, Edsger Dijkstra introdujo el concepto de modularidad, por el cual todo programa puede implementarse como integración de subprogramas más pequeños y manejables y que interactúan entre ellos.

Por último, en el proceso de desarrollo de programas de computadora (software) pueden distinguirse diferentes etapas:

- Análisis del problema.
- Diseño del algoritmo.
- Codificación.
- Compilación y ejecución.
- Verificación y depuración.
- Documentación.

El objetivo de este libro es capacitar al lector, en su función de programador, para la construcción de programas de computadora correctos y eficientes, mediante la utilización de los lenguajes de programación estructurada C y Pascal.

1.14 Contenido de la página Web de apoyo.....



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Autoevaluación.



Video explicativo (02:25 minutos aprox.).



Evaluaciones propuestas.*



Presentaciones. *

2

Datos y sentencias simples Operaciones de entrada/salida

Contenido

2.1 Introducción	26
2.2 Tipos de datos simples	26
2.3 <i>Little endian vs. big endian</i>	28
2.4 Modificadores de tipos en C	28
2.5 Palabra reservada <code>void</code>	29
2.6 Otros modificadores	29
2.7 Tipos de datos definidos por el usuario	30
2.8 Construcción de sentencias básicas	31
2.9 Operadores	32
2.10 Operaciones de entrada/salida	37
2.11 Resumen.....	41
2.12 Problemas propuestos.....	42
2.13 Problemas resueltos.....	43
2.14 Contenido de la página Web de apoyo.....	55

Objetivos

- Introducir a las estructuras de datos y al uso de sentencias simples para su manipulación, lo que constituye el puntapié inicial de cualquier curso de algoritmia o programación.
- Brindar un panorama amplio y detallado de estructuras y sentencias básicas.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

2.1 Introducción.

Durante su ejecución un programa procesa y manipula datos (en mayor o menor medida), que residen en la memoria de la computadora, como se comentó en el capítulo anterior. Los datos son la “materia prima” con la que se alimenta un programa para generar los resultados; son entidades concretas y mensurables en términos del espacio que ocupan y del valor que representan, como lo pueden ser números, letras o cadenas de caracteres.

Todos los datos se ajustan a un “molde” que define la “forma” del dato (su tamaño, cómo se organizan los bits y los bytes para su representación, etc.). Asimismo, nos referimos a este molde como el tipo del dato. De esta manera, todos los datos pertenecen a un tipo determinado, lo que permite definir al dato concreto como un número entero, un número real, un carácter, un entero largo, etc.

Lenguajes como C o Pascal son fuertemente tipados, en cuanto a que es obligatorio para el programador asignar un tipo determinado a cada dato procesado, lo que permite detectar errores de operaciones en programas y, no menos importante, determinar cómo ejecutar las operaciones sobre los datos.

Para terminar de aclarar la diferencia entre el tipo del dato y el dato en sí, podemos afirmar que el tipo es para el dato lo mismo que un molde es para un pastel. El molde no se come, pero es el que da la forma y define al pastel. Además, con un solo molde pueden cocinarse varios pasteles. De igual forma, un tipo de dato permite crear varias instancias concretas de ese tipo.

2.2 Tipos de datos simples.

Los tipos de datos simples son el conjunto básico de tipos provistos por el lenguaje y que, eventualmente, permiten construir tipos de datos más complejos. Todos los lenguajes de programación brindan un conjunto más o menos rico de tipos de datos predeterminados, como los tipos enteros, reales, los caracteres, entre otros.

En algunos casos los tipos de datos suelen variar, para el mismo lenguaje, entre computadoras diferentes. Éste es el caso del tipo `int` (entero) en lenguaje C, que está definido como el tamaño “natural” de la arquitectura de base. Así, en un procesador de 32 bits, el tipo `int` ocupa 32 bits (4 bytes), mientras que en una computadora de 64 bits, ocupa 8 bytes. Sin embargo, en general se puede afirmar que los rangos habituales son los siguientes:

Tabla 2-1 - Tipos de datos y rangos habituales en lenguaje C.

<i>Tipo</i>	<i>Tamaño (bytes)</i>	<i>Rango de valores</i>
<code>char</code>	1	0 ... 255
<code>int</code>	4	-2 147 483 648 ... 2 147 483 647
<code>signed int</code>	4	-2 147 483 648 ... 2 147 483 647
<code>unsigned int</code>	4	0 ... 4 294 967 295
<code>short int</code>	2	-32 768 ... 32 767
<code>signed short int</code>	2	-32 768 ... 32 767
<code>unsigned short int</code>	2	0 ... 65 535
<code>long int</code>	8	-2 147 483 648 ... 2 147 483 647
<code>signed long int</code>	8	-2 147 483 648 ... 2 147 483 647
<code>unsigned long int</code>	8	0 ... 4 294 967 295
<code>float</code>	4	1,17E-38 ... 3,40E+38
<code>double</code>	8	2,22E-308 ... 1,79E+308

El formato de los tipos `float` y `double` está definido por el estándar IEEE 754 para representación de números en coma flotante.

Por su parte, el tipo `unsigned char` es algo “especial” en cuanto que en él, como rango de posibles valores, se definen los números enteros desde 0 hasta 255 y no un rango de caracteres o letras, como se esperaría. Esto se debe a que un elemento de tipo `char` almacena el código (entero de un byte) que representa, en el estándar ASCII, a cada carácter. Por ejemplo, en este estándar, el carácter ‘A’ se representa con la secuencia de 8 bits (un byte) 01000001 que, en representación decimal, es el número entero 65. Por esta razón es que, en lenguajes como C, el tipo `char` puede utilizarse también como un tipo entero de un byte.

En el lenguaje Pascal, los tipos de datos simples provistos (y sus rangos) son los siguientes (asumiendo un procesador de 32 bits):

Tabla 2-2 - Tipos de datos y rangos habituales en lenguaje Pascal.

<i>Tipo</i>	<i>Tamaño (bytes)</i>	<i>Rango de valores</i>
<code>char</code>	1	Todos los caracteres ASCII
<code>integer</code>	2	-32 768 ... 32 767
<code>word</code>	2	0 ... 65 535
<code>shortint</code>	1	-128 ... 127
<code>byte</code>	1	0 ... 255
<code>longint</code>	4	-2 147 483 648 ... 2 147 483 647
<code>longword</code>	4	0 ... 4 294 967 295
<code>single</code>	4	1,5E-45 ... 3,40E+38
<code>real</code>	8	5,0E-324 ... 1,70E+308
<code>double</code>	8	5,0E-324 ... 1,70E+308
<code>boolean</code>	1	Verdadero o falso
<code>string</code>	máximo 255	Todos los caracteres ASCII

Como diferencias principales podemos observar que Pascal incorpora el tipo `boolean`, para manejo de valores booleanos (verdadero y falso) que, en el caso de C, puede implementarse utilizando una variable de tipo `char` (1 byte), almacenando los valores 0 (para falsedad) y distinto de 0 (para verdad). La otra diferencia importante es que, en el caso de Pascal se considera un tipo básico la cadena de caracteres (`string`), mientras que en C una cadena se construye en forma explícita como una agregación de caracteres.

El rango de posibles valores que puede soportar un tipo de dato no es una característica “caprichosa”, sino que está determinado por la cantidad de bits con la que se codifica un dato. Internamente, un microprocesador (lo que habitualmente se denomina CPU) manipula valores binarios: 0 y 1. Estos dos posibles “estados” se representan, a nivel físico, como dos niveles diferentes de tensión eléctrica (por ejemplo, +5 volts para el valor “1” y -5 volts para el valor “0”). Así, si un tipo de dato es representado por un solo bit, entonces su rango está compuesto por dos valores: 0 y 1. Si, en cambio, se utilizan dos bits para representar un dato, el rango está compuesto por cuatro valores: 00, 01, 10 y 11.

En general, si un tipo de datos se representa con N bits, la cantidad de valores diferentes posible está dada por 2^N . Con este razonamiento, podemos explicar el porqué de los diferentes rangos mostrados anteriormente. Por ejemplo, el tipo `char` se representa con un byte (8-bits), por lo que pueden codificarse $2^8 = 256$ valores diferentes. Por su parte, asumiendo que el tipo entero (`int` en C, `integer` en Pascal) ocupa 4-bytes (32-bits), la cantidad de valores posibles es de $2^{32} = 4\,294\,967\,296$. Esto significa que un tipo entero de 4-bytes puede representar desde el número 0 hasta el número 4 294 967 295 (inclusive). En ocasiones, el primer bit en la codificación

(el bit más significativo o el más a la izquierda) suele utilizarse para representar el signo (+/-) del número en cuestión. Uno representa negativos y/o positivos así, para un tipo entero de 4-bytes **con signo**, podemos representar $2^{31} = 2\ 147\ 483\ 648$ valores diferentes: Desde -2 147 483 648 hasta +2 147 483 647. Con esta introducción básica al concepto de "codificación binaria", se pasará a explicar el concepto de endianness, que tiene que ver con la forma y orden en que se almacenan los bytes que conforman un dato (cuando el dato está compuesto por más de un byte).



Los términos *little-endian* y *big-endian* provienen de la novela Los Viajes de Gulliver, en la que los personajes debatían respecto de por dónde comenzar a comer huevos, si por el lado pequeño o por el lado grande.

2.3 Little endian vs. big endian.

Una de las herencias más espinosas proveniente de los albores de la informática tiene que ver con la manera en que se ordenan los bytes que conforman un dato. Por ejemplo, el número 450 representado como un dato de tipo `int` estaría definido por la siguiente secuencia de 32 bits (4 bytes):

00000000 00000000 00000001 11000010

Internamente, en la memoria de la computadora, estos 4 bytes podrían almacenarse como:

	00000000	00000000	00000001	11000010	
0		1	2	3	

Fig. 2-1. Esquema de almacenamiento de bytes en la memoria de la computadora.

O también podrían almacenarse en el orden opuesto:

	11000010	00000001	00000000	00000000	
0		1	2	3	

Fig. 2-2. Esquema de almacenamiento de bytes en la memoria de la computadora.

El primer esquema, donde el byte más significativo se almacena en la posición más baja de memoria, se conoce como *big-endian*, mientras que el segundo esquema se denomina *little-endian*. El criterio usado no es aleatorio, sino que depende de cada arquitectura. Por ejemplo, los procesadores Intel x86 utilizan el esquema *little-endian*, los Intel Itanium, en cambio, son *big-endian*, y otros procesadores son capaces de soportar ambos esquemas, aunque no en forma simultánea (IBM PowerPC y Sun SPARC, entre otros).

Para la programación de alto nivel, el esquema de ordenamiento de bytes que se utilice en el hardware no debería ser un problema, ya que es algo resuelto por el compilador al transformar el programa en lenguaje de máquina. Sin embargo, en ocasiones podrían generarse situaciones inesperadas si una computadora que usa, por ejemplo, *little-endian* envía datos a través de una red a otra computadora remota que, por su parte, se ajusta al esquema *big-endian*.



El Instituto Nacional Estadounidense de Estándares (ANSI, por sus siglas en inglés: *American National Standards Institute*) es una organización sin fines de lucro que desarrolla estándares para productos, servicios, procesos y sistemas en los Estados Unidos. En la Web de apoyo encontrará un vínculo a la página del instituto.

2.4 Modificadores de tipos en C.

Para ser estrictos, los tipos de datos básicos del lenguaje C son: `char`, `int`, `float` y `double`. Las palabras reservadas `signed`, `unsigned`, `short` y `long` son modificadores que permiten ajustar el tipo de dato al que se aplican. Por ejemplo, si se antepone `unsigned` al tipo `int`, se forma un tipo de dato entero sin signo. Si se aplica el modificador `long`, el tipo entero es largo (mayor rango de valores).

Todos estos modificadores pueden aplicarse a los tipos básicos `int` y `char`. El modificador `long` puede aplicarse también al tipo `double`. El estándar ANSI elimina el tipo `long float` porque es equivalente a `double`.

La diferencia entre los enteros con signo y sin signo es la interpretación del bit más significativo del entero. Si se especifica un entero con signo, el compilador genera código que supone que el bit más significativo se usará para indicar el signo del número: 0 para números mayores o iguales a cero, y 1 para números menores que cero.

2.5 Palabra reservada void.

En el lenguaje C, la palabra reservada `void` no es un tipo en sí, aunque en ciertas circunstancias puede utilizarse como tal. Por ejemplo, en la implementación de una función podría indicarse que ésta “no retorna ningún valor” mediante el uso de `void`:

```
void mostrar_menu()
```

En otros escenarios también puede utilizarse como un tipo de dato genérico. Por ejemplo, cuando se declara un puntero (tema que se tratará en profundidad más adelante), se lo podría definir como puntero a `void` o, lo que es equivalente, un puntero a un dato genérico.

2.6 Otros modificadores.

El lenguaje C brinda, además, los siguientes modificadores:

`const`

Una variable declarada como `const` no puede modificarse durante la ejecución del programa, esto es, permanece constante. Debido a su naturaleza inalterable, se provee un mecanismo para asignar un valor inicial a una variable constante cuando ésta es declarada, como se muestra a continuación:

```
int main()
{
    const char letra = 'A';
    ...
}
```

En algunos casos el valor inicial también podría aportarlo algún otro medio dependiente del hardware.

El uso del modificador `const` permite que el compilador realice verificaciones de consistencia, como detectar que en algún punto del programa se intenta cambiar un valor que, se suponía, debía ser constante. Por lo tanto, es recomendable su utilización.

`volatile`

Indica al compilador que el valor de una variable puede cambiar por medios no especificados en forma explícita en el programa. Podría ser el caso de una variable compartida (y modificable) por dos procesadores o más (en un sistema con múltiples procesadores). Así, si la variable está acompañada del modificador `volatile`, el compilador no aplicará optimizaciones, como mantenerla durante la ejecución del programa en la memoria interna del procesador (registros, memoria caché), ya que en ese caso su valor podría llegar a ser inconsistente respecto del almacenado en memoria.

`extern`

Es un especificador de tipo de almacenamiento que permite que todos los componentes de un programa reconozcan la misma instancia de una variable global. En C se puede escribir

un programa cuyo código esté dispuesto en varios archivos compilables a objeto en forma separada y que, finalmente, se enlazan para generar el programa ejecutable. En ese escenario uno de los archivos de código podría contener la declaración global siguiente:

```
int numero_global;
```

mientras en otro archivo se podría declarar una variable de tipo entero llamada `numero_global` que, en realidad, haga referencia a la misma variable `numero_global` declarada antes. Para ello, se usaría el modificador `extern`:

```
extern int numero_global;
```

Esta declaración permite al compilador un único espacio de almacenamiento físico para este tipo de variables, y es tarea del enlazador de los archivos objeto, resolver las referencias a las variables externas.

`static`

También se trata de un especificador de tipo de almacenamiento que permite que una variable permanezca dentro del archivo donde está declarada, fuera del alcance de otras componentes del programa.

Las variables `static` pueden ser locales o globales; la diferencia es el alcance. En el caso de una variable local, no supera el ámbito de la función donde ha sido declarada; además, conserva su valor entre llamadas a la función.

En el caso de una variable global, el compilador crea una variable conocida sólo en el archivo donde está declarada; o sea que, aunque es global, las demás rutinas incluidas en otros archivos no la reconocerán, por lo que no alterarán su contenido en forma directa y se evitarán efectos secundarios.

En general, este tipo de variables permite ocultar partes de un programa a otras componentes de él.

`register`

Este especificador de tipo de almacenamiento aconseja al compilador que la variable en cuestión se aloje en un registro del procesador (y no en la memoria principal) a lo largo de la ejecución del programa, con el fin de mejorar su desempeño. Dado que se trata de un "consejo", el compilador eventualmente podría ignorarlo.

Este modificador sólo puede aplicarse a variables locales y a parámetros formales de una función, y su uso se recomienda en las variables de control de ciclos. También hay que recordar que una variable de este tipo no tiene dirección que pueda obtenerse mediante el operador de dirección &.

2.7 Tipos de datos definidos por el usuario.

Todos los tipos de datos estudiados hasta ahora son los elementales provistos por el lenguaje, y pueden utilizarse directamente. Sin embargo, un aspecto muy interesante de lenguajes como C y Pascal es su capacidad para que el programador cree estructuras de datos a partir de estos datos simples, que favorecen la legibilidad de los programas y simplifican su mantenimiento.

Los tipos de datos definidos por el usuario, que se desarrollarán a lo largo del libro, se clasifican en:

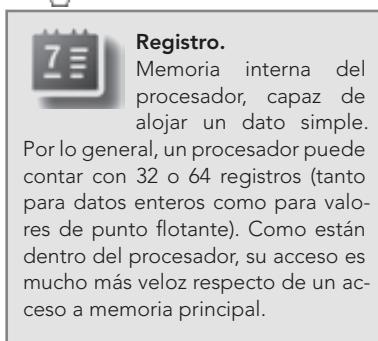
- Escalares definidos por el usuario.
- Estructuras (en C) y registros (en Pascal).
- Arreglos de caracteres.
- Arreglos en general.



Registro.

Memoria interna del procesador, capaz de alojar un dato simple.

Por lo general, un procesador puede contar con 32 o 64 registros (tanto para datos enteros como para valores de punto flotante). Como están dentro del procesador, su acceso es mucho más veloz respecto de un acceso a memoria principal.



- Archivos.
- Punteros.

2.8 Construcción de sentencias básicas.

En un programa de computadora, las sentencias describen las acciones algorítmicas que deben ejecutarse. Expresado de otra manera, un programa es una secuencia de sentencias, que pueden clasificarse en:

- Ejecutables: Especifican operaciones de cálculos aritméticos y entrada/salida de datos.
- No ejecutables: No realizan acciones concretas, pero ayudan a la legibilidad del programa y no afectan su ejecución.

Dentro de las sentencias ejecutables, existen aquellas que permiten llamar a una función y las que se utilizan para asignar un valor a una variable. Estrictamente, la asignación es una operación que sitúa un valor determinado en una posición de la memoria. Esta operación suele representarse en pseudocódigo con el símbolo “ \leftarrow ”, para denotar que el valor situado a su derecha se almacena en la variable situada a la izquierda:

variable \leftarrow expresión

Variable es un identificador válido declarado con anterioridad y expresión es una variable, constante, literal o fórmula para evaluar.

En C, la asignación se realiza con el operador = que se denomina operador de asignación:

```
int numero;
numero = 5;
...
```

En Pascal, en cambio, el operador de asignación es :=, que permite que el operador = pueda usarse para comparación (sin necesidad de cambiar su semántica, como ocurre en C):

```
var numero: integer;
numero := 5;
...
```

Hay algunas asignaciones un tanto diferentes de las convencionales; éstas corresponden al contador y al acumulador. Un contador es una variable entera que se incrementa, cuando se ejecuta, en una unidad o en una cantidad constante:

```
int contador;
contador = 0;
contador = contador + 1;
```

En C también se puede utilizar el operador unario ++:

```
int contador;
contador = 0;
contador++; /* equivalente a contador = contador + 1 */
```

Por su parte, un acumulador es una variable que se incrementa en una cantidad variable. En el fragmento siguiente se muestra un bucle en el que se solicita al usuario el ingreso de números por teclado, que se van acumulando en la variable acum:



Variable es un identificador válido declarado con anterioridad.



Expresión es una variable, constante, literal o fórmula para evaluar.



Un contador es una variable entera que se incrementa, cuando se ejecuta, en una unidad o en una cantidad constante.



Un acumulador es una variable que se incrementa en una cantidad variable.

```

acum = 0;
do
{
    printf("Ingrese un valor: ");
    scanf("%d", &valor);
    acum = acum + valor; /* acumulación */
}
while (valor != 0);

```

En C, la acumulación también puede llevarse a cabo con el operador unario `+=`, como se muestra a continuación:

```

acum = 0;
do
{
    printf("Ingrese un valor: ");
    scanf("%d", &valor);
    acum += valor; /* acumulación */
}
while (valor != 0);

```

2.9 Operadores.

En los lenguajes de programación, un operador se aplica sobre una (operador unario) o dos (operador binario) variables para modificar o utilizar de alguna manera su valor. Un ejemplo de operador unario es el ya visto operador de incrementación para lenguaje C:

```
contador++;
```

El operador de asignación es un buen ejemplo de operador binario:

```
letras = 'A';
```

En los casos típicos los distintos operadores pueden clasificarse en:

- Aritméticos.
- Relacionales y lógicos.
- De manejo de bits.

2.9.1 Operadores aritméticos en C.

Los operadores { `+` , `*` , `-` , `/` } pueden utilizarse con tipos enteros o decimales. Si ambos son enteros, el resultado es entero, pero si alguno de ellos es de punto flotante, el resultado es de punto flotante.

El operador aritmético `%` devuelve el resto de la división entera entre sus dos operandos enteros. Por ejemplo, el resto de dividir 13 por 5 es 3.

El operador `++` incrementa en una unidad el valor de la variable sobre la que se aplica. La contraparte del operador `++` es el operador `--`. Si la variable en cuestión es un puntero (esto es, una variable entera que almacena una dirección de memoria), entonces la dirección almacenada no se incrementa o decrementa en una unidad, sino en tantos bytes como sea el tamaño del tipo de dato apuntado. Los punteros y su aritmética se tratarán con detenimiento en próximos capítulos.

Algunos ejemplos:

```

int i, j;
i = 10;           /* i vale diez */
j = 7;            /* j vale 7 */
j++;              /* ahora j vale 8 */
i--;              /* ahora i vale 9 */

```

En el caso de expresiones donde intervengan varios operadores aritméticos, éstos se evaluarán de acuerdo con sus respectivos niveles de precedencia. Por ejemplo, en la expresión:

```
resultado = a * b + c * d;
```

primero se resuelven los productos $a * b$ y $c * d$, y luego se efectúa su suma, que se asigna en la variable resultado. Esto es así debido a que el operador de multiplicación tiene mayor grado de precedencia respecto del operador de suma. A continuación se listan los operadores citados, en orden de precedencia decreciente:

++ , --
* , / , %
+ , -

Además, para los operadores de igual grado de precedencia, la evaluación se hace de izquierda a derecha. Por ejemplo, ¿qué valor se asigna a la variable resultado en el siguiente ejemplo?:

```

a = 4;
b = 1;
c = 2;
d = 3;
resultado = a * b / c * d;

```

Ya que todos los operadores en cuestión tienen igual grado de precedencia, entonces, de izquierda a derecha, primero se resuelve $a * b$, luego $(a * b) / c$ y, por último, $((a * b) / c) * d$, lo que da como resultado el valor 6. Diferente sería el escenario siguiente:

```

a = 4;
b = 1;
c = 2;
d = 3;
resultado = a * b / (c * d);

```

Los paréntesis que encierran el producto $c * d$ cambian la precedencia de las operaciones y, por lo tanto, su orden de evaluación.

2.9.2 Operadores aritméticos en Pascal.

Pascal soporta los mismos operadores aritméticos que C, con algunas diferencias sintácticas.

En C, por ejemplo, la división entera se lleva a cabo con el operador de división (/) y utilizando operandos enteros. En Pascal, en cambio, debe explicitarse que una división es entera, mediante el operador binario div:

```

num1 := 5;
num2 := 2;
resultado := num1 div num2;

```

El resto de una división entera puede obtenerse con el operador mod:

```
num1 := 5;
num2 := 2;
resultado := num1 mod num2;
```

Los equivalentes a la incrementación (++) y decrementación (--) son, en Pascal, inc () y dec (), respectivamente.

```
inc (contador);
```

2.9.3 Operadores relacionales y lógicos en C.

La lista de operadores relacionales es la siguiente:

Tabla 2-3 - Operadores relacionales en C.

Operador	Acción
>	Mayor que
>=	Mayor o igual que
<	Menor
<=	Menor o igual que
==	Igual
!=	Distinto

La lista de operadores lógicos es la siguiente:

Tabla 2-4 - Operadores lógicos en C.

Operador	Acción
&&	Y (AND)
	O (OR)
!	No (NOT)

En lenguaje C todo valor distinto de 0 significa "verdadero", mientras que el valor 0 significa "falso". Así, una expresión en la que intervienen operadores relacionales o lógicos genera un valor booleano: 1 para indicar "verdadero" y 0 para indicar "falso".

Estos operadores tienen menor grado de precedencia respecto de los aritméticos y, entre ellos, se verifica el siguiente orden de precedencia (de mayor a menor):

!
> , >= , < , <=
== , !=
&&

Por otra parte, el lenguaje C soporta un operador con tres argumentos, el operador ternario condicional, que permite expresar una selección de alternativa doble if-then-else, pero en una sola sentencia, de acuerdo con la sintaxis siguiente:

```
condición ? valor si verdadero : valor falso;
```

De esta forma, si la condición se evalúa como verdadera, se toma valor si verdadero; en caso contrario, se toma valor si falso, como se muestra en el siguiente ejemplo:

```
mensaje = numero>=0 ? "es positivo" : "es negativo";
```

2.9.4 Operadores relacionales y lógicos en Pascal.

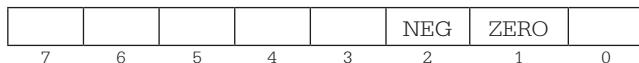
En la tabla siguiente se establecen las equivalencias entre los operadores relacionales y lógicos de Pascal y los del lenguaje C:

Tabla 2-5 - Equivalencias entre operadores relacionales y lógicos en lenguajes C y Pascal.

En Pascal	En C
>	>
\geq	\geq
<	<
\leq	\leq
=	\equiv
\neq	\neq
and	$\&\&$
or	$\ $
not	!

2.9.5 Operadores de manejo de bits en C.

En una computadora, el byte es la unidad más pequeña de manipulación de datos. Sin embargo, en ocasiones es necesario aplicar modificaciones puntuales a algunos de los bits que conforman un byte, lo que resulta en particular útil en el campo de la programación de dispositivos electrónicos empotrados o software de bajo nivel, entre otros. Por ejemplo, un microcontrolador podría disponer de un registro de estado de 8 bits (1 byte), como se muestra en el diagrama siguiente:



En este ejemplo, el bit 2 (NEG) se establece en 1 para indicar que la última operación aritmético-lógica generó un resultado negativo, o en 0 en caso contrario, mientras que el bit 1 (ZERO) se establece en 1 para indicar que la última operación aritmético-lógica generó un resultado nulo, o en 0 en caso contrario. En ese escenario, el programador necesitaría poder “leer” el valor de cada bit para, en función de eso, controlar la ejecución del programa (por ejemplo, el bit NEG podría utilizarse para decidir la alternativa en una estructura `if-then-else`). El mecanismo del que dispone para este fin son los operadores para manejo de bits, que se listan en la tabla siguiente:

Tabla 2-6 - Operadores para manejo de bits.

Operador	Acción
&	AND
	OR
^	XOR
~	Complemento a uno
>>	Desplazamiento a derecha
<<	Desplazamiento a izquierda

En los siguientes ejemplos, se realizarán operaciones a nivel de bits utilizando el byte 10100011, cuyo valor en hexadecimal corresponde a 0xA3. Este byte será operado contra una máscara.

En los siguientes ejemplos, se realizarán operaciones a nivel de bits, utilizando el byte 10100011, cuyo valor en hexadecimal corresponde a 0xA3.

Ejemplo de operación AND:

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = byte & 0xFC;
```

Los números enteros pueden expresarse con diferentes representaciones. El valor decimal 163 puede expresarse como 0xA3 en hexadecimal o como 10100011 en binario.

Byte	1	0	1	0	0	0	1	1	0xA3
Máscara	1	1	1	1	1	1	0	0	0xFC
Resultado	1	0	1	0	0	0	0	0	0xA0

Ejemplo de operación OR:

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = byte | 0x04;
```

Byte	1	0	1	0	0	0	1	1	0xA3
Máscara	0	0	0	0	0	1	0	0	0x04
Resultado	1	0	1	0	0	1	1	1	0xA7

Ejemplo de operación XOR:

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = byte ^ 0xFF;
```

Byte	1	0	1	0	0	0	1	1	0xA3
Máscara	1	1	1	1	1	1	1	1	0xFF
Resultado	0	1	0	1	1	1	0	0	0x5C

Ejemplo de operación complemento a uno (negación):

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = ~byte;
```

Byte	1	0	1	0	0	0	1	1	0xA3
Resultado	0	1	0	1	1	1	0	0	0x5C

Ejemplo de operación de desplazamiento a izquierda una posición (el bit que ingresa por la derecha es 0).

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = byte << 1;
```

Byte	1	0	1	0	0	0	1	1	0xA3
Resultado	0	1	0	0	0	1	1	0	0x46

Ejemplo de operación de desplazamiento a derecha dos posiciones (los bits que ingresan por la izquierda son 0).

```
unsigned char byte = 0xA3;
unsigned char resultado;
resultado = byte >> 2;
```

Byte	1	0	1	0	0	0	1	1	0xA3
Resultado	0	0	1	0	1	0	0	0	0x28

2.9.6 Operadores de manejo de bits en Pascal.

En la tabla siguiente se resumen los operadores de manejo de bits de C tratados en el ítem 2.9.5 y su equivalencia en lenguaje Pascal.

Tabla 2-7 - Equivalencias entre operadores de manejo de bits en lenguaje Pascal y C.	
<i>En Pascal</i>	<i>En C</i>
and	&
or	
xor	^
not	~
shr	>>
shl	<<

2.9.7 Otros operadores.

Hay gran número de operadores en el lenguaje C, además de los presentados antes. Aquí haremos referencia sólo a los más importantes.

El operador [] permite el acceso, mediante un índice, a un elemento en un arreglo:

```
int arreglo_enteros[10];
arreglo_enteros[2] = 0;           /* asigna 0 en la posición 2 */
```

El operador () se utiliza para llamar a una función:

```
mostrar_menu();
```

El operador sizeof retorna el tamaño de la variable o el tipo de dato indicado como argumento, y es de gran utilidad y uso recomendable para asegurar la portabilidad de programas entre computadoras, donde un mismo tipo de dato podría tener tamaños diferentes:

```
tamano_entero = sizeof(int);
```

Por último, se destacan los operadores para manipulación de punteros, que se tratarán más adelante, en el capítulo correspondiente a punteros y manejo de memoria dinámica.

2.10 Operaciones de entrada/salida.

Un programa no es una entidad aislada. Necesita, en mayor o menor medida, interactuar con el “mundo exterior”, tanto para obtener los datos de entrada para procesar, como para enviar los resultados generados. A los mecanismos que permiten establecer esta “interfaz” entre el programa y su entorno se los denomina operaciones de entrada/salida. El ingreso de datos por teclado, la salida de resultados en pantalla o la escritura y la lectura de archivos, son algunos ejemplos de este tipo de operaciones.

En lenguaje C, las operaciones de entrada/salida están implementadas en la unidad de biblioteca stdio (*standard input/output*), que se ajusta al estándar ANSI (el concepto de unidad de biblioteca se tratará con detenimiento más avanzado el libro). Por ahora podemos indicar que es un conjunto de funciones, definiciones de tipos y constantes, que puede incluirse dentro de un programa para su uso, reutilizando el código ya implementado). De las funciones provistas en esta unidad de biblioteca, empezaremos estudiando printf () y scanf (), para salida por pantalla e ingreso de datos por teclado, respectivamente. El programa que haga uso de todas estas funciones deberá incluir el archivo de cabecera (*header*) correspondiente, mediante la directiva al preprocesador #include:

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

2.10.1 Función printf().

Esta función permite escribir una cadena de caracteres en salida estándar (stdout), que en los casos típicos está dirigida a la pantalla de la computadora. Su nombre proviene de “print



En lenguaje C, las operaciones de entrada/salida están implementadas en la unidad de biblioteca stdio (*standard input/output*).



La función printf() permite escribir una cadena de caracteres en salida estándar (stdout)

formatted", que significa que puede controlarse el formato de la escritura. El siguiente ejemplo imprime en pantalla la leyenda "Hola Mundo!":

```
#include <stdio.h>
int main()
{
    printf("Hola Mundo!");
    return 0;
}
```

La función `printf()` presenta algunas características muy interesantes, como un número variable de parámetros. El primero de ellos corresponde a la cadena de formato que, como su nombre lo indica, permite controlar el formato de la visualización de los datos. En el ejemplo "Hola Mundo!" no se aplica un formato en particular, y la leyenda se muestra como es.

El formato puede controlarse mediante el uso de especificadores de formato, que son caracteres "especiales". En el ejemplo siguiente, la cadena "Hola Mundo!" se muestra tabulada (especificador '\t') y con un salto de línea al final (especificador '\n'):

```
#include <stdio.h>
int main()
{
    printf("\tHola Mundo!\n");
    return 0;
}
```

Uno de los usos más frecuentes de los especificadores de formato es la inclusión de valores numéricos en la salida. Por ejemplo, a continuación se muestra en pantalla la leyenda "El resultado de sumar 2 y 3 es 5":

```
#include <stdio.h>
int main()
{
    printf("El resultado de sumar %d y %d es %d \n", 2, 3, 5);
    return 0;
}
```

En la cadena de formato podemos observar el uso del especificador %d, que permite imprimir un número entero en notación decimal con signo. Estos caracteres especiales se reemplazan tomando los valores a partir del segundo parámetro de `printf()`, respetando el orden. A continuación se listan algunos de los modificadores de formato más útiles:

Tabla 2-8 - Modificadores de formato más útiles.

Especificador	Descripción
%d, %i	Impresión de un número entero decimal con signo
%u	Impresión de un número entero decimal sin signo
%x	Impresión de un número entero hexadecimal
%f	Impresión de un número de punto flotante
%lf	Impresión de un número de punto flotante de precisión doble
%c	Impresión de un carácter
%s	Impresión de una cadena de caracteres

2.10.2 Vulnerabilidades de printf().

Esta función es sujeto de una vulnerabilidad en caso de que la cadena de formato que se utiliza sea ingresada por el usuario, como se muestra a continuación:

```
char cadena[100];  
/* En este punto, el usuario ingresa una  
cadena de caracteres por teclado, que  
se almacena en la variable 'cadena' */  
  
...  
printf(cadena);
```

Los modificadores de formato incluidos en la cadena de formato podrían utilizarse con otros fines, que permitieran al usuario tener control sobre la ejecución del programa y, más aún, sobre el sistema completo. Este ataque se conoce como **ataque de la cadena de formato** (*format string attack*) y se evita si no se utiliza jamás como cadena de formato una entrada del usuario, sino que se hace lo que se muestra a continuación:

```
char cadena[100];  
/* En este punto, el usuario ingresa una  
cadena de caracteres por teclado, que  
se almacena en la variable 'cadena' */  
  
...  
printf("%s", cadena);
```

2.10.3 Función scanf().

La función `scanf()` es la contraparte de `printf()` y permite el ingreso de datos a través de entrada estándar (`stdin`), aplicando un formato. La función devuelve por el nombre la cantidad total de datos leídos.

En el ejemplo siguiente se lee un número entero decimal desde el teclado, que se almacena en la variable `numero_entrada` y que luego se imprime en pantalla:

```
#include <stdio.h>  
  
int main()  
{  
    int numero_entrada;  
    scanf("%d", &numero_entrada);  
    printf("El número ingresado es: %i \n", numero_entrada);  
    return 0;  
}
```

El primer parámetro de `scanf()` es la cadena de formato, que le indica a la función cómo debe interpretar el dato leído. El segundo parámetro corresponde a la variable en memoria donde quedará almacenado ese dato, y su tipo debe ser consistente con lo que se espera como entrada. Como puede observarse, la variable `numero_entrada` está precedida por el operador unario `&` (*ampersand*). Esto, que se tratará en el capítulo correspondiente a manejo de punteros,



La función `scanf()` es la contraparte de `printf()` y permite el ingreso de datos a través de entrada estándar (`stdin`), aplicando un formato.

permite que la función `scanf()` reciba la dirección en memoria de `numero_entrada` (y no su contenido), lo cual es lógico, ya que a `scanf()` le interesa conocer dónde debe poner el dato de entrada.

Los especificadores de formato utilizados por la función `scanf()` son, en su mayoría, los mismos que se usan con la función `printf()`.



La función `scanf()` presenta un problema crítico que pone en riesgo la seguridad del programa que la ejecuta y, por esta razón, se desaconseja su uso.

2.10.4 Vulnerabilidades de `scanf()`.

La función `scanf()` presenta un problema crítico que pone en riesgo la seguridad del programa que la ejecuta y, por esta razón, se desaconseja su uso. Analicemos el siguiente caso:

```
#include <stdio.h>

int main()
{
    char cadena[10];

    printf("Ingrese una cadena de caracteres\n");
    scanf("%s", cadena);
    printf("Cadena ingresada: %s \n", cadena);

    return 0;
}
```

En principio, parece un programa “inocente” aunque, en realidad, encierra un error de programación que suele encontrarse con mucha frecuencia. La variable `cadena` tiene lugar para almacenar, como máximo, 10 caracteres. No obstante, ¿qué sucedería si el usuario ingresara por teclado más de 10 caracteres? La información que no quepa en la variable `cadena` estará “pisando” una porción de memoria que no le corresponde y que podría contener otros datos. Peor aún, el usuario podría sobreescribir a conciencia datos vitales para la ejecución del programa, y controlar su ejecución a su gusto. Este escenario ha sido uno de los ataques más populares en la historia de la computación, y se conoce como **ataque por desborde de memoria** (*buffer overflow attack*).

Para evitar esta situación es recomendable el uso de una función de entrada más segura, como es el caso de `fgets()`. A diferencia de `scanf()`, esta función no permite definir el formato de entrada, con lo que se requiere un paso adicional para convertir la cadena leída en, por ejemplo, un número. Sin embargo, la característica más sobresaliente es que recibe en uno de sus parámetros la cantidad máxima de caracteres que se leerán. Veamos un ejemplo:

```
#include <stdio.h>

#define MAX_SIZE 10

int main()
{
    char cadena[MAX_SIZE];

    printf("Ingrese una cadena de caracteres\n");
    fgets(cadena, MAX_SIZE, stdin);
    printf("Cadena ingresada: %s \n", cadena);

    return 0;
}
```

En este caso no importa cuántos caracteres ingresó el usuario; sólo se consideran tantos como se indiquen en el segundo parámetro de `fgets()`. Además, el tercer parámetro establece el origen de los datos de entrada (en el ejemplo se utiliza la entrada estándar: el teclado).

2.10.5 Entrada y salida en Pascal.

A diferencia de C, el soporte a entrada/salida en Pascal está incorporado en el propio lenguaje. Las funcionalidades más relevantes son los procedimientos `write()`/`writeln()` y `read()`/`readln()`, para escritura y lectura.

`write()`/`writeln()` permiten escribir una cadena de caracteres por salida estándar (la segunda inserta, además, un salto de línea al final de la escritura). A diferencia de `printf()`, no se utilizan modificadores para insertar los datos que se han de imprimir, sino que éstos se intercalan en la posición correspondiente, como se muestra a continuación:

```
program entrada_salida;
var num1, num2: integer;
begin
    num1 := 5;
    num2 := 2;
    writeln('La suma de ', num1, ' y ', num2, ' es ', num1+num2);
end.
```

Los procedimientos `read()`/`readln()` son equivalentes a la función `scanf()` de C, y se utilizan para leer datos por entrada estándar. El mismo ejemplo anterior, pero cargando los operandos por teclado, luce de la siguiente manera:

```
program entrada_salida;
var num1, num2: integer;
begin
    write('Ingrese un número: ');
    readln(num1);
    write('Ingrese otro número: ');
    readln(num2);
    writeln('La suma de ', num1, ' y ', num2, ' es ', num1+num2);
end.
```

2.11 Resumen.

Durante su ejecución, un programa procesa datos que residen en la memoria de la computadora, como números, letras o cadenas de caracteres. Todos ellos se ajustan a un “molde”, y nos referimos a él como el tipo del dato. Lenguajes como C o Pascal son fuertemente tipados, en cuanto a que es obligatorio que el programador asigne un tipo determinado a cada dato.

Los lenguajes de programación ofrecen al programador un conjunto de tipos de datos predefinidos (por ejemplo, tipos simples, como números enteros, reales, caracteres o booleanos) a partir de los cuales pueden construirse otros, definidos por el usuario (algunos más complejos, como los tipos estructurados –arreglos, registros o tablas–).

El procesamiento de los datos dentro de un programa se lleva a cabo mediante la aplicación de operadores. Éstos se aplican sobre una (operador unario) o dos (operador binario) variables para modificar o utilizar de alguna manera el valor de éstas. El operador de incremento (`++`) en lenguaje C es un ejemplo de operador unario, mientras que los operadores binarios más elementales son los aritméticos: Suma, resta, división y producto.

Además de los operadores aritméticos, el programador también puede hacer uso de operadores relacionales (mayor que, mayor o igual que, menor que, menor o igual que, igual, distinto) y lógicos (y, o, no). Otros operadores de utilidad significativa son los que permiten la manipulación de bits dentro de un byte. En ocasiones es necesario aplicar modificaciones puntuales a algunos de los bits que conforman un byte, como podría ser una operación lógica (AND, OR, XOR) contra otro bit o desplazarlo hacia izquierda o derecha un número determinado de posiciones.

Por último, en este capítulo también se realizó una introducción a las operaciones de entrada/salida básicas, como la visualización de datos por pantalla o la entrada por teclado, y se alertó sobre las vulnerabilidades propias de estas operaciones. Junto con el capítulo I, el capítulo II permite que el lector escriba sus propios programas básicos, utilizando datos y operaciones simples.

2.12 Problemas propuestos.

- 1) Desarrolle un algoritmo que permita leer dos valores distintos, y determinar cuál de los dos es el mayor.
- 2) Realice un algoritmo que permita leer dos valores y determinar cuál de los dos es el menor.
- 3) Realice un algoritmo que sume dos números.
- 4) Desarrolle un algoritmo que permita leer tres valores y almacenarlos en las variables A, B y C, respectivamente. El algoritmo debe imprimir cuál es el mayor y cuál el menor. Recuerde constatar que los tres valores introducidos por el teclado sean distintos. Presente un mensaje de alerta en caso de que se detecte la introducción de valores iguales.
- 5) Desarrolle un algoritmo que realice la sumatoria de los números enteros comprendidos entre el 1 y el 10, esto es, $1 + 2 + 3 + \dots + 10$.
- 6) Desarrolle un algoritmo que realice la sumatoria de los números enteros múltiplos de 5, comprendidos entre el 1 y el 100, o sea, $5 + 10 + 15 + \dots + 100$. El programa deberá imprimir los números en cuestión y finalmente su sumatoria.
- 7) Desarrolle un algoritmo que realice la sumatoria de los números enteros pares comprendidos entre el 1 y el 100, esto es, $2 + 4 + 6 + \dots + 100$. El programa deberá imprimir los números en cuestión y finalmente su sumatoria.
- 8) Desarrolle un algoritmo que lea los primeros 300 números enteros y determine cuántos de ellos son impares; al final deberá indicar su sumatoria.
- 9) Determine la hipotenusa de un triángulo rectángulo del que se conocen las longitudes de sus dos catetos. Desarrolle el algoritmo correspondiente.
- 10) Desarrolle un algoritmo que permita determinar el área y el volumen de un cilindro, dados su radio (R) y su altura (H).
- 11) Desarrolle un algoritmo que permita leer un valor cualquiera N y escriba si ese número es par o impar.
- 12) Desarrolle un algoritmo que le permita determinar de una lista de números:
 - 12.1. ¿Cuántos están entre el 50 y el 75, ambos inclusive?
 - 12.2. ¿Cuántos son mayores que 80?
 - 12.3. ¿Cuántos son menores que 30?
 El algoritmo debe finalizar cuando n (el total de números de la lista) sea igual a 0.
- 13) Desarrolle un algoritmo que permita convertir calificaciones numéricas según la siguiente consigna:
 $A = 19 \text{ y } 20; B = 16, 17 \text{ y } 18; C = 13, 14 \text{ y } 15; D = 10, 11 \text{ y } 12;$
 $E = 1 \text{ hasta } 9.$ Se asume que la nota está comprendida entre 1 y 20. Realice esto hasta que se ingrese nota igual a cero.
- 14) Realice un algoritmo que determine el pago para realizar por la entrada a un espectáculo donde se pueden comprar sólo hasta cuatro localidades, donde al costo de dos entradas se les descuenta el 10%, al de tres, el 15%, y a la compra de cuatro se le descuenta el 20%.
- 15) Dada la duración en minutos de una llamada calcular el costo, considerando:
 - Hasta tres minutos el costo es \$ 0,50.
 - Por encima de tres minutos es \$ 0,50 más \$ 0,1 por cada minuto adicional a los tres primeros.

2.13 Problemas resueltos.

1- Desarrolle un algoritmo para sumar dos números.

Solución en lenguaje C

```
#include <stdio.h>
int main (void)
{
    float a , b , s;
    printf("Algoritmos para sumar dos números\n");
    printf("-----\n\n");
    printf("Introduzca dos números: ");
    scanf("%f %f", &a , &b);

    s = a + b;
    printf ("\nEl resultado es %6.2f \n" , s);
    return 0;
}
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

Solución en lenguaje Pascal

```
PROGRAM alg_01_cap02;
VAR a, b, s: REAL;
BEGIN
    WRITELN('Algoritmo para sumar dos números') ;
    WRITELN('-----') ;
    WRITELN;
    WRITE('Introduzca dos números : ') ;
    READLN(a , b);
    s := a + b;
    WRITELN;
    WRITELN('El resultado es ', s:6:2);
END.
```

2- Diseñe un algoritmo que, dado un número real que entra como dato, nos indique si está contenido dentro de límites predeterminados.

Solución en lenguaje C

```
#include <stdio.h>
#define LINF 100
#define LSUP 200
int main (void)
```

```

{
    float dato;

    printf("Algoritmo de comprobación de límites\n");
    printf("-----\n\n");

    printf("Introduzca el número : ");
    scanf("%f", &dato);

    if ( dato >= LINF )
    {
        if ( dato <= LSUP )
        {
            printf ("Está dentro del intervalo [%i,%i]\n" , LINF, LSUP);
        }
        else
        {
            printf("Supera el máximo\n");
        }
    }
    else
    {
        printf("No alcanza el mínimo\n");
    }

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_02_cap02;

CONST LINF = 100;
      LSUP = 200;

VAR dato: REAL;

BEGIN
  WRITELN('Algoritmo de comprobación de límites');
  WRITELN('-----');
  WRITELN ;

  WRITE('Introduzca el número : ');
  READLN(dato);
  WRITELN ;

  IF (dato >= LINF) THEN
  BEGIN
    IF (dato <= LSUP) THEN
    BEGIN
      WRITELN('Está dentro del intervalo [ ' , LINF , ',' , LSUP , ' ]')
    END
  ELSE

```

```
BEGIN
    WRITELN('Supera el máximo');
    END
ELSE
BEGIN
    WRITELN('No alcanza el mínimo');
END;
END.
```

3- Plantee un algoritmo que convierte kilómetros en millas.

Solución en lenguaje C

```
#include <stdio.h>

int main (void)
{
    int tope, kilometro;
    float factor, milla;

    printf("Conversión de kilómetros a millas\n");
    printf("-----\n\n");

    factor = 1.6093;

    do
    {
        printf("Introduzca el máximo de kilómetros a convertir ");
        scanf("%i", &tope);
    }
    while ( tope < 1 );

    printf("\n Kilómetro           Millas\n");
    printf("-----\n\n");

    kilometro = 1;

    while (kilometro <= tope)
    {
        milla = kilometro / factor;
        printf("%6i      %4.1f\n", kilometro, milla);
        kilometro++;
    }
    printf ("-----\n");

    return 0;
}
```

Solución en lenguaje Pascal

```

PROGRAM alg_03_cap02;
VAR tope, kilometro : INTEGER;
    factor, milla : REAL;
BEGIN
    WRITELN('Conversión de kilómetros a millas');
    WRITELN('-----');
    WRITELN;
    factor := 1.6093;
    REPEAT
        WRITE('Introduzca el máximo de kilómetros a convertir : ');
        READLN(tope);
    UNTIL (tope >= 1);

    WRITELN;
    WRITELN('Kilómetro Milla ');
    WRITELN('-----');

    kilometro := 1;
    WHILE Kilometro <= tope DO
        BEGIN
            milla := kilometro / factor;
            WRITELN (kilometro:6, ' ', milla:4:1);
            kilometro := kilometro + 1;
        END;
    WRITELN('-----');
END.

```

4- Programe un algoritmo que calcule la suma de los n primeros números enteros. Utilice la estructura iterativa while.

Solución en lenguaje C

```

#include <stdio.h>

int main (void)
{
    int n, numero, suma;

    printf("Suma de los n primeros números enteros\n");
    printf("-----\n\n");

    printf("¿ Hasta qué entero quiere sumar ? ");
    scanf("%i", &n);

    numero = 1;
    suma = 0;

```

```

while ( numero <= n )
{
    suma += numero; /* Equivale a suma = suma + numero; */
    numero++;        /* Equivale a numero = numero + 1; */
}

printf("\nLa suma es %i\n", suma);
return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_04_cap02;

VAR n, numero, suma : INTEGER;

BEGIN
  WRITELN('Suma de los n primeros números enteros');
  WRITELN('-----');
  WRITELN ;
  WRITE('¿ Hasta qué entero quiere sumar ? ');
  READLN(n);

  numero := 1;
  suma := 0;

  WHILE numero <= n DO
  BEGIN
    suma := suma + numero;
    numero := numero + 1;
  END;
  WRITELN('La suma es ', suma);
END.

```

5- Diseñe un algoritmo que calcule la suma de los n primeros números enteros. Utilice la estructura iterativa `for`.

Solución en lenguaje C

```

#include <stdio.h>

int main (void)
{
    int n, numero, suma;

    printf("Suma de los n primeros números enteros\n");
    printf("-----\n\n");

    printf("¿ Hasta qué entero quiere sumar? ");
    scanf("%i", &n);

    suma = 0;

```

```

for (numero = 1; numero <= n; numero++)
{
    suma += numero;
}

printf ("\nLa suma es %i\n", suma);

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_05_cap02;

VAR n, numero, suma : INTEGER;

BEGIN
    WRITELN('Suma de los n primeros números enteros');
    WRITELN('-----');
    WRITELN;

    WRITELN('¿ Hasta qué entero quiere sumar? ');
    READLN(n);

    suma := 0;

    FOR numero := 1 TO n DO
        BEGIN
            SUMA := SUMA + NUMERO;
        END;

    WRITE('La suma es ', suma);

END.

```

6- Desarrolle un algoritmo para calcular el área de un círculo.

Solución en lenguaje C

```

#include <stdio.h>

#define PI 3.1415

int main (void)
{
    float r, area;

    printf("Algoritmos para calcular el área de un círculo\n");
    printf("-----\n\n");

    printf("Introduzca el radio: ");
    scanf("%f", &r);

    area = PI * r * r;

    printf("\nEl resultado es %.2f\n" , area);
}

```

```
    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM alg_06_cap02;
VAR r, area: REAL;
BEGIN
  WRITELN('Algoritmo calcular el área de un círculo');
  WRITELN('-----');
  WRITELN;
  WRITE('Introduzca el radio: ');
  READLN(r);
  area := PI * r * r; {La constante PI está definida en el lenguaje}
  WRITELN;
  WRITELN('El resultado es ', area:6:2);
END.
```

7- Programe un algoritmo que, dados dos números enteros que entran como datos, indique si uno es divisor del otro.

Solución en lenguaje C

```
#include <stdio.h>

int main (void)
{
    int dato1, dato2;

    printf("Divisibilidad entre dos números enteros\n");
    printf("-----\n\n");

    printf("Introduzca dos números enteros : ");
    scanf("%i %i", &dato1 , &dato2);

    if ( dato1 > dato2 )
    {
        /* dato2 puede ser divisor */
        if ( dato1%dato2 == 0 )
        {
            printf("%i es divisor de %i\n" , dato2, dato1);
        }
        else
        {
            printf("%i no es divisor de %i\n" , dato2, dato1);
        }
    }
    else
    {
```

```

        printf("%i no es divisor de %i porque es mayor\n" , dato2, dato1);
    }

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_07_cap02;

VAR dato1, dato2: INTEGER ;

BEGIN
    WRITELN('Divisibilidad entre números enteros');
    WRITELN('-----');
    WRITELN ;
    WRITE('Introduzca dos números enteros : ');
    READLN(dato1 , dato2);
    WRITELN ;
    IF ( dato1 > dato2 ) THEN
        BEGIN
            { dato2 puede ser divisor }
            IF ( dato1 mod dato2 = 0 ) THEN
                BEGIN
                    WRITELN(dato2 , ' es divisor de ' , dato1);
                END
            ELSE
                BEGIN
                    WRITELN(dato2 , ' no es divisor de ' , dato1);
                END
        END
    ELSE
        BEGIN
            WRITELN(dato2 , ' no es divisor de ' , dato1, ' porque es mayor');
        END;
    END.

```

- 8- Escriba un algoritmo que calcule el producto de los n primeros números naturales. Compruebe que el valor entrado de n sea válido ($n \geq 0$). Utilice la estructura iterativa while.

Solución en lenguaje C

```

#include <stdio.h>

int main (void)
{
    int n, aux;
    unsigned long factorial;

```

```
printf("Factorial\n");
printf("-----\n\n");
do
{
    printf("Introduzca un número entero : ");
    scanf("%i", &n);
}
while ( n < 0 );

factorial = 1;
aux = 1;

while ( aux <= n )
{
    factorial *= aux; /* Equivale a factorial = factorial * aux; */
    aux++;             /* Equivale aux = aux + 1; */
};

printf("\nEl factorial de %i es %i\n", n, factorial);

return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM alg_08_cap02;
VAR n, aux, factorial: LONGWORD;
BEGIN
    WRITELN('Factorial');
    WRITELN('-----');
    WRITELN;
    REPEAT
        WRITE('Introduzca un número entero: ');
        READLN(n);
    UNTIL n >= 0;
    factorial := 1;
    aux := 1;
    WHILE ( aux <= n ) DO
        BEGIN
            factorial := factorial * aux;
            INC(aux); { Equivale a aux := aux + 1; }
        END;
    WRITELN('El factorial de ', n, ' es ', factorial);
END.
```

9- Programe un algoritmo que calcule e imprima los n primeros términos de la serie de Fibonacci definida como:

```
Fibo (0) = 0
Fibo (1) = 1
Fibo (i) = Fibo(i-1) + Fibo(i-2)    para i>1
```

Solución en lenguaje C

```
#include <stdio.h>

int main (void)
{
    int n;                      /* Número de términos a calcular */
    unsigned long fibo;          /* Representa el término Fibo (i) */
    unsigned long actual;        /* Representa el término Fibo (i-1) */
    unsigned long anterior;      /* Representa el término Fibo (i-2) */
    int contador;                /* Representa el término Fibo 'i' */

    printf("Serie de Fibonacci\n");
    printf("-----\n\n");

    do
    {
        printf("¿Cuántos términos de la serie quiere calcular? (2 o más) ");
        scanf("%i", &n);
    }
    while (n < 2);

    /*Fibo(0) = 0 y Fibo(1) = 1 */
    anterior = 0;
    actual = 1;

    /*Escribe los dos primeros términos*/
    printf("%i\n", anterior);
    printf("%i\n", actual);

    contador = 2;

    while (contador < n)
    {
        /* Calcula el nuevo término Fibo(i) = Fibo(i-1) + Fibo(i-2) */
        fibo = anterior + actual;
        printf("%i\n", fibo);

        /*Actualiza Fibo(i-2), Fibo(i-1) y 'i' */
        anterior = actual;
        actual = fibo;
        contador++;
    }

    return 0;
}
```

Solución en lenguaje Pascal

```

PROGRAM alg_09_cap02;

VAR n:           INTEGER;
    fibo:        LONGWORD;
    actual:      LONGWORD;
    anterior:    LONGWORD;
    contador:    INTEGER;

BEGIN
    WRITELN('Serie de Fibonacci');
    WRITELN('-----');
    WRITELN;

    REPEAT
        WRITE('¿ Cuántos términos de la serie quiere calcular ? (2 o más) ');
        READLN(n);
    UNTIL (n >= 2);

    { Fibo(0) = 0 y Fibo(1) = 1 }
    anterior := 0;
    actual := 1;

    { Escribe los dos primeros términos }
    WRITELN(anterior);
    WRITELN(actual);

    contador := 2;

    WHILE (contador < n) DO
        BEGIN
            { Calcula el nuevo término Fibo(i) = Fibo(i-1) + Fibo(i-2) }
            fibo:= anterior + actual;
            WRITELN(fibo);

            { Actualiza Fibo (i-2) ; Fibo(i-1) y 'i' }
            anterior := actual;
            actual := fibo;
            contador := contador + 1;
        END;
    END.

```

10- Escriba un programa que pruebe diferentes formatos de impresión.

Solución en lenguaje C

```

#include <stdio.h>
int main (void)
{
    /*Definición de variables. En C no existe el tipo BOOLEAN */

```

```

int      dato1;
float    dato2;
char     dato3, dato4;

printf("Pruebas de formatos de impresión\n") ;
printf("-----\n\n") ;

/* Inicializamos las variables */
dato1 = 205;
dato2 = 205.5;
dato3 = 'a';
dato4 = 'b';

/* Pruebas */
printf("Entero 205 sin formato 2 veces : %i %i\n" , dato1 , dato1);
printf("Entero 205 con formato (:6) : %6i\n" , dato1);
printf("Entero 205.5 sin formato : %f\n" , dato2);
printf("Entero 205.5 con formato (exp) : %e\n" , dato2);
printf("Entero 205.5 con formato (12) : %12f\n" , dato2);
printf("Entero 205.5 con formato (12.0) : %12.0f\n" , dato2);
printf("Entero 205.5 con formato (12.2) : %12.2f\n\n" , dato2);

printf("Linea completa con entero 205 (6), real 205.5(8.2) y 'mesa' (8)\n");
printf ("%6i %8.2f %8s\n\n" , dato1 , dato2 , "mesa");

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_10_cap02;

VAR dato1:          INTEGER ;
    dato2:          REAL ;
    dato3, dato4: CHAR ;
    dato5, dato6: BOOLEAN ;

BEGIN
    WRITELN('Pruebas de formatos de impresión');
    WRITELN('-----');
    WRITELN;

    { Inicialización de variables }
    dato1 := 205;
    dato2 := 205.5;
    dato3 := 'a';
    dato4 := 'b';
    dato5 := TRUE ;
    dato6 := FALSE ;

    { Pruebas }
    WRITELN('Entero 205 sin formato 2 veces : ' , dato1 , dato1);

```

```
WRITELN('Entero 205 con formato (:6) : ' , dato1:6);
WRITELN('Real 205.5 sin formato      : ' , dato2);
WRITELN('Real 205 con formato (:6)      : ' , dato2:6);
WRITELN('Real 205 con formato (:10)     : ' , dato2:10);
WRITELN('Real 205 con formato (:10:0)   : ' , dato2:10:0);
WRITELN('Real 205 con formato (:10:2)   : ' , dato2:10:2);
WRITELN;
WRITELN('CHAR "a" y "b" sin formato    : ' , dato3 , dato4);
WRITELN('CHAR "a" y "b" con formato (:6) : ' , dato3:6 , dato4:6);
WRITELN('Literal "mesa" y "silla" sin formato    : ' , 'mesa' ,
'silla');
WRITELN('Literal "mesa" y "silla" con formato(:6) : ' , 'mesa':6 ,
'silla':6);
WRITELN;
WRITELN('Línea completa con entero 205(:6), real 205.5(:8:2) y "mesa"(:8)');
WRITELN(dato1:6 , dato2:8:2 , 'mesa':8) ;
WRITELN;
WRITELN('Boolean verdad y falso sin formato    : ' , dato5      , dato6);
WRITELN('boolean verdad y falso con formato (:8) : ' , dato5:8      , dato6:8);

END.
```



2.14 Contenido de la página Web de apoyo.

El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Autoevaluación.



Video explicativo (02:19 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas.*



Presentaciones. *

3

Subrutinas

Contenido

3.1 Introducción	58
3.2 Funciones	59
3.3 Ámbito de las declaraciones.....	62
3.4 Parámetros	65
3.5 Argumentos por línea de comandos	67
3.6 Mapa de memoria.....	69
3.7 Consideraciones de desempeño	72
3.8 Resumen.....	74
3.9 Problemas propuestos.....	75
3.10 Problemas resueltos.....	76
3.11 Contenido de la página Web de apoyo.....	88

Objetivos

- Dividir un problema en otros más sencillos o pequeños, con el fin de abordarlos en forma individual y de manera que, en su conjunto, resuelvan el problema original (Técnica de "divide y vencerás").
- Dominar la metodología *top-down*.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

3.1 Introducción.

Según la complejidad del problema que se ha de resolver, un programa podría extenderse en miles o millones de líneas de código. Es posible que el lector haya escuchado esta afirmación en reiteradas oportunidades, por parte de un amigo o un profesor de la universidad, aunque a menudo no se tiene una apreciación palpable de lo que esto significa. Para expresarlo en términos concretos, podríamos mencionar casos más familiares, e indicar que la versión 2.6 del núcleo de Linux tiene más de 5 millones de líneas de código, que Windows XP está programado en unas 40 millones de líneas, o que el navegador Firefox 1.5, por su parte, contiene más de 2 millones. Si nos tomamos unos minutos para reflexionar sobre lo que estos números significan, no será difícil imaginar lo caótico que resulta trabajar en desarrollos de semejantes tamaños, por el esfuerzo humano y la proliferación de errores (*bugs*) de programación que se generaría. ¿Qué solución tenemos para este dilema?

En el año 480 aC, en las proximidades de la Isla de Salamina (Grecia), 1200 barcos de guerra de origen persa, con 150 000 tripulantes a bordo, se enfrentaron con las fuerzas navales griegas, inferiores en poderío y número (poco menos de 400 naves). A pesar de la diferencia en las fuerzas, los barcos griegos eran más pequeños y, por esa razón, capaces de moverse con mayor agilidad. Los persas no pudieron resistir en este escenario y, tras ocho horas de combate, iniciaron la retirada. La clave del triunfo griego se debió al uso de embarcaciones más pequeñas pero más manejables con respecto a las persas, que eran de mayor tamaño. Éste es uno de numerosos ejemplos de una estrategia aplicable en el ámbito de la guerra, la política, la economía y las ciencias, y que se conoce como “dividir y conquistar” o “dividir y reinar”.

En términos más formales, la estrategia de “dividir y conquistar” significa particionar un problema en otros más pequeños y más fáciles de resolver en forma individual (más “manejables”) que, en conjunto, solucionen el problema original. Aplicado a la programación, permite descomponer un software en módulos más pequeños, de manera que un programa con, por ejemplo, 2 millones de líneas de código pueda elaborarse a partir de un conjunto de subprogramas, cada uno más pequeño y más fácil de programar y mantener. Por ejemplo, el sistema de control de una máquina expendedora de gaseosas podría descomponerse en partes más simples, aplicando la estrategia de “dividir y conquistar”, como se muestra a continuación:

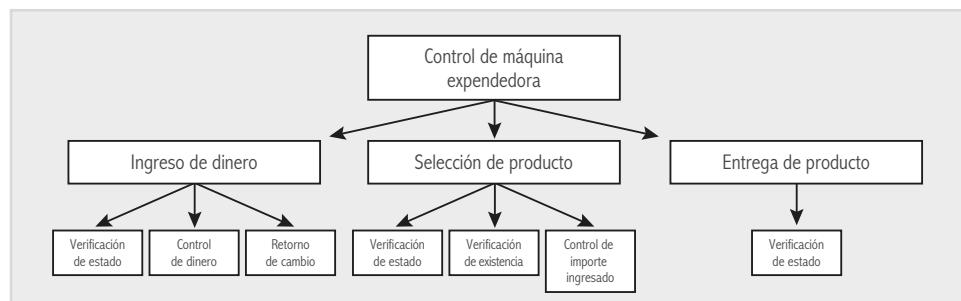


Fig. 3-1. Sistema de control de una máquina expendedora de gaseosas.

Si bien se trata de un ejemplo simplificado, permite observar una de las consecuencias más esperables de la modularización correcta: reutilización de código. Por ejemplo, independientemente de la acción que efectúa la máquina expendedora, siempre se ejecuta una verificación de su estado (para no aceptar más compras si hubiese una moneda trabada o una gaseosa que obstruye la salida de productos). Este módulo, que en el gráfico se denominó verificación de estado, se construiría una vez y se reutilizaría en diferentes partes del sistema en cuestión.

Sin embargo, definir cada uno de los módulos (o subprogramas) no siempre es una tarea simple. Cada uno de ellos debe implementar una funcionalidad concreta y bien definida (por

ejemplo, retorno de cambio); esto es, sus límites deben estar claros. Así, la calidad de la modularización puede expresarse según sus grados de cohesión y acoplamiento:

- **Cohesión:** Se refiere a la afinidad o familiaridad de los elementos internos de un módulo. Por ejemplo, si el módulo control de dinero además imprime un mensaje con el importe total ingresado, esta funcionalidad no es coherente con lo estrictamente relacionado con el control del dinero. Se busca la mayor cohesión posible.
- **Acoplamiento:** Se refiere al grado de dependencia entre módulos. Si un módulo puede ejecutar su tarea por sí solo, con independencia de los demás, entonces el acoplamiento es bajo. Se busca el menor acoplamiento posible.

Por último, es necesario indicar que cada módulo no es una entidad aislada. Por el contrario, recibirá datos desde el exterior (entradas) que, una vez procesados, generan un resultado que el módulo retorna a su entorno (salida). Esto se verá con más claridad en las secciones siguientes, construyendo funciones que reciban parámetros y retornen un resultado.

3.2 Funciones.

En lenguaje C, la modularización se lleva a cabo mediante el uso de funciones, que luego cooperarán entre sí para realizar las tareas necesarias de manera de obtener el resultado deseado por el usuario final. La sintaxis de una función en C es la siguiente:

```
<tipo de retorno> <nombre de la función> (<lista de parámetros>
{
    <implementación de la función>
}
```

Por ejemplo, una función que recibe dos números enteros como parámetros y retorna la suma de ambos luciría de la siguiente manera:

```
int suma(int a, int b)
{
    int resultado;
    resultado = a + b;
    return resultado;
}
```

En este ejemplo la función se llama suma, recibe dos parámetros de tipo entero (a y b), y retorna un entero (que es la suma de los parámetros). Una función siempre retorna un solo valor (en casos eventuales podría no devolver nada si se usa void como tipo de retorno). Si fuese necesario devolver más de un valor, debería realizarse por medio de los parámetros, que dejarían de ser sólo de entrada, para convertirse en parámetros de entrada y salida. Esto se implementa pasando referencias a los argumentos (no los argumentos en sí), como se explicará más adelante en este capítulo.

En la definición de una función, a la parte encerrada entre llaves { y } se la denomina cuerpo, mientras que el encabezado integrado por el nombre de la función, el tipo de retorno y su lista de parámetros, se conoce como interfaz o prototipo, y es el punto por el que la función interactúa con el mundo exterior (recibiendo los datos de entrada y retornando un resultado como salida). Es usual que las funciones se definan a continuación de la función main(), en cuyo caso es necesario, para cada una de ellas, escribir el prototipo antes del main(), como se muestra en el ejemplo siguiente:



En lenguaje C, la modularización se lleva a cabo mediante el uso de funciones, que luego cooperarán entre sí para realizar las tareas necesarias de manera de obtener el resultado deseado por el usuario final.

```

#include <stdio.h>

/* Prototipo de la función 'suma()' */
int suma(int a, int b);

int main()
{
    int operando1, operando2, resultado;

    operando1 = 5;
    operando2 = 10;
    resultado = suma(operando1, operando2);

    printf("%d + %d = %d \n", operando1, operando2, resultado);

    return 0;
}

int suma(int a, int b)
{
    int resultado;
    resultado = a + b;
    return resultado;
}

```



Una función puede ser llamada desde cualquier parte del programa que la contiene, posterior a su declaración/definición.



Encuentre un simulador sobre función y procedimiento en la Web de apoyo.

Una función puede ser llamada desde cualquier parte del programa que la contiene, posterior a su declaración/definición. Se la invoca con su nombre, seguido de una lista opcional de argumentos (o parámetros). Los argumentos van entre paréntesis y, si hubiera más de uno, separados por comas. Cuando la función no devuelve valor alguno, se reemplaza el tipo de retorno por la palabra reservada void:

```

void mostrar_mensaje()
{
    printf("Esta función no retorna nada \n");
}

```

Por otra parte, en lenguaje Pascal también existe el concepto de función como el que se acaba de exponer. No obstante, además se provee soporte para otro mecanismo de modularización: El procedimiento.

Una función en Pascal se ajusta a la sintaxis siguiente:

```

function <nombre de la función> (<parámetros>): <tipo de retorno>;
begin
    <implementación de la función>
end;

```

Por ejemplo:

```
function suma(a, b: integer): integer;  
  
var resultado: integer;  
begin  
    resultado := a + b;  
    suma := resultado;  
end;
```

Nótese que, a diferencia de C, en el encabezado se explicita que se está declarando una función, mediante el uso de la palabra reservada `function`.

También es importante aclarar que para retornar un valor, el mismo debe asignarse al nombre de la función, como si esta se tratase de una variable (lo cual se puede apreciar en el ejemplo anterior, en la línea: `suma := resultado;`).

En Pascal un procedimiento es lo que en C definimos como una función que no retorna resultado (`void`). Su sintaxis es similar a la de una función, salvo que no se establece un tipo de retorno:

```
procedure <nombre del procedimiento> (<parámetros>);  
  
begin  
    <implementación del procedimiento >  
end;
```

Por ejemplo:

```
procedure mostrar_mensaje();  
  
begin  
    writeln('Bienvenido!');  
end;
```

A continuación se expone el ejemplo completo, en el que se hace uso de la función `suma()` y del procedimiento `mostrar_mensaje()`, donde puede observarse cómo se llevan a cabo las llamadas a ellos:

```
program ejemplo;  
  
function suma(a, b: integer): integer;  
var resultado: integer;  
begin  
    resultado := a + b;  
    suma := resultado;  
end;  
  
procedure mostrar_mensaje();  
begin  
    writeln('Bienvenido!');  
end;  
  
var operando1, operando2, resultado: integer;  
  
begin
```



En Pascal, un procedimiento es lo que en C definimos como una función que no retorna resultado (`void`).

```

{ Llamada al procedimiento 'mostrar_mensaje()' }
mostrar_mensaje; {En Pascal es opcional el uso de paréntesis '()' cuando la subrutina no
recibe parámetros, en C no}
operando1 := 5;
operando2 := 10;

{ Llamada a la función 'mostrar_mensaje()' }
resultado := suma(operando1, operando2);

writeln(operando1, ' + ', operando2, ' = ', resultado);

end.

```



Declaraciones locales: Pueden emplearse en cualquier punto dentro de la función; su existencia y alcance están limitados por la función que las contiene.

Declaraciones globales: se realizan fuera de cualquier función y su alcance es el de todas las funciones del programa.

3.3 Ámbito de las declaraciones.

El ámbito de un identificador se refiere a la “región” dentro de la cual está declarado y puede utilizarse. Este concepto se aplica a todos los tipos de declaraciones, no sólo a constantes y variables.

En el caso particular de constantes y variables definidas dentro del cuerpo de una función (incluyendo, para el caso de C, las de la función main()), pueden emplearse en cualquier punto dentro de este bloque, exclusivamente. A estas declaraciones se las denomina locales, ya que su existencia y alcance están limitados por la función que las contiene. En cambio, cuando las declaraciones se realizan fuera de cualquier función, se las considera globales, y su alcance es el de todas las funciones del programa. En el ejemplo siguiente se muestra el uso de declaraciones locales:

```

#include <stdio.h>

/* Prototipos de funciones */
int calcular_maximo(int, int, int);
void imprimir_numeros(int, int, int);

int main()
{
    /* Variables locales de la función 'main()' */
    int n1, n2, n3, maximo;

    printf("Ingrese tres números enteros: ");
    scanf("%d %d %d", &n1, &n2, &n3);

    imprimir_numeros(n1, n2, n3);
    maximo = calcular_maximo(n1,n2,n3);
    printf("El máximo es: %d\n", maximo);

    return 0;
}

int calcular_maximo(int a, int b, int c)
{
    /* Variable local de la función 'calcular_maximo()' */
    int mayor;

    mayor = a;

    if (b > mayor)
        mayor = b;
    if (c > mayor)

```

```

        mayor = c;
        return mayor;
    }

void imprimir_numeros(int a, int b, int c)
{
    /* Función sin variables locales */
    printf("Usted ingresó los siguientes números: %d %d %d \n",
    a, b, c);
}

```

En este ejemplo, las variables `n1`, `n2`, `n3` y `maximo` son locales a `main()`, y sólo pueden utilizarse dentro del cuerpo de esa función. De igual forma, `mayor` sólo puede accederse dentro de la función `calcular_maximo()`, y su existencia está limitada por la existencia de esta función. En cambio, la función `imprimir_numeros()` no posee declaraciones locales.

Se desaconseja el uso de variables globales, salvo que su presencia sea estrictamente justificada, ya que impide la reutilización del código, entre otras cosas. En cambio, el uso de variables locales contribuye a una mejor legibilidad del programa y minimiza la probabilidad de errores por referencias incorrectas. Analicemos el ejemplo siguiente:

```

#include <stdio.h>

int suma();

/* Variables globales */
int operando1, operando2, resultado;

int main()
{
    operando1 = 2;
    operando2 = 3;
    resultado = suma();

    printf("%d + %d = %d \n", operando1, operando2, resultado);

    return 0;
}

int suma()
{
    int resultado;
    resultado = operando1 + operando2;
    return resultado;
}

```

Se trata de un código simple, donde se implementa una función llamada `suma()` para sumar dos valores enteros. Obsérvese la declaración de tres variables globales: `operando1`, `operando2` y `resultado`.

La función `suma()` toma el contenido de las variables globales `operando1` y `operando2`, y almacena la suma en `resultado`. Sin embargo, para deducir esto es necesario prestar atención al código, en especial a la implementación de `suma()`. Esto significa que con sólo mirar el prototipo de esa función no queda claro qué tipos de operandos acepta (enteros o reales) ni qué cantidad. El prototipado de funciones debe permitir que el que lee el código deduzca el



Se desaconseja el uso de variables globales, ya que impide la reutilización del código. En cambio, el uso de variables locales contribuye a una mejor legibilidad del programa y minimiza la probabilidad de errores por referencias incorrectas.

comportamiento de las funciones, sin necesidad de gastar tiempo tratando de entender sus particularidades de implementación.

No obstante, en el programa presentado hay un problema aún más grave: Un error de programación que no es detectado por el compilador. Si se mira con atención el cuerpo de `suma()`, se verá que la variable local se denomina `resultado`, mientras que la suma se almacena en la variable global `resultado`. Si bien esto funciona, el valor returnedo por `suma()` es incorrecto, es basura. Si no se utilizaran variables globales, el error hubiese sido detectado por el compilador, al no encontrar definida la variable `resultado`.


En el caso de las variables numéricas no inicializadas, no asumir que su valor por omisión es cero.

Es importante aclarar que si una función tiene una declaración local llamada `X`, y también existe una declaración global con ese nombre, entonces la local tiene mayor precedencia que la global.

El concepto de declaraciones globales y locales también se aplica al lenguaje Pascal. Toda variable o constante, o, incluso, todo tipo de dato declarado por el usuario que se encuentre dentro del ámbito de un procedimiento o una función, es local a esa rutina. Por su parte, el programa principal no acepta declaraciones locales, sino sólo globales. A continuación se muestra el ejemplo del “cálculo del máximo”, pero ahora en lenguaje Pascal:

```
program ejemplo;
function calcular_maximo(a, b, c: integer): integer;
{ Variable local de la función 'calcular_maximo()' }
var mayor: integer;
begin
  mayor := a;
  if (b > mayor) then
    mayor := b;
  if (c > mayor) then
    mayor := c;
  calcular_maximo := mayor;
end;

procedure imprimir_numeros(a, b, c: integer);
{ Procedimiento sin variables locales }
begin
  writeln('Usted ingresó: ', a, ' ', b, ' ', c);
end;

{ Variables globales }
var n1, n2, n3, maximo: integer;
begin
  write('Ingrese tres números enteros: ');
  readln(n1, n2, n3);
  imprimir_numeros(n1, n2, n3);
  maximo := calcular_maximo(n1,n2,n3);
  writeln('El máximo es: ', maximo);
end.
```

Es importante notar que si bien las variables `n1`, `n2`, `n3` y `maximo` son globales, su uso se restringe sólo al programa principal habiéndolas declarado luego de todos los procedimientos y las funciones.

3.4 Parámetros.

Los parámetros de una función constituyen la “vía” a través de la cual la rutina interactúa con el exterior, intercambiando datos. Puede ser una analogía útil indicar que los parámetros son para una función lo que una aduana para un país; por allí pasan todas las mercancías que entran y salen. El “contrabando”, por ejemplo, es la entrada y la salida de mercancías por fuera del sistema aduanero y, por lo tanto, el Estado no tiene control sobre esos productos. De igual manera, una función que recibe o retorna datos a través de mecanismos ajenos a sus parámetros o su valor de retorno pierde el control de lo que está intercambiando con el medio. Éste es un escenario típico del uso de variables globales.

Antes de continuar, es necesario hacer una aclaración. Si bien se los utiliza como sinónimos, parámetro y argumento no son lo mismo. El parámetro es el nombre de la variable que utiliza la función en forma interna para hacer uso de cada valor que le está pasando el que la llamó, mientras que el argumento es el valor en sí. También se suele hacer referencia a los parámetros como parámetros formales, y a los argumentos como parámetros reales. Cuando se llama a una función o a un procedimiento, los argumentos (o parámetros reales) deben ser coherentes en cantidad, tipo y orden respecto de los parámetros (o parámetros formales) especificados en la declaración de la rutina.

```
/* Llamada */           Parámetros reales (argumentos)
maximo = calcular_maximo (n1, n2, n3);

...
int calcular_maximo (int a, int b, int c)           Parámetros formales
{
    int mayor;

    mayor = a;
    if (b > mayor)
        mayor = b;
    if (c > mayor)
        mayor = c;

    return mayor;
}
```

Por lo general se conoce como pasaje de parámetros lo que en realidad es la copia de los argumentos de una función a una zona de memoria propia de ella, ubicada en la pila de memoria (o *stack*). Hay diferentes formas de llevar a cabo el pasaje de parámetros en la llamada a una función o a un procedimiento. En los casos típicos los lenguajes de programación suelen dar soporte a las siguientes:

- **Pasaje por valor:** Los argumentos son copiados en las variables que conforman la lista de parámetros. Así, si la función o el procedimiento modifican el contenido de sus parámetros, el argumento original no se altera.
- **Pasaje por referencia:** En lugar de una copia, se pasa la dirección de memoria de cada argumento (una “referencia”), de manera que la función o el procedimiento podrían modificar el valor original. Esto puede ser útil para que la rutina retorne datos a través de los parámetros.
- **Pasaje por nombre:** Dentro del cuerpo de la rutina, cada vez que se utilice un parámetro, será reemplazado textualmente por el argumento correspondiente. Es más fuerte que el pasaje por referencia, ya que no se accede al argumento a través de una referencia, sino que directamente se trabaja sobre él.

En C existe sólo una forma de pasaje de parámetros: El pasaje por valor. Esto explica que para que una función modifique el contenido de un argumento recibido, es necesario pasarle



Los parámetros constituyen la “vía” a través de la cual la rutina interactúa con el exterior, intercambiando datos.



El parámetro es el nombre de la variable que utiliza la función en forma interna para hacer uso de cada valor que le está pasando el que la llamó, mientras que el argumento es el valor en sí.

por valor la dirección de ese argumento. Al recibir la dirección, la rutina puede modificar el argumento en cuestión, lo que es equivalente al pasaje por referencia soportado por otros lenguajes, como Pascal. Analicemos el ejemplo siguiente, donde se implementa una función para intercambiar el contenido de dos variables:

```
#include <stdio.h>

/* Prototipo */
void intercambiar(char* a, char* b);

int main()
{
    char letral, letra2;

    letral = 'A';
    letra2 = 'B';

    intercambiar(&letral, &letra2);
    printf("%c %c \n", letral, letra2);

    return 0;
}

void intercambiar(char* a, char* b)
{
    char temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

La función `intercambiar()` intercambia el contenido de sus argumentos. Para esto es necesario que esa función pueda modificar los argumentos que recibe (de otra manera, el resultado no se vería reflejado afuera). Dado que en C los pasajes siempre son por valor, lo que se hace es pasar las direcciones de las variables `letral` y `letra2`, en lugar de su contenido. Para ello se utiliza el operador unario `&` que permite obtener la dirección de memoria de una variable. En el ejemplo anterior se hace uso de manejo de punteros, tema que se verá algunos capítulos más adelante. Se recomienda al lector no detenerse ahora tratando de entender esos puntos; es probable que queden más claros con un gráfico:

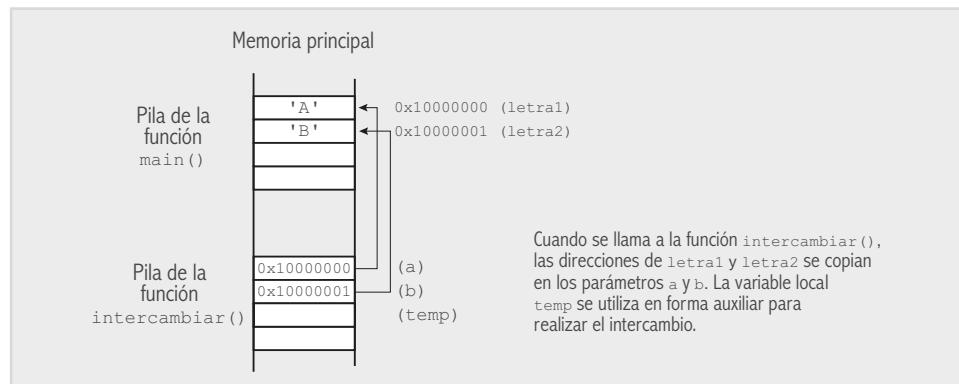


Fig. 3-2. Pasaje de parámetros en lenguaje C (pasaje por valor).

El lenguaje Pascal, en cambio, brinda soporte para el pasaje por valor y también por referencia. Este último se implementa mediante el uso de la palabra reservada `var`. A continuación se presenta el ejemplo dado antes (intercambio del contenido de dos variables) implementado en Pascal:

```
program pasaje_de_parametros;
procedure intercambiar(var a,b: char);
var temp: char;
begin
  temp := a;
  a := b;
  b := temp;
end;

var letra1, letra2: char;

begin
  letra1 := 'A';
  letra2 := 'B';

  intercambiar(letra1, letra2);
  writeln(letra1, ' ', letra2);
end.
```



Si bien para el programador resulta “transparente”, el uso de `var` para pasar un parámetro por referencia implica el empleo de punteros internamente. Aquí no hay magia; la única forma de alterar un argumento es pasar su dirección de memoria, su referencia.

Nótese que, en el caso de Pascal, no es necesario realizar un manejo explícito de punteros para implementar el pasaje por referencia. Es suficiente con anteponer la palabra reservada `var` a los parámetros en cuestión.

3.5 Argumentos por línea de comandos

De la misma manera que un procedimiento o una función pueden recibir argumentos como parámetros, la ejecución de un programa también puede recibir datos externos en forma de argumentos, lo que suele ser muy útil en muchas situaciones. Estos argumentos se pasan a través de la línea de comandos.

La línea de comandos es la interfaz entre el usuario y el sistema operativo, mediante la cual es posible ejecutar “comandos” (programas). En la actualidad, los sistemas operativos se “muestran” de una manera gráfica hacia el usuario, y un “doble-clic” es suficiente para lanzar una aplicación. Por detrás, la ejecución se lleva a cabo a través de la línea de comandos, con la posibilidad de pasar argumentos.

En lenguaje C el pasaje de argumentos por línea de comandos se lleva a cabo indicando una lista de parámetros a la función `main()`, que es la que se ejecutará primero. La forma de esta lista de parámetros no es arbitraria, sino que está compuesta por dos de ellos, `argc` y `argv`, como se muestra a continuación:

```
int main(int argc, char* argv[])
{
  ...
  return 0;
}
```



Por convención se utilizan los identificadores o nombres de parámetros argc y argv, cuyo significado es argument count y argument vector respectivamente.

El parámetro argc indica la cantidad de argumentos que recibe el programa, considerando el nombre del programa como el primero de ellos. Por su parte, argv es un arreglo (obsérvese el uso de []) de cadenas de caracteres (por ahora no conviene detenerse en el detalle de que char* es una cadena de caracteres; esto ya se desarrollará más adelante). De esta forma, cuando un programa comienza su ejecución, recibe uno o más argumentos en forma de cadenas de caracteres.

Para aclarar el uso de argumentos por línea de comandos, veamos el ejemplo siguiente, en el que el programa `programa_suma` recibe dos enteros como argumentos, los suma y muestra el resultado:

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

int main(int argc, char* argv[])
{
    int num1, num2;

    /* Se espera que reciba 3 argumentos: el nombre
     * del programa y los dos números enteros a sumar. */
    if (argc != 3)
    {
        printf("Modo de uso: %s <entero> <entero> \n", argv[0]);
        return 1;
    }

    printf("Bienvenido al programa %s \n\n", argv[0]);

    /* atoi() convierte una cadena de caracteres en un entero */
    num1 = atoi(argv[1]);
    num2 = atoi(argv[2]);

    printf("%d + %d = %d \n", num1, num2, num1+num2);

    return 0;
}
```



Un arreglo es un tipo de dato capaz de almacenar varios elementos del mismo tipo. En el caso particular del arreglo argv, todos los elementos son cadenas de caracteres. El concepto de arreglo se tratará con detalle en el capítulo 4.

En la imagen siguiente se puede observar un ejemplo de ejecución de este programa en la consola del sistema operativo Windows. El primer intento fue sin argumentos, mientras que en la segunda oportunidad se pasaron los valores 2 y 3.

```
C:\>programa_suma.exe
Modo de uso: programa_suma.exe <numero entero> <numero entero>
C:\>programa_suma.exe 2 3
Bienvenido al programa programa_suma.exe
2 + 3 = 5
C:\>_
```

Fig. 3-3. Ejecución de `programa_suma` en la consola del sistema operativo Windows.

Al comienzo del programa es muy saludable verificar si la cantidad de argumentos recibidos es la esperada. Para ello se consulta el contenido de `argc`. Luego, para acceder a los argumentos, se utiliza el arreglo `argv`, teniendo en cuenta que la primera posición (`argv[0]`) contiene el nombre del programa, la segunda (`argv[1]`) contiene el primer argumento, y así sucesivamente.

Por su parte, Pascal también brinda soporte para el pasaje de argumentos por línea de comandos, mediante las siguientes funciones estándar:

- `ParamCount()`: Devuelve la cantidad de argumentos que recibió el programa, sin considerar el nombre de éste. Es equivalente a `argc` en el caso de C (salvo porque `argc` considera el nombre del programa como el primer argumento).
- `ParamStr()`: Recibe un entero largo `n` como parámetro, y retorna el `n`-ésimo argumento como una cadena de caracteres. Si se llama a `ParamStr(0)`, retorna el nombre del programa, si se convoca a `ParamStr(1)`, retorna el primer argumento, y así sucesivamente.

A continuación se muestra el mismo ejemplo dado antes para el caso de C, pero ahora implementado en Pascal:

```
program ejemplo_argumentos;

var num1, num2: integer;
    error_code: integer;

begin
    if (ParamCount <> 2) then
        begin
            writeln('Modo de uso: ', ParamStr(0),
' <numero entero> <numero entero>');
            exit;{Fuerza la finalización y salida del programa, de
manera análoga a la sentencia 'return' de C}
        end;

    writeln('Bienvenido al programa ', ParamStr(0));

    { val() permite convertir una cadena }
    { de caracteres en su valor numérico. }
    val(ParamStr(1), num1, error_code);
    val(ParamStr(2), num2, error_code);

    writeln(num1, ' + ', num2, ' = ', num1+num2);

end.
```

3.6 Mapa de memoria

Se recomienda al lector no avanzar con el resto del capítulo si los temas anteriores no quedaron lo suficientemente claros, ya que a continuación se expondrán algunos conceptos de mayor complejidad. Más aún, eventualmente el lector podría pasar hacia el próximo capítulo, y en un futuro retomar lo que resta del presente.

Antes nos referimos a la memoria de la computadora afirmando, entre otras cosas, que es el sitio donde residen las variables. Lo cierto es que se puede indicar mucho más que eso. En todo sistema de cómputos pueden distinguirse una memoria principal (popularmente conocida



Descargue desde la Web
del libro el software para
programar en Cy Pascal.



Leer o escribir en memoria principal insume un tiempo en el orden de los nanosegundos (10⁻⁹), y en el orden de los milisegundos (10⁻³) para el caso del almacenamiento secundario. Si comparamos un acceso a memoria principal con una caminata de 400 m, entonces acceder a memoria secundaria equivaldría a viajar a la Luna.

como "memoria RAM") y una memoria secundaria o auxiliar (conformada por el disco duro, los discos ópticos, las memorias electrónicas externas, entre otros). La primera se caracteriza por ser una memoria rápida para lectura/escritura y volátil, en cuanto que los datos que almacena se conservan mientras la memoria reciba alimentación eléctrica. Por su parte, el tiempo de acceso (latencia) a un medio de almacenamiento secundario es varios órdenes de magnitud superior, pero la información almacenada persiste aún si el dispositivo no recibiese alimentación eléctrica.

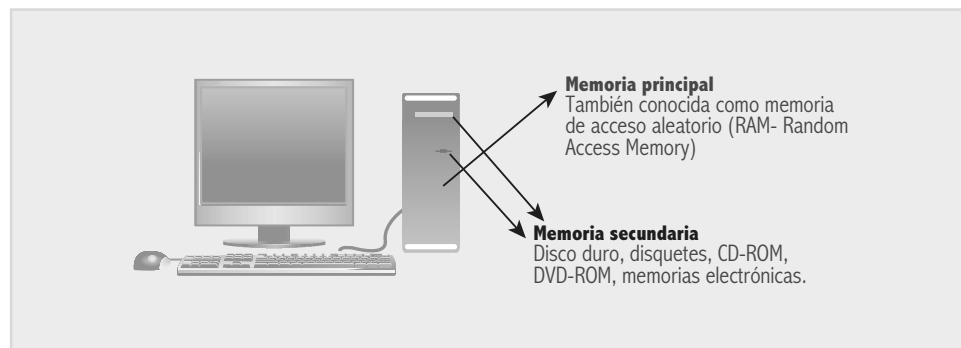


Fig. 3-4. Memoria principal y memoria secundaria.

Durante la ejecución de un programa (proceso), todos los datos manipulados (incluido el propio código del programa compilado) se alojan en la memoria principal, ya que ésta ofrece menor latencia respecto de los medios secundarios. Sin embargo, si fuese necesario darle persistencia en el tiempo a esos datos, en el programa deberá indicarse en forma explícita el acceso a la memoria secundaria (por ejemplo, para almacenar resultados en el disco duro; esto se tratará algunos capítulos más adelante, cuando se introduzca el concepto de archivo).

En la memoria principal podemos distinguir diferentes "regiones", como se muestra en la imagen siguiente:

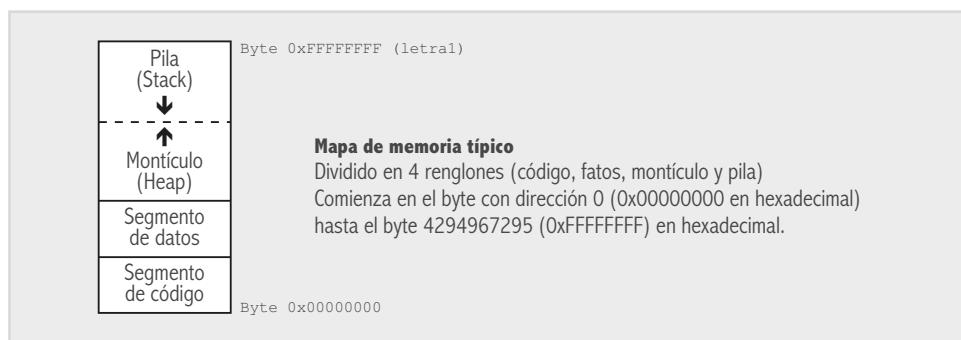


Fig. 3-5. "Regiones" de la memoria principal.

Empezando por las direcciones más bajas, el segmento de código es la porción de la memoria donde se carga el programa que comenzará a ejecutarse. El segmento de datos alberga las declaraciones globales (por ejemplo, variables y constantes globales). El montículo (más popularmente conocido como *heap*) es la región de memoria que se toma y se libera de manera dinámica durante la ejecución del programa (esto se tratará en detalle en algunos capítulos). Por último, la pila es donde se "apilan" porciones de memoria pertenecientes a funciones y/o procedimientos, cuando se los llama. En el gráfico anterior las flechas indican en qué sentido crecen el montículo y la pila (¿Puede deducir por qué lo hacen en direcciones opuestas?).

En este momento interesa analizar el comportamiento de la pila, ya que es la porción de memoria utilizada por procedimientos y funciones. Al inicio, cuando el programa comienza a ejecutarse, podemos indicar que la pila está “vacía” (esto no es estrictamente cierto en C, donde el programa comienza con la ejecución de una función `-main()` – y haría uso de la pila; sin embargo, ignoremos este detalle ahora). Cuando se llama a una función, ésta tendrá asignada una porción dentro del *stack* donde almacenará sus variables locales y los valores de los parámetros que recibe por valor. Cuando la rutina termina de ejecutarse, esa porción de la pila se libera. Analicemos el ejemplo siguiente, un tanto más complejo:

```
#include <stdio.h>

#define FALSE 0
#define TRUE 1

int carga_dato();
int valida_dato(int);

int main()
{
    int dato;

    dato = carga_dato();
    printf("Dato ingresado: %d \n", dato);

    return 0;
}

int carga_dato()
{
    int numero;
    do
    {
        printf("Ingrese un número entero positivo: \n");
        scanf("%d", &numero);
    }
    while (valida_dato(numero) == FALSE);

    return numero;
}

int valida_dato(int dato)
{
    if (dato >= 0)
        return TRUE;
    return FALSE;
}
```

El programa presentado tiene un comportamiento muy simple: Solicita un número entero positivo por entrada estándar y lo valida (comprueba que sea mayor o igual a cero). Lo interesante de este ejemplo es que dentro de la función `carga_dato()` se hace una llamada a otra función, `valida_dato()`. Estamos en presencia de un “anidamiento” de llamadas, que se “apilarán” en el *stack*, como se muestra en la imagen siguiente:

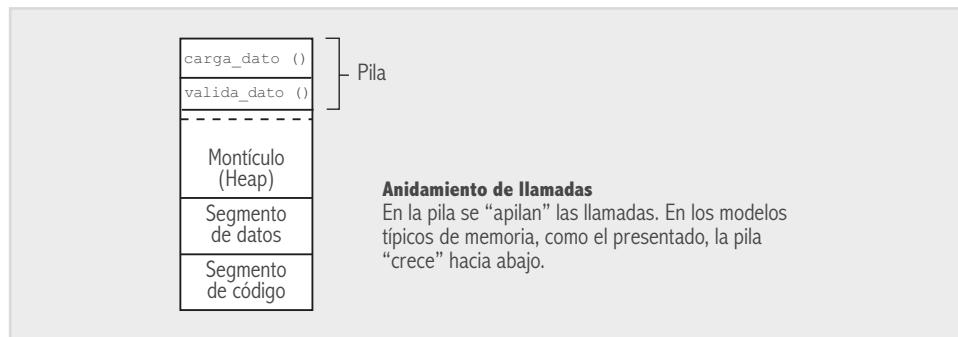


Fig. 3-6. "Anidamiento" de llamadas.

El anidamiento de llamadas puede ser tan profundo como se desee, limitado por el tamaño de la pila. Si ésta se desborda, estamos en presencia del error *stack overflow*. Por último, la pila será deshecha en el sentido inverso respecto de cómo se armó: Primero se libera la región de *valida_dato()* y luego la de *carga_dato()*.

3.7 Consideraciones de desempeño.

De lo que se acaba de exponer puede deducirse que la llamada a un procedimiento o función es una acción poco “económica” para el programa, que debe realizar un trabajo importante para tomar una porción de la pila y copiar allí los valores de los argumentos. Más aún, y sin intenciones de explorar cuestiones más complejas, la llamada a una subrutina obliga al procesador a realizar un “salto” en el flujo de instrucciones que se viene ejecutando, lo que es muy perjudicial para el desempeño. Miremos con atención los ejemplos siguientes:

```
#include <stdio.h>
int suma(int, int);
int main()
{
    int operando1, operando2, resultado;
    operando1 = 5;
    operando2 = 10;
    resultado = suma(operando1,operando2);
    printf("resultado = %d \n", resultado);
    return 0;
}

int suma(int a, int b)
{
    return a + b;
}
```

```
#include <stdio.h>
int main()
{
    int operando1, operando2, resultado;
    operando1 = 5;
    operando2 = 10;
    resultado = operando1 + operando2;
    printf("resultado = %d \n", resultado);
    return 0;
}
```

La diferencia más significativa entre ambos está en que el programa de la izquierda llama a una función para efectuar la suma, mientras que el de la derecha realiza la suma de manera explícita. Si para llevar a cabo una operación tan simple es necesario llamar a una función, con todo lo que eso implica, entonces no estaríamos amortizando ese costo, y sería más conveniente reemplazar la llamada por la suma concreta, como se muestra en el ejemplo de la derecha.

Sin embargo, tampoco es aconsejable desmodularizar un programa en busca de un mayor desempeño, porque estaríamos perdiendo todas las ventajas propias de la modularización que se describieron en la primera parte de este capítulo. Lo que se recomienda en ese escenario es utilizar una optimización disponible en algunos compiladores, conocida como expansión en línea (o *Inlining*). Esta optimización consiste en que el compilador reemplace la invocación a una función por el código de ella, para evitar el costo de la llamada. Esta tarea automática se lleva a cabo en tiempo de compilación; por lo tanto, el programa no fue desmodularizado desde el punto de vista del programador.

La expansión en línea está soportada por los compiladores de C mediante el uso de la palabra reservada `inline`, antepuesta al encabezado de la función, que “aconseja” al compilador expandir esa función:

```
#include <stdio.h>

inline int suma(int, int);

int main(int argc, char *argv[])
{
    int operando1, operando2, resultado;

    operando1 = 5;
    operando2 = 10;

    resultado = suma(operando1, operando2);
    printf("resultado = %d \n", resultado);

    return 0;
}

inline int suma(int a, int b)
{
    return a + b;
}
```

El uso de `inline` no está soportado por el estándar ANSI por lo que, a menudo, se suelen utilizar macros (recordar que la macro será reemplazada por el preprocesador en cada punto del programa donde se haga referencia a ella, lo que es equivalente a expandir en línea):

```
#include <stdio.h>

#define SUMA(OP1,OP2) ((OP1)+(OP2))

int main(int argc, char *argv[])
{
    int operando1, operando2, resultado;

    operando1 = 5;
    operando2 = 10;

    resultado = SUMA(operando1,operando2);
    printf("resultado = %d \n", resultado);

    return 0;
}
```



La expansión en línea consiste en que el compilador reemplace la invocación a una función por el código de ella, para evitar el costo de la llamada. Esta tarea automática se lleva a cabo en tiempo de compilación.



Memoria Caché.

Pequeña memoria, de gran velocidad de acceso, empotrada dentro del mismo circuito integrado del procesador, que le permite a éste disminuir la cantidad de lecturas a memoria principal (más lenta) en busca de instrucciones para ejecutar.

El uso de la técnica de expansión en línea, como búsqueda de mejor desempeño, debe ser cuidadoso y racional. La expansión en línea consiste en reemplazar todas las llamadas a una función por el cuerpo de ella, duplicando código y aumentando el tamaño del archivo ejecutable generado. Esto puede tener, como efecto secundario, un aumento de las fallas en la memoria caché de instrucciones, con lo que no sólo no mejoraríamos la *performance*, sino que la perjudicaríamos. Por esta razón es que suele desaconsejarse el uso del modificador `inline`, ya que el compilador por lo general tiene la “inteligencia” suficiente como para aplicar este tipo de optimización según su criterio (que suele ser más acertado que el del programador).

3.8 Resumen.

Conforme nuestros programas aumentan en tamaño y se tornan más y más complejos, se vuelve imperiosa la necesidad de particionarlos en módulos más pequeños, que resuelvan tareas simples y concretas, y que puedan depurarse con mayor facilidad. Esta estrategia de programación, habitualmente conocida como “dividir y conquistar”, implica particionar un problema en otros más pequeños y más fáciles de resolver en forma individual (más “manejables”), que, en conjunto, resuelven el problema original. Aplicado a la programación, permite descomponer un software en módulos más pequeños, de manera que un programa con miles o millones de líneas de código pueda elaborarse a partir de un conjunto de subprogramas, cada uno más pequeño y más fácil de programar y mantener.

Cada subprograma debe implementar una funcionalidad concreta y bien definida; esto significa que sus límites deben estar claros. Así, la calidad de la modularización puede expresarse según sus grados de cohesión y acoplamiento: El primero se refiere a la afinidad de los elementos internos de un módulo, mientras que el segundo tiene que ver con el grado de dependencia entre módulos (si un módulo puede ejecutar su tarea por sí solo, con independencia de los demás, entonces el acoplamiento es bajo).

Además, cada subprograma recibirá datos desde el exterior (entradas) que, una vez procesados, generan un resultado que el módulo retorna a su entorno (salida). Esto define la interfaz del módulo con el mundo exterior.

En lenguaje C, la modularización se implementa con funciones: Subprogramas que procesan un conjunto de parámetros de entrada y generan un resultado. En Pascal, además de funciones, el programador también cuenta con procedimientos (análogos a una función, salvo porque no se espera que generen resultado alguno hacia quien los invocó).

Los parámetros de un subprograma (función o procedimiento) constituyen la “vía” a través de la cual la rutina interactúa con el exterior, intercambiando datos. Cuando se llama a una función o un procedimiento, hay tres alternativas para asignar valores concretos a los parámetros: Por valor (se trabaja sobre una copia de la variable, por lo tanto, no se altera el valor original), por referencia (se trabaja sobre la variable original por medio de una referencia a ella) y por nombre (se trabaja en forma directa sobre la variable original). Un subprograma que recibe o retorna datos mediante mecanismos ajenos a sus parámetros o su valor de retorno pierde el control de lo que está intercambiando con el entorno. Éste es un escenario típico del uso de variables globales.

Por último, en este capítulo también se realizó una introducción al concepto de mapa de memoria. En todo sistema de cómputos pueden distinguirse una memoria principal y una secundaria (discos duros, discos ópticos, memorias electrónicas externas, entre otros). La memoria principal se caracteriza por ser rápida para lectura/escritura y volátil, en cuanto a que los datos que almacena se conservan mientras se reciba alimentación eléctrica. Durante la ejecución de un programa todos los datos que son manipulados (incluido el mismo código del

programa compilado) se alojan en memoria principal, en la que pueden distinguirse diferentes “regiones”:

- **Segmento de código:** Es la porción de la memoria donde se carga el programa que comenzará a ejecutarse.
- **Segmento de datos:** Alberga las declaraciones globales (por ejemplo, variables y constantes globales).
- **Montículo** (o *heap*): Es la región de memoria que se toma y se libera en forma dinámica durante la ejecución del programa.
- **Pila** (o *stack*): Es donde se “apilan” porciones de memoria pertenecientes a funciones, procedimientos, o ambos, cuando se los llama.

3.9 Problemas propuestos.

- 1) Escribir un programa que pida una cadena de caracteres (cuya longitud máxima sea de 80 caracteres) y devuelva la cadena escrita al revés.

Ayuda: Para saber la longitud de la cadena se puede usar la función `strlen()` de la librería `string.h`.

- 2) Realizar un programa que lea una cadena de caracteres de una longitud menor que 80 y visualice los caracteres de la siguiente forma:

Primero, último, segundo, penúltimo, tercero, antepenúltimo, ...

- 3) Escribir una función que cambie las letras mayúsculas de una cadena a minúsculas y viceversa. El programa principal pedirá una cadena por pantalla y se la pasará a esa función esperando el resultado correcto que se mostrará por pantalla.

- 4) Escribir un programa que pida primero un carácter por teclado y que luego pida una cadena. El programa calculará cuántos caracteres tiene la cadena hasta que lea el carácter introducido primero. Se deberá mostrar un mensaje en pantalla con el número de caracteres introducidos hasta llegar al carácter primero.

- 5) Escribir un programa que cuente el número de letras, dígitos y signos comunes de puntuación de una cadena introducida por teclado.

Ayuda: Para saber si un carácter es numérico, es preciso comparar que su valor es mayor que ‘0’ y menor que ‘9’; para saber si es alfabético, se debe comprobar que está entre ‘a’ y ‘z’, y considerar signos de puntuación al resto de los caracteres.

Nota: No considerar la ñ ni las letras acentuadas, ya que tienen códigos ASCII fuera del rango a-z.

- 6) Realizar un programa que lea una cadena de caracteres con espacios en blanco excesivos: Elimine los espacios en blanco iniciales y finales, y sólo deje uno entre cada dos palabras.
- 7) Diseñar un programa que pida una cadena de caracteres y devuelva otra sin signos de puntuación ni números. La cadena devuelta debe tener todos los caracteres en mayúsculas.
- 8) Realizar un programa que escriba todos los números enteros menores que un cierto entero N y que a su vez sean múltiplos de dos números enteros A y B introducidos por teclado. Utilizar para ello una función que admita dos parámetros I, J, e indique si I es múltiplo de J.
- 9) Escribir una función (con su correspondiente programa de prueba) que tome un valor entero y devuelva el número con sus dígitos en sentido inverso. Por ejemplo, dado el número 7631, la función deberá devolver 1367.
- 10) Escribir una función que tenga tantas líneas en blanco como se haya pedido con anterioridad al usuario en el programa principal.

3.10 Problemas resueltos.

1- Cree un algoritmo para convertir kilómetros en millas. La conversión se debe llevar a cabo mediante una función. También implemente una subrutina para mostrar una leyenda con el resultado.

Solución en lenguaje C

```
#include <stdio.h>

#define FACTOR 0.6214

/* Prototipos */
float convertir(int);
void imprimir_resultado(int,float);

int main (void)
{
    int kms;
    float millas;

    printf("Conversión de kilómetros a millas\n");
    printf("-----\n\n");

    printf("Introduzca la cant. de km a convertir: ");
    scanf("%i", &kms);

    millas = convertir(km);
    imprimir_resultado(km, millas);

    return 0;
}

float convertir(int km)
{
    return FACTOR * km;
}

void imprimir_resultado(int km, float millas)
{
    printf("%d km equivalen a %.2f millas", km, millas);
}
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

Solución en lenguaje Pascal

```
PROGRAM alg_01_cap03;

CONST FACTOR = 0.6214;

{ Función: convertir()
{ Recibe como parámetro una cantidad expresada en kilómetros y }
{ retorna su equivalente en millas. }
FUNCTION convertir(kms: INTEGER): REAL;

BEGIN
    convertir := FACTOR * km;
```

```

END;

{ Procedimiento: imprimir_resultado() }
{ Recibe una cantidad expresada en kilómetros y su equivalente en }
{ millas, y muestra una leyenda informando la relación. }
PROCEDURE imprimir_resultado(kms: INTEGER; millas: REAL);

BEGIN
  WRITELN(kms, ' km equivalen a ', millas:4:2, ' millas');
END;

VAR km: INTEGER;
    millas: REAL;

BEGIN
  WRITELN('Conversión de kilómetros a millas');
  WRITELN('-----');
  WRITELN;
  WRITE('Introduzca la cant. de km a convertir: ');
  READLN(km);

  millas := convertir(km);
  imprimir_resultado(km, millas);
END.

```

2- Cree un algoritmo que calcule e imprima una tabla de funciones trigonométricas.

Solución en lenguaje C

```

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

#define PI 3.1415

/* Prototipos */
void mostrar_tabla(float, float, float);

int main()
{
    float valor_inicial, valor_final, inc;
    printf("Tabla de funciones trigonométricas\n");
    printf("-----\n\n");

    printf("Introduzca el valor inicial: ");
    scanf("%f", &valor_inicial);

    do
    {
        printf("Introduzca el valor final: ");
        scanf("%f", &valor_final);
    }

```

```

while (valor_final <= valor_inicial);

do
{
    printf("Introduzca el incremento: ");
    scanf("%f", &inc);
}
while (inc <= 0);

mostrar_tabla(valor_inicial, valor_final, inc);

return 0;
}

void mostrar_tabla(float valor_inicial, float valor_final, float inc)
{
    float grados, radianes;

    printf("  Grados      Seno      Coseno      Tangente\n");
    printf("-----\n");
    for (grados = valor_inicial; grados <= valor_final; grados += inc)
    {
        radianes = grados * PI / 180; /* Conversión grados a radianes */
        printf("%8.2f", grados);
        printf("%10.4f %10.4f %10.4f\n", sin(radianes), cos(radianes),
tan(radianes));
    }
}
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_02_cap03;

{ Para utilizar las funciones sin(), cos() y tan() }
USES math;

PROCEDURE mostrar_tabla(valor_inicial, valor_final, inc: REAL);
VAR grados, radianes: REAL;
BEGIN
    WRITELN('  Grados      Seno      Coseno      Tangente');
    WRITELN('-----');

    grados := valor_inicial;
    WHILE (grados <= valor_final) DO
    BEGIN
        radianes := grados * PI / 180; { Conversión grados a radianes }
        WRITE(grados:10:4);
        WRITELN(sin(radianes):10:4, cos(radianes):10:4, tan(radianes):10:4);
        grados := grados + inc;
    END;
END;

```

```

VAR valor_inicial, valor_final, inc: REAL;
{ Programa principal }
BEGIN
  WRITELN('Tabla de funciones trigonométricas');
  WRITELN('-----');
  WRITELN;

  WRITE('Introduzca el valor inicial: ');
  READLN(valor_inicial);

  REPEAT
    WRITE('Introduzca el valor final: ');
    READLN(valor_final);
  UNTIL (valor_final > valor_inicial);

  REPEAT
    WRITE('Introduzca el incremento: ');
    READLN(inc);
  UNTIL (inc > 0);

  mostrar_tabla(valor_inicial, valor_final, inc);

END.

```

- 3- Programe un algoritmo que calcule el área de una corona circular a partir de una rutina que calcule el área de un círculo. Utilice paso de parámetro por valor y por referencia.

Solución en lenguaje C

```

#include <stdio.h>

#define PI 3.1415

/* Función: area()
 * Recibe como primer parámetro una referencia (puntero) a la */
/* variable donde se almacenará el área calculada (equivale a un */
/* pasaje por referencia). Como segundo parámetro recibe el radio */
/* en cm (pasaje por valor). */
void area(float*, float);

/* programa principal */
int main ()
{
    float radio_ext, radio_int, area_ext, area_int, area_corona;

    printf("Área de una corona circular\n");
    printf("-----\n\n");

    do
    {
        printf("Introduzca el radio exterior (en cm): ");
        scanf("%f", &radio_ext);
    }

```

```

while (radio_ext <= 0.0);

/* Cálculo del área exterior */
area(&area_ext , radio_ext);

do
{
    printf("Introduzca el radio interior (en cm): ");
    scanf("%f", &radio_int);
}
while ( (radio_int <= 0.0) || (radio_int > radio_ext) );

/* Cálculo del área interior */
area(&area_int, radio_int);

area_corona = area_ext - area_int;

printf("El área de la corona circular es %4.2f cm2 \n", area_corona);

return 0;
}

void area(float* area_cir, float radio)
{
    *area_cir = PI * radio * radio;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_03_cap03;

CONST PI = 3.1415;

{ Procedimiento: area()
  Recibe como primer parámetro una referencia (puntero) a la }
{ variable donde se almacenará el área calculada (pasaje por }
{ referencia). Como segundo parámetro recibe el radio en cm }
{ (pasaje por valor). }

PROCEDURE area(VAR area_cir: REAL; radio: REAL);

BEGIN
    area_cir := PI * radio * radio;
END;

VAR radio_ext, radio_int, area_ext, area_int, area_corona: REAL;
{ Programa principal }
BEGIN

    WRITELN('Área de una corona circular');
    WRITELN('-----');

    REPEAT
        WRITE('Introduzca el radio exterior (en cm): ');
        READLN(radio_ext);

```

```

UNTIL (radio_ext > 0.0);

{ Cálculo del área exterior }
area(area_ext , radio_ext);

REPEAT
  WRITE('Introduzca el radio interior (en cm): ');
  READLN(radio_int);
UNTIL ((radio_int > 0.0) and (radio_int <= radio_ext));

{ Cálculo del área interior }
area(area_int , radio_int);

area_corona := area_ext - area_int;

WRITELN('El área de la corona circular es ', area_corona:4:2, ' cm2');

END.

```

- 4- Diseñe un algoritmo que calcule el área de una corona circular a partir de una rutina que calcule el área de un círculo. Utilice solamente paso por valor, y que el resultado se retorne a través del nombre de la función.

Solución en lenguaje C

```

#include <stdio.h>

#define PI 3.1415

/* Función: area()
 * Recibe como primer parámetro el radio en cm (por valor) y */
/* retorna el área por el nombre. */
float area(float);

/* programa principal */
int main ()
{
  float radio_ext, radio_int, area_ext, area_int, area_corona;

  printf("Área de una corona circular\n");
  printf("-----\n\n");

  do
  {
    printf("Introduzca el radio exterior (en cm): ");
    scanf("%f", &radio_ext);
  }
  while (radio_ext <= 0.0);

  /* Cálculo del área exterior */
  area_ext = area(radio_ext);

  do
  {
    printf("Introduzca el radio interior (en cm): ");

```

```

        scanf("%f", &radio_int);
    }
    while ( (radio_int <= 0.0) || (radio_int > radio_ext) );
    /* Cálculo del área interior */
    area_int = area(radio_int);

    area_corona = area_ext - area_int;

    printf("El área de la corona circular es %4.2f cm2 \n", area_corona);

    return 0;
}

float area(float radio)
{
    return PI * radio * radio;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_04_cap03;

CONST PI = 3.1415;

{ Procedimiento: area()
{ Recibe como primer parámetro el radio en cm (por valor) y }
{ retorna el área por el nombre. }
FUNCTION area(radio: REAL): REAL;

BEGIN
    area := PI * radio * radio;
END;

VAR radio_ext, radio_int, area_ext, area_int, area_corona: REAL;
{ Programa principal }
BEGIN
    WRITELN('Área de una corona circular');
    WRITELN('-----');

    REPEAT
        WRITE('Introduzca el radio exterior (en cm): ');
        READLN(radio_ext);
    UNTIL (radio_ext > 0.0);

    { Cálculo del área exterior }
    area_ext := area(radio_ext);

    REPEAT
        WRITE('Introduzca el radio interior (en cm): ');
        READLN(radio_int);
    UNTIL ((radio_int > 0.0) and (radio_int <= radio_ext));

    { Cálculo del área interior }

```

```

area_int := area(radio_int);
area_corona := area_ext - area_int;
WRITELN('El área de la corona circular es ', area_corona:4:2, ' cm2');
END.

```

5- Programe un algoritmo que calcule la integral definida de una función f () entre dos puntos. Utilice paso de parámetro por valor.

Solución en lenguaje C

```

#include <stdio.h>

/* Función: f()
 * Representa la función matemática a evaluar. En este caso */
/* utilizamos f(x) = 2x, pero se puede sustituir por cualquier */
/* otra. */
float f(float x);

/* Función: integrar()
 * Como primer parámetro recibe el límite inferior y como */
/* segundo parámetro, el límite superior. Calcula la integral */
/* entre estos puntos, utilizando como incremento el valor */
/* recibido como tercer parámetro. */
float integrar(float, float, float);

/* Programa Principal */
int main ()
{
    float lim_inf, lim_sup, inc;
    printf("Cálculo de una integral definida\n");
    printf("-----\n\n");
    printf("Introduzca el límite inferior: ");
    scanf("%f", &lim_inf);
    do
    {
        printf("Introduzca el límite superior: ");
        scanf("%f", &lim_sup);
    }
    while (lim_sup < lim_inf);
    do
    {
        printf("Introduzca el incremento: ");
        scanf("%f", &inc);
    }
    while (inc <= 0);
    float resultado = integrar(lim_inf, lim_sup, inc);
    printf("El resultado de la integral es: %f\n", resultado);
}

```

```

printf("Valor de la integral = %5.2f\n",
       integrar(lim_inf, lim_sup, inc) );
return 0;
}

float f(float x)
{
    return 2 * x;
}

float integrar(float lim_inf, float lim_sup, float inc)
{
    float integral, x;
    x = lim_inf;
    integral = 0.0;
    while (x < (lim_sup-lim_inf))
    {
        integral += inc * f(x + inc/2.0);
        x += inc;
    }
    integral += (lim_sup - x) * f((lim_inf+x)/2.0);
    return integral;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_05_cap03;

{ Función: f() }
{ Representa la función matemática a evaluar. En este caso }
{ utilizamos f(x) = 2x, pero se puede sustituir por cualquier }
{ otra. }

FUNCTION f(x: REAL): REAL;

BEGIN
    f := 2 * x;
END;

{ Función: integrar() }
{ Como primer parámetro recibe el límite inferior y como }
{ segundo parámetro, el límite superior. Calcula la integral }
{ entre estos puntos, utilizando como incremento el valor }
{ recibido como tercer parámetro. }

FUNCTION integrar(lim_inf, lim_sup, inc: REAL): REAL;

VAR integral, x: REAL;

BEGIN
    x := lim_inf;
    integral := 0.0;
    WHILE (x < (lim_sup-lim_inf)) DO

```

```

BEGIN
    integral := integral + inc * f(x + inc/2.0);
    x := x + inc;
END;

integral := integral + (lim_sup - x) * f((lim_inf+x)/2.0);
integrar := integral;
END;

VAR lim_inf, lim_sup, inc: REAL;
{ Programa Principal }
BEGIN

WRITELN('Cálculo de una integral definida');
WRITELN('-----');

WRITE('Introduzca el límite inferior: ');
READLN(lim_inf);

REPEAT
    WRITE('Introduzca el límite superior: ');
    READLN(lim_sup);
UNTIL (lim_sup >= lim_inf);

REPEAT
    WRITE('Introduzca el incremento: ');
    READLN(inc);
UNTIL (inc > 0);

WRITELN('Valor de la integral = ',
        integrar(lim_inf, lim_sup, inc):5:2 );

END.

```

- 6- Cree un algoritmo que calcule la integral definida de una función $f()$ entre dos puntos. Los límites inferior y superior, y el incremento, se reciben como argumentos por línea de comandos.

Solución en lenguaje C

```

#include <stdio.h>
#include <stdlib.h>
/* Función: f() */ 
/* Representa la función matemática a evaluar. En este caso */
/* utilizamos  $f(x) = 2x$ , pero se puede sustituir por cualquier */
/* otra. */ 
float f(float x);

/* Función: integrar() */ 
/* Como primer parámetro recibe el límite inferior y como */
/* segundo parámetro, el límite superior. Calcula la integral */
/* entre estos puntos, utilizando como incremento el valor */
/* recibido como tercer parámetro. */ 

```

```

float integrar(float, float, float);

/* Programa Principal */
int main (int argc, char *argv[])
{
    float lim_inf, lim_sup, inc;

    /* La cantidad de parámetros recibida es la correcta? */
    if (argc != 4)
    {
        printf("Modo de uso: %s <lim_inf> <lim_sup> <inc>\n", argv[0]);
        return 1;
    }

    /* Convirtiendo los parámetros de cadenas a float */
    lim_inf = strtod(argv[1], NULL);
    lim_sup = strtod(argv[2], NULL);
    inc = strtod(argv[3], NULL);

    printf("Cálculo de una integral definida\n");
    printf("-----\n\n");

    printf("Valor de la integral = %5.2f\n",
           integrar(lim_inf, lim_sup, inc) );

    return 0;
}

float f(float x)
{
    return 2 * x;
}

float integrar(float lim_inf, float lim_sup, float inc)
{
    float integral, x;

    x = lim_inf;
    integral = 0.0;

    while (x < (lim_sup-lim_inf))
    {
        integral += inc * f(x + inc/2.0);
        x += inc;
    }

    integral += (lim_sup - x) * f((lim_inf+x)/2.0);

    return integral;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_06_cap03;
{ Función: f() }
{ Representa la función matemática a evaluar. En este caso }

```

```

{ utilizamos f(x) = 2x, pero se puede sustituir por cualquier }
{ otra. }
FUNCTION f(x: REAL): REAL;

BEGIN
  f := 2 * x;
END;

{ Función: integrar()
{ Como primer parámetro recibe el límite inferior y como }
{ segundo parámetro, el límite superior. Calcula la integral }
{ entre estos puntos, utilizando como incremento el valor }
{ recibido como tercer parámetro. }
FUNCTION integrar(lim_inf, lim_sup, inc: REAL): REAL;

VAR integral, x: REAL;

BEGIN
  x := lim_inf;
  integral := 0.0;
  WHILE (x < (lim_sup-lim_inf)) DO
    BEGIN
      integral := integral + inc * f(x + inc/2.0);
      x := x + inc;
    END;
  integral := integral + (lim_sup - x) * f((lim_inf+x)/2.0);
  integrar := integral;
END;

VAR lim_inf, lim_sup, inc: REAL;
error_code: INTEGER;

{ Programa Principal }
BEGIN

{ La cantidad de parámetros recibida es la correcta? }
IF (ParamCount <> 3) THEN
BEGIN
  WRITELN('Modo de uso: ', ParamStr(0), ' <lim_inf> <lim_sup> <inc>');
  EXIT;
END;

{ Convirtiendo los parámetros de cadenas a float }
val(ParamStr(1), lim_inf, error_code);
val(ParamStr(2), lim_sup, error_code);
val(ParamStr(3), inc, error_code);

WRITELN('Cálculo de una integral definida');
WRITELN('-----');
WRITELN('Valor de la integral = ',
       integrar(lim_inf, lim_sup, inc):5:2 );

END.

```

3.11 Contenido de la página Web de apoyo.....



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- Función y procedimiento.



Autoevaluación.



Video explicativo (04:39 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas.*



Presentaciones. *

4

Tipos estructurados homogéneos Vectores y matrices

Contenido

4.1 Introducción	90
4.2 Arreglos lineales.....	90
4.3 Declaración y uso de arreglos lineales	91
4.4 Arreglos multidimensionales	94
4.5 Arreglos como parámetros de subprogramas ...	96
4.6 Cadenas de caracteres.....	100
4.7 Enumeraciones.....	102
4.8 Resumen.....	103
4.9 Problemas propuestos.....	104
4.10 Problemas resueltos.....	105
4.11 Contenido de la página Web de apoyo.....	124

Objetivos

- Introducir el concepto de conjuntos de datos de una dimensión o vectores (arreglos lineales).
- Presentar conjuntos de datos de más de una dimensión o matrices (arreglos multidimensionales).



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.



Un arreglo lineal es un tipo de dato que se construye a partir de elementos más simples, cada uno de ellos identificado con un índice

4.1 Introducción.

Los ejemplos presentados en los capítulos anteriores trabajaban con datos simples, como números (enteros y reales) y caracteres. Sin embargo, en la práctica suele ser necesario procesar conjuntos de datos, como podrían ser listas de clientes, cuentas bancarias, alumnos, etc. Por ejemplo, ¿cómo podría implementarse un algoritmo que recorra el listado de clientes de una compañía para calcular el monto total adeudado por todos ellos? Está claro que sería necesario utilizar una estructura de control repetitiva que itere sobre cada cliente, consultando su deuda y acumulándola en una variable. No obstante, ¿cómo mantenemos los datos de, por ejemplo, 5 000 clientes en el programa? ¿Significa que deben declararse 5 000 variables, una para cada cliente? Como queda expuesto en esta breve reflexión, es necesario que el lenguaje de programación brinde soporte para mantener conjuntos de datos (no tan sólo variables simples como en los capítulos anteriores). Este soporte está dado por lo que se conoce como vectores (arreglos lineales) y matrices (arreglos multidimensionales), y que trataremos a continuación.

4.2 Arreglos lineales

Un arreglo lineal (también conocido como vector) es un tipo de dato que se construye a partir de elementos más simples, cada uno de ellos identificado con un índice. Por ejemplo, en el gráfico siguiente vemos un arreglo de 7 caracteres, donde el primero de ellos ('B') tiene subíndice 0, y el último ('P'), subíndice 6.

'B'	'V'	'H'	'V'	'T'	'U'	'P'
0	1	2	3	4	5	6

Se puede acceder a cada uno de los elementos que lo componen, tanto para lectura como para escritura, a partir del subíndice que lo identifica. En este sentido, la palabra "arreglo" se usa como sinónimo de orden, ya que queda clara la diferencia respecto de un conjunto no ordenado:

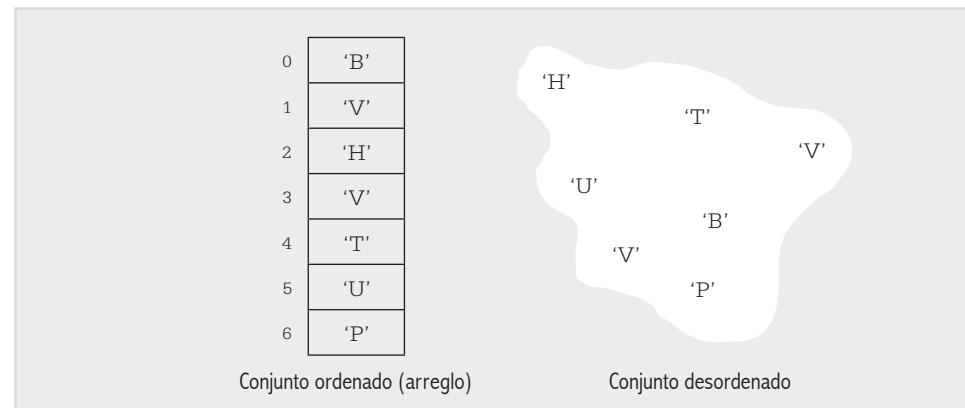


Fig. 4-1. Diferencia entre conjunto ordenado y conjunto no ordenado.



Un arreglo:

Es una estructura homogénea.
Es una estructura lineal de acceso directo.
Es una estructura estática.

Un arreglo posee las características siguientes:

- **Es una estructura homogénea:** Esto quiere decir que todos los datos almacenados en esa estructura son del mismo tipo (por ejemplo, todos de tipo entero o todos de tipo carácter, o todos reales, etc.).
- **Es una estructura lineal de acceso directo:** Esto significa que se puede acceder a los datos en forma directa, con sólo proporcionar su posición (en forma de subíndice).

- Es una estructura estática:** Su tamaño (número y tipo de elementos) se define en tiempo de compilación y no cambia durante la ejecución del programa. Esto es en particular cierto para los arreglos que manipularemos en el presente capítulo. En capítulos posteriores, cuando se trate el manejo de memoria dinámica, veremos arreglos cuyo tamaño puede modificarse en tiempo de ejecución.

4.3 Declaración y uso de arreglos lineales.

Los lenguajes de programación empleados en este libro, C y Pascal, brindan soporte para la construcción y el uso de arreglos lineales y multidimensionales. En el caso de C, un vector se declara indicando el tipo de dato base y la cantidad de elementos almacenados, según la sintaxis siguiente:

```
<tipo base> <nombre de la variable arreglo>[<cant. elementos>];
```

Por ejemplo, una variable arreglo para almacenar 10 caracteres se construye según se muestra a continuación:

```
char arreglo_letras[10];
```

Se puede acceder a cada una de las posiciones tanto para lectura como para escritura, y para esto es necesario indicar el subíndice:

```
char arreglo_letras[10];
char letra;

/* Escritura de la letra 'A' en la primera posición */
arreglo_letras[0] = 'A';

/* Lectura de la primera posición */
letra = arreglo_letras[0];
```

Cuando se trabaja con arreglos, es recomendable declarar un tipo de dato de usuario, antes de la declaración de la variable. En C, los tipos de dato de usuario se construyen utilizando la palabra reservada `typedef`:

```
/* Declaración de un tipo de dato arreglo */
typedef char t_arreglo[10];

/* Declaración de la variable */
t_arreglo arreglo_letras;
```

Cuando se declara un tipo de dato de usuario conviene anteponer “`t_`” al nombre del tipo (o alguna otra convención de nombres: `t_arreglo`, `arreglo_t`, `tipo_arreglo`, `tArreglo`, etc.) para favorecer la claridad del código y facilitar la distinción entre tipos y variables a lo largo del programa. Esto se aplica a la declaración de tipos en general, no sólo para arreglos.

En la práctica, los arreglos se tratan mediante estructuras de control repetitivas, lo que facilita en grado considerable el procesamiento de los datos que almacena, en especial cuando es necesario aplicar una misma operación a todos los elementos del vector. En el ejemplo siguiente se inicializa un arreglo con valores enteros generados en forma aleatoria, y se buscan los valores máximo y mínimo:

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

#define N 10
```



En el caso de C, un vector se declara indicando el tipo de dato base y la cantidad de elementos almacenados.



Es importante aclarar que en C los arreglos se indexan desde la posición 0 hasta la posición $N-1$ inclusive, siendo N el tamaño del arreglo. En lenguajes como Pascal, la indexación se hace habitualmente (aunque no necesariamente) desde 1 hasta N .



Encuentre un simulador sobre operaciones entre Vectores (suma, resta, multiplicación) en la Web de apoyo.

```

int main()
{
    int enteros[N];
    int max, min, i;

    /* Inicialización del arreglo con números aleatorios */
    srand(time(NULL));
    for (i=0; i<N; i++)
    {
        enteros[i] = rand();
        printf("enteros[%d] = %d \n", i, enteros[i]);
    }

    /* Búsqueda del máximo y el mínimo */
    max = enteros[0];
    min = enteros[0];
    for (i=1; i<N; i++)
    {
        if (enteros[i] > max)
            max = enteros[i];

        if (enteros[i] < min)
            min = enteros[i];
    }

    printf("Valor máximo = %d\n", max);
    printf("Valor mínimo = %d\n", min);

    return 0;
}

```



Encuentre un simulador sobre máximo y mínimo de un vector en la Web de apoyo.



En este ejemplo se distinguen la parte de inicialización del arreglo y la parte de procesamiento de él. La inicialización se lleva a cabo con números generados de manera aleatoria, mediante el uso de las funciones `srand()` y `rand()`. La primera permite inicializar la serie de números pseudoaleatorios a partir de una “semilla” recibida como parámetro (en este caso, se utiliza la hora del sistema como tal). Luego, cada posición del vector se inicializa con un número entero pseudoaleatorio generado con llamadas sucesivas a `rand()`.

La manipulación de arreglos en el lenguaje Pascal es análoga al caso de C. La declaración de un vector se ajusta a la sintaxis siguiente:

var <nombre arreglo>: array[1..<cant elementos>] **of** <tipo base>;

Por ejemplo:

```
var arreglo_letras: array[1..10] of char;
```

Es importante notar que, a diferencia de C, no sólo se define la cantidad de elementos sino también el dominio de los subíndices. En el ejemplo anterior este rango va de 1 a 10, pero también podría definirse entre 0 y 9, entre -5 y 4 o, incluso, utilizando caracteres. La única restricción para definir el rango es utilizar valores ordinales, como números enteros o letras.

Para declarar un tipo de dato arreglo debe hacerse uso de la palabra reservada `type` (que permite construir tipos de datos de usuario en general, no sólo arreglos):



La “semilla” recibida con parámetro solo se realiza una vez en el programa; por esto es que su llamada está fuera del ciclo `for`.

```
/* Declaración de un tipo de dato arreglo */
type t_arreglo = array[1..10] of char;

/* Declaración de la variable */
var arreglo_letras: t_arreglo;
```

El ejemplo de “máximos y mínimos”, mostrado antes para el caso de C, se presenta a continuación para el lenguaje Pascal:

```
program ejemplo;

const N = 10;

var enteros: array[1..N] of integer;
    max, min, i: integer;

begin

{ Inicialización del arreglo con números aleatorios }
randomize;
for i:=2 to N do
begin
    enteros[i] := random(100);
    writeln('enteros[' , i, '] = ', enteros[i]);
end;

{ Búsqueda del máximo y el mínimo }
max := enteros[1];
min := enteros[1];
for i:=2 to N do
begin
    if (enteros[i] > max) then
        max := enteros[i];

    if (enteros[i] < min) then
        min := enteros[i];
end;

writeln('Valor máximo = ', max);
writeln('Valor mínimo = ', min);

end.
```

Una observación final, antes de seguir adelante. Como ya se indicó al principio de este capítulo, un arreglo es una estructura (en oposición al concepto de dato simple, atómico o invisible). Así, en el ejemplo siguiente se puede afirmar que `arreglo_letras` es una variable estructurada:

```
char arreglo_letras[N];
```

Sin embargo, `arreglo_letras[3]` es una variable atómica o simple, ya que con el subíndice se hace referencia a un elemento particular dentro del arreglo (en este caso un carácter).

4.4 Arreglos multidimensionales.

El concepto de arreglo lineal puede extenderse agregando más dimensiones para dar lugar a los arreglos multidimensionales. La implementación interna y la operatoria sobre este tipo de estructuras no cambia, salvo porque para acceder a una posición, el usuario debe proporcionar tantos subíndices como sea la dimensión del arreglo. Un ejemplo ilustrativo sería el caso de una matriz de dos dimensiones:



Encuentre un simulador sobre operaciones entre matrices (suma, resta, multiplicación de matrices, multiplicación de un escalar por una matriz) en la Web de apoyo.

	0	1	2	3
0	23	8	19	67
1	1	49	3	98
2	88	30	0	7
3	99	11	2	86

Elemento [2,1]

Fig. 4-2. Matriz de dos dimensiones.

En el siguiente fragmento de código en lenguaje C se inicializa con ceros un arreglo bidimensional de enteros de MxN elementos (M y N podrían ser iguales, en cuyo caso la matriz sería cuadrada):

```
int enteros[M][N];
int i, j;
for (i=0; i<M; i++)
{
    for (j=0; j<N; j++)
    {
        enteros[i][j] = 0;
    }
}
```

En el caso de Pascal, la misma inicialización se llevaría a cabo como se muestra en el fragmento de código siguiente:

```
var enteros: array[1..M, 1..N] of integer;
i, j: integer;
for i:=1 to M do
begin
  for j:=1 to N do
  begin
    enteros[i,j] := 0;
  end;
end;
```

El anidamiento de estructuras `for` es típico cuando se procesan arreglos multidimensionales; más aún, la profundidad del anidamiento está determinada por la dimensión del arreglo (en los ejemplos mostrados aquí se anidaron dos estructuras `for` para procesar arreglos bidimensionales). Por extensión, pueden construirse arreglos con mayor número de dimensiones (2, 3, 4, ..., N dimensiones).

Antes de continuar es preciso dedicar unas líneas para desarrollar cómo se almacena un arreglo multidimensional en memoria principal, cuestión que, lejos de ser compleja, tampoco es trivial. El problema, que se ilustra en la figura siguiente, radica en cómo “acomodar” un arreglo multidimensional en memoria, siendo ésta una estructura lineal por construcción:

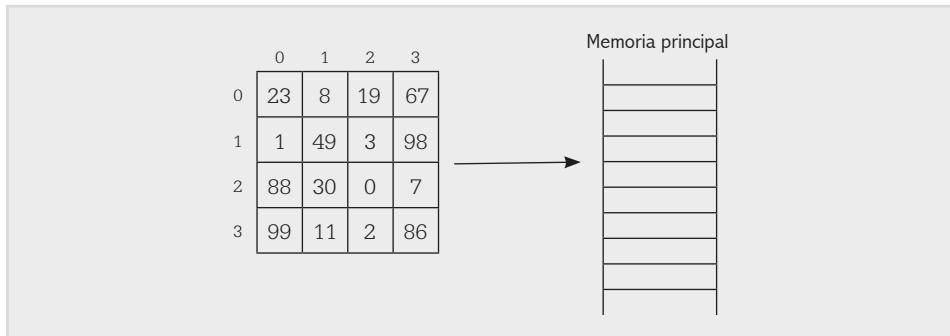


Fig. 4-3. Arreglo multidimensional n memoria de estructura lineal por construcción.

Así, es posible distinguir dos esquemas (que dependen del lenguaje de programación):

- Orden por filas (*row-major order*): Los elementos de la matriz se ubican por filas en memoria principal, antes de pasar a la fila siguiente. Éste es el esquema adoptado por lenguajes como C y Pascal, entre otros.
- Orden por columnas (*column-major order*): Los elementos de la matriz se ubican por columnas en memoria principal, antes de pasar a la columna siguiente, y son utilizados por el lenguaje Fortran (*Formula Translating System*).

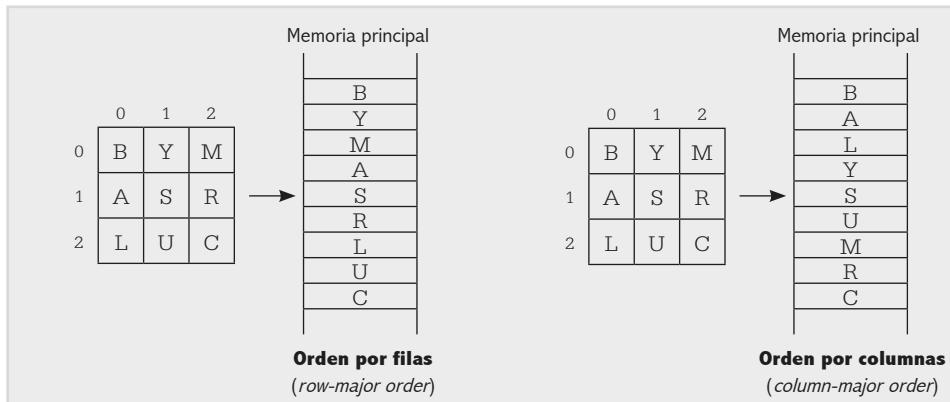


Fig. 4-4. Orden por filas (lenguajes C y Pascal) y orden por columnas (lenguaje Fortran).

En ocasiones el tipo de orden tiene implicaciones en cuestiones de desempeño (*performance*) cuando se recorre un arreglo multidimensional. Por ejemplo, en lenguajes como C o Pascal, recorrer una matriz por columnas y filas podría mostrar peor desempeño que hacerlo por filas y columnas. Estas consideraciones requieren conocimientos adicionales que escapan a un libro de fundamentos de programación y, por lo tanto, no serán profundizadas aquí.

4.5 Arreglos como parámetros de subprogramas.

En la llamada a una función o a un procedimiento se establece una lista (opcional) de parámetros que constituyen los datos de entrada a la subrutina. En el capítulo 3, el pasaje de parámetros se ilustró utilizando datos simples (números enteros o reales, caracteres, datos booleanos). Sin embargo, lenguajes como C o Pascal también soportan el pasaje de arreglos lineales y multidimensionales como parámetros.

En el caso de C, el pasaje de arreglos como parámetros debe explicarse con detenimiento y se recomienda especial atención por parte del lector. En este lenguaje, el nombre de un arreglo **es una etiqueta para la dirección de memoria del primer elemento del mismo**, como se muestra en la figura siguiente:

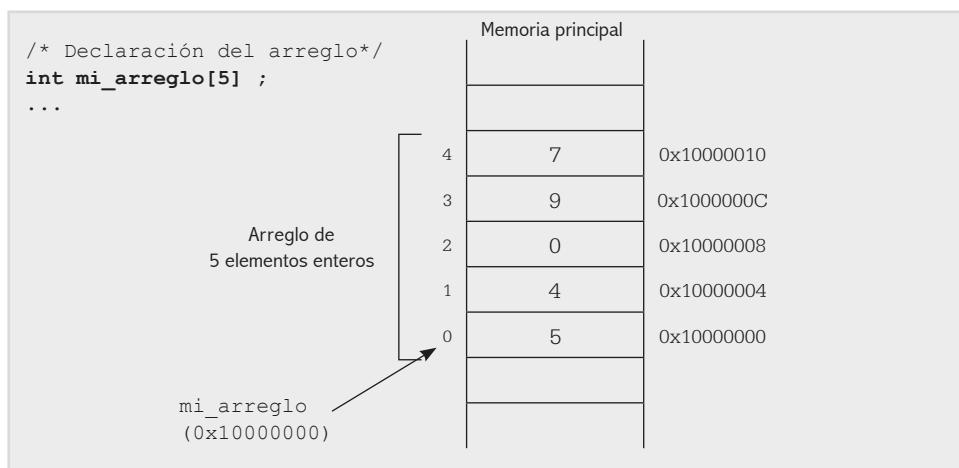


Fig. 4-5. Arreglo de 5 números enteros almacenados en memoria principal.

Indicar que el nombre del arreglo y la dirección de memoria del primer elemento son equivalentes implica que, en un pasaje de parámetros, **en verdad se pasa la dirección del primer elemento** (un puntero) y no el contenido del arreglo.

En esta figura se muestra un arreglo de 5 números enteros, llamado `mi_arreglo` y almacenado en memoria principal a partir de la dirección `0x10000000` (en hexadecimal). Suponiendo enteros de 4 bytes (32 bits), el segundo elemento del arreglo se ubica en la dirección `0x10000004` de memoria, el tercero en la dirección `0x10000008`, y así sucesivamente.

Indicar que el nombre del arreglo y la dirección de memoria del primer elemento son equivalentes implica que, en un pasaje de parámetros, **en verdad se pasa la dirección del primer elemento** (un puntero) y no el contenido del arreglo. Este pasaje de referencia permite que la función altere el contenido del arreglo recibido y estos cambios se percibirán en el ámbito de quien llamó a la subrutina. Veamos un ejemplo ilustrativo:

```

#include <stdio.h>
#define N 5
/* Prototipos */
void cargar_arreglo(int numeros[], int n);
float promediar(int numeros[], int n);

int main()
{
    int numeros[N];
    cargar_arreglo(numeros, N);
}

```

```

printf("Promedio = %.2f\n", promediar(numeros, N) );
return 0;
}

void cargar_arreglo(int numeros[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        printf("Ingrese un entero: ");
        scanf("%i", &numeros[i]);
    }
}

float promediar(int numeros[], int n)
{
    int i, sumatoria = 0;
    for (i=0; i<n; i++)
        sumatoria += numeros[i];
    return (float)sumatoria / n;
}

```

Este programa sencillo carga un arreglo de números enteros con valores ingresados por el usuario, para luego calcular el promedio de todos ellos. La carga de datos se lleva a cabo en la función `cargar_arreglo()` que, como primer parámetro, recibe el arreglo en cuestión (o la dirección de memoria del primer elemento). Esta función modifica el vector, escribiendo un nuevo dato en cada posición, y este cambio es “visible” desde la función `main()` debido a que el pasaje es por referencia (tanto `main()` como `cargar_arreglo()` acceden a las mismas posiciones en memoria principal, por estar trabajando con las mismas direcciones de memoria).

Asimismo, nótese que no se especifica el tamaño del arreglo en el listado de parámetros formales de cada función, que también es una consecuencia de lo que se acaba de explicar. El pasaje de un arreglo como parámetro es, en la práctica, el pasaje de la dirección de memoria de su primer elemento; entonces, la rutina recibe un dato simple, atómico, escalar: **Una dirección de memoria**. Por lo tanto, escribir `"int numeros []"` es apenas una “formalidad”, y el pasaje es en sí el de un dato simple (un número entero que es una dirección de memoria) y no, como podría suponerse, el contenido de todo el arreglo. Nótese, también, como otra consecuencia de lo mismo, que es necesario indicar a la función la longitud del arreglo que se está pasando por parámetro; de otra forma, la función no tendría medios para determinar cuáles son los límites del vector (sabe dónde comienza, porque está recibiendo la dirección del primer elemento; sin embargo, no sabe dónde termina).

Si el pasaje de un arreglo como parámetro de una función es, en verdad, el pasaje de una dirección de memoria, entonces el fragmento de código

```

void cargar_arreglo(int numeros[], int n)
{
    ...
}

```

puede reescribirse como se muestra a continuación:

```
void cargar_arreglo(int* numeros, int n)
{
    ...
}
```

En el segundo caso se utiliza nomenclatura de punteros; el signo "*" significa que el parámetro `numeros` es un puntero a un entero (una variable *A* es un puntero cuando almacena la dirección de memoria de un dato *B*; entonces, se indica que la variable *A* "apunta" a *B*). Ambas formas (usando "[]" o "*") son equivalentes y el segundo caso se explicará con detenimiento en el capítulo dedicado a manejo de punteros.

Por extensión puede deducirse el pasaje de un arreglo multidimensional como parámetro de una rutina. Examinemos el ejemplo siguiente, donde la función `cargar_matriz()` permite cargar una matriz bidimensional con caracteres:

```
#include <stdio.h>

#define M 5
#define N 5

/* Prototipo */
void cargar_matriz(char matriz[][][N], int filas, int columnas);

int main()
{
    char matriz_letras[M][N];

    cargar_matriz(matriz_letras, M, N);

    return 0;
}

void cargar_matriz(char matriz[][][N], int filas, int columnas)
{
    int i,j;

    for (i=0; i<filas; i++)
    {
        for (j=0; j<columnas; j++)
        {
            printf("Ingrese un carácter: ");
            scanf("%c", &matriz[i][j]);
        }
    }
}
```

La construcción "char matriz[][][N]" indica que el parámetro `matriz` es un arreglo de dos dimensiones cuyos elementos son caracteres. Tal vez el lector esperaba "char matriz[][]" ya que antes, para el caso de arreglos lineales, se indicó que no es necesario especificar la cantidad de elementos, ya que en términos estrictos sólo se pasa una dirección de memoria (y no el contenido de todo el vector). Esta cuestión puede aclararse si se interpreta la matriz bidimensional como si se tratase de un arreglo lineal, según el diagrama siguiente:

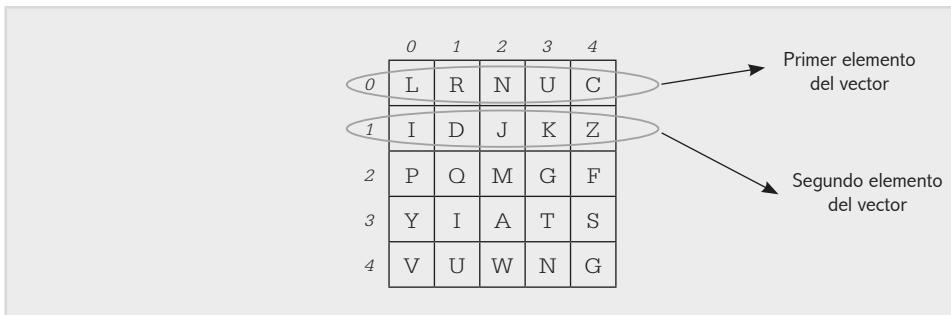


Fig. 4-6. Matriz bidimensional al "vector de sectores".

De esta manera, lo que hasta ahora había sido una matriz de $M \times N$ caracteres, ahora puede verse como un vector (arreglo lineal) donde cada elemento es otro vector de N caracteres. Cuando se pasa un arreglo como parámetro, se exige especificar el tipo de dato base (o sea, el tipo de dato de cada elemento del arreglo). Por ejemplo, en la construcción "int numeros []" no se especifica la longitud pero se especifica que el tipo de dato base es int. En el caso de la matriz de caracteres, si se la interpreta como un "vector de vectores", el tipo de dato base es un vector de 5 caracteres. Así, al escribir "char matriz [] [N]" indicamos que el parámetro es un vector de longitud desconocida ("[] [N]"), donde cada elemento tiene como tipo base otro vector de N caracteres ("char [N]").

En este punto se recomienda al lector analizar con detenimiento lo explicado hasta aquí y, luego, deducir por extensión el pasaje de parámetros de arreglos con 3 o más dimensiones.

En el caso de Pascal, los pasajes de arreglos como parámetros en procedimientos y funciones es un tanto más elegante e intuitivo para quien se está iniciando en la programación estructurada. El programa mostrado antes de cálculo de promedio se muestra a continuación, ahora para Pascal:

```

program ejemplo;
const N = 5;
type t_numeros = array[1..N] of integer;
procedure cargar_arreglo(var numeros: t_numeros; n: integer);
var i: integer;
begin
  for i:=1 to n do
    begin
      write('Ingrese un entero: ');
      readln(numeros[i]);
    end;
end;
function promediar(numeros: t_numeros; n: integer): real;
var i, sumatoria: integer;
begin
  for i:=1 to n do
    sumatoria := sumatoria + numeros[i];
  promediar := sumatoria / n;
end;

```

```

{Programa principal}
var numeros: t_numeros;
begin
    cargar_arreglo(numeros, N);
    writeln('Promedio = ', promediar(numeros, N) );
end.

```

Se observa que, en Pascal, un parámetro de tipo arreglo tiene la misma sintaxis que cualquier otro tipo de parámetro: "<nombre parámetro>: <tipo de dato>". Si es necesario pasar el arreglo por referencia, entonces se antepone la palabra reservada var, como sucede con el pasaje de parámetros por referencia para cualquier tipo de dato: "var <nombre parámetro>: <tipo de dato>". El caso de arreglos multidimensionales, por su parte, es idéntico al caso de vectores.

4.6 Cadenas de caracteres.

Un arreglo de caracteres es el recurso que ofrece C para trabajar con cadenas desde un programa. Cuando se declara:

```
char cadena[30];
```

se entiende que cadena es un arreglo lineal de caracteres de 30 bytes de tamaño (30 x 1 byte, siendo 1 byte el tamaño de cada carácter) capaz de alojar 29 caracteres y el carácter especial '\0' de "fin de cadena". Por ejemplo, si se declara "char palabra[6]", podrían almacenarse hasta 5 caracteres además de '\0':

'H'	'o'	'l'	'a'	'\0'	
0	1	2	3	4	5

Así, en el lenguaje C, una cadena es un arreglo de caracteres sin particularidad especial alguna, salvo la existencia del carácter '\0', que permite aplicar sobre estos arreglos un conjunto de funciones para manipulación de cadenas, que se encuentran en la unidad de biblioteca `string.h`. Algunas de estas funciones se listan a continuación:

Tabla 4-1 - Funciones para manipulación de cadenas que se aplican sobre los arreglos.

Función	Descripción
char* strcpy (char* s1, char* s2)	Copia s2 en s1 (incluido '\0') y retorna s1
char* strncpy (char* s1, char* s2, int n)	Copia hasta n caracteres de s2 en s1 (incluido '\0' en s1 si hay espacio) y retorna s1
char* strcat (char* s1, char* s2)	Concatena s2 a s1 y regresa s1
char* strncat (char* s1, char* s2, int n)	Concatena hasta n caracteres de s2 a s1 y regresa s1
int strcmp (char* s1, char* s2)	Compara s1 y s2, retornando 0 si son iguales, un entero negativo si s1 es menor, o un entero positivo si s1 es mayor
int strncmp (char* s1, char* s2, int n)	Ídem anterior, pero sólo se consideran los primeros n caracteres de ambas cadenas
char* strchr (char* s, int c)	Busca el carácter c en s y regresa un puntero a él si se encuentra, o NULL en caso contrario
char* strstr (char* s1, char* s2)	Busca la cadena s2 en s1 y regresa un puntero a ella si se encuentra, o NULL en caso contrario
int strlen (char* s)	Retorna la longitud (cantidad de caracteres) del arreglo s (sin contar '\0')

Para que estas funciones puedan aplicarse, es fundamental delimitar el fin de una cadena con el carácter '\0'. De otra forma, los resultados obtenidos serían indefinidos (por ejemplo, si se aplicase `strlen()` sobre una cadena no delimitada, esta función contaría caracteres hasta encontrar una ocurrencia de '\0' en alguna otra parte de memoria principal). A continuación se muestra un ejemplo simple de uso de la función `strlen()`:

```
char palabra[] = {'H','o','l','a','\0'};
printf("%d\n", strlen(palabra)); /* Devuelve 4 */
...
```

Ya que una cadena es un arreglo lineal de caracteres, es posible acceder a ellos mediante la sintaxis de arreglos:

```
palabra[0] = 'H';
palabra[1] = 'o';
...
```

El uso de cadenas de caracteres en Pascal, en cambio, se basa en la existencia del tipo predefinido `string` y de un conjunto de operadores para su manipulación. Con la siguiente declaración se construye una cadena de 255 caracteres:

```
var cadena: string;
```

Se puede especificar el tamaño máximo de la cadena de la siguiente manera:

```
var cadena: string[30];
```

A diferencia de C, las cadenas en Pascal no incluyen un carácter para indicar su fin, sino que utilizan la primera posición para almacenar la longitud (razón por la cual se indexan a partir de la posición 1). Por ejemplo, si se declara "var palabra: string[5]", podrían almacenarse hasta 5 caracteres:

4	'H'	'o'	'l'	'a'	
0	1	2	3	4	5

También pueden aplicarse operaciones como comparación, concatenación, cálculo de longitud, etc. En algunos casos, éstas se llevan a cabo con operadores predefinidos; en otros, se requiere la llamada a funciones o procedimientos. En el ejemplo siguiente se muestra el uso de alguno de ellos:

```
program ejemplo;

var cadena1, cadena2, cadena3: string[50];

begin

  {Asignación}
  cadena1 := 'Hola ';
  cadena2 := 'Mundo';

  {Concatenación}
  cadena3 := cadena1 + cadena2;
  writeln(cadena3);

  {Comparación}
  if (cadena1 < cadena2) then
    writeln('cadena1 es menor que cadena2')
```

En este ejemplo, la cadena 'palabra' es inicializada por extensión, enumerando uno a uno cada elemento (inclusive el carácter final de fin de cadena). Este tipo de inicialización equivale a: `char palabra[]: "Hola"` (en donde el compilador agrega el carácter de fin de cadena implícitamente).

A diferencia de C, las cadenas en Pascal no incluyen un carácter para indicar su fin, sino que utilizan la primera posición para almacenar la longitud.

En Pascal, tanto los caracteres simples como las cadenas se delimitan con comillas simples, a diferencia de C, donde se utilizan comillas dobles para cadenas y comillas simples para caracteres.

```

else
    if (cadena1 > cadena2) then
        writeln('cadena1 es mayor que cadena2')
    else
        writeln('Las cadenas son iguales');

{Longitud}
writeln('Longitud de cadena1 = ', length(cadena1));
writeln('Longitud de cadena2 = ', length(cadena2));
writeln('Longitud de cadena3 = ', length(cadena3));

end.

```

4.7 Enumeraciones.

El tipo enumerado permite construir datos con propiedades ordinales. La característica de este tipo de dato es que sólo puede tomar los valores de su enumeración. A continuación se presenta un ejemplo de declaración y uso de un tipo enumerado en C:

 El tipo enumerado permite construir datos con propiedades ordinales. Su característica es que sólo puede tomar los valores de su enumeración.

```

#include <stdio.h>

typedef enum {bicicleta, moto, coche,
tren, barco, avion} t_transporte;

int main()
{
    t_transporte vehiculo;
vehiculo = barco;
    switch (vehiculo)
    {
        case bicicleta:
            printf("El vehículo es bicicleta.\n");
            break;
        case moto:
            printf("El vehículo es moto.\n");
            break;
        case coche:
            printf("El vehículo es coche.\n");
            break;
        case tren:
            printf("El vehículo es tren.\n");
            break;
        case barco:
            printf("El vehículo es barco.\n");
            break;
        case avion:
            printf("El vehículo es avión.\n");
            break;
        default:
            printf("El tipo de vehículo no existe!\n");
    }
    return 0;
}

```

Pascal también brinda soporte para el tipo enumerado, según se muestra en el ejemplo siguiente:

```
program ejemplo;

type t_transporte = (bicicleta, moto, coche,
                      tren, barco, avion);

var vehiculo: t_transporte;

begin

  vehiculo := barco;

  case (vehiculo) of
    bicicleta:
      writeln('El vehículo es bicicleta.');
    moto:
      writeln('El vehículo es moto.');
    coche:
      writeln('El vehículo es coche.');
    tren:
      writeln('El vehículo es tren.');
    barco:
      writeln('El vehículo es barco.');
    avion:
      writeln('El vehículo es avión.');
    else
      writeln('El tipo de vehículo no existe!');
  end;
end.
```

4.8 Resumen.

En el capítulo II se presentaron los tipos de datos básicos, predefinidos de los lenguajes C y Pascal: Tipos enteros, de coma flotante (reales), caracteres, cadenas de caracteres y tipos booleanos. En este capítulo se presentaron los arreglos lineales (vectores) y multidimensionales (matrices), que el usuario puede definir a partir de los tipos simples disponibles.

Un arreglo lineal (vector) es un tipo de dato que se construye a partir de elementos más simples, cada uno de ellos identificado con un índice, como podría ser el caso de un arreglo de caracteres. Cada uno de los elementos que lo componen puede ser accedido, tanto para lectura como para escritura, a partir del subíndice que lo identifica (en el caso de los arreglos lineales, es suficiente utilizar un solo subíndice, debido a que se trabaja en una sola dimensión). El concepto de arreglo lineal puede extenderse agregando más dimensiones para dar lugar a los arreglos multidimensionales. La implementación interna y la operatoria sobre este tipo de estructuras no cambia, salvo por el hecho de que, para acceder a una posición, el usuario debe proporcionar tantos subíndices como sea la dimensión del arreglo (por ejemplo dos subíndices en el caso de una matriz bidimensional).

Es importante definir cuáles son las características principales de un arreglo:

- Es una estructura homogénea: Todos los datos almacenados en esa estructura son del mismo tipo (por ejemplo todos de tipo entero, o todos de tipo carácter, o todos reales, etc.).
- Es una estructura de acceso directo: Se puede acceder a los datos en forma directa, con sólo proporcionar su posición (mediante uno o más subíndices).
- Es una estructura estática: Su tamaño (número y tipo de elementos) se define en tiempo de compilación y no cambia durante la ejecución del programa.

Los arreglos, como todo dato, pueden ser pasados como parámetros de funciones o procedimientos. En este sentido, se debe prestar particular atención al pasaje de arreglos como parámetros en C. En este lenguaje, el nombre de un arreglo **es una etiqueta para la dirección de memoria del primer elemento del mismo**. Esto implica que, en un pasaje de parámetros en C, **en verdad se pasa la dirección del primer elemento** (un puntero) y no el contenido del arreglo. Este pasaje de referencia permite que la función altere el contenido del arreglo recibido y estos cambios se perciban en el ámbito de quien llamó a la subrutina. Por su parte, en Pascal el pasaje de arreglos como parámetros es análogo al de cualquier tipo de dato: Para efectuar un pasaje por referencia debe utilizarse la palabra reservada `var`.

Por su parte, las cadenas de caracteres son un caso especial dentro de los arreglos. Una cadena es un arreglo lineal de caracteres que incluye información que permite identificar el límite de la misma. En C, por ejemplo, en la posición siguiente al último carácter válido se incluye el carácter de fin de cadena ‘0’. En Pascal, en cambio, la cadena empieza en la posición 1 de arreglo, mientras que en la posición 0 se almacena la longitud.

En la última parte de este capítulo se presentó el concepto de enumeración, que permite definir conjuntos de datos con propiedades ordinales. Una variable de este tipo sólo puede almacenar uno de los valores definidos en la enumeración.

4.9 Problemas propuestos.

- 1) Hacer un algoritmo que pida 10 edades y mostrarlas en orden inverso al que se ingresaron.
- 2) Ingresar un elemento en una posición indicada en un arreglo de n elementos (números), hasta que el usuario desee salir.
- 3) Mostrar los números de un arreglo en forma ascendente.
- 4) Mostrar alumnos ordenados por mayor puntaje de promedio.
- 5) Mostrar artículos de almacén ordenados por mayor precio, y exhibir los 5 artículos de menor precio.
- 6) Hacer un programa que permita el ingreso de Nombre[X], Teléfono[X], donde X va desde 1 hasta 100. Luego mostrar la lista de los usuarios en orden inverso al que se ingresó.
- 7) En una empresa de 1 000 trabajadores se aumentarán los salarios de acuerdo con el tiempo de servicio; para este incremento se tomará en cuenta lo siguiente:

- | | |
|--|---------------|
| Tiempo de servicio: de 1 a 5 años | Aumento: 10%. |
| Tiempo de servicio: de 5 a 10 años | Aumento: 25%. |
| Tiempo de servicio: de 10 a 20 años | Aumento: 40%. |
| Tiempo de servicio: de 20 años o más | Aumento: 55%. |
| a) Hacer un algoritmo para invertir sobre sí un arreglo lineal de “P” elementos (no se puede utilizar un arreglo auxiliar). | |
| b) Obtener una lista del personal en orden creciente con respecto al sueldo modificado. | |
| c) Dada una lista de 100 personas, se pide una relación de las personas con más de 35 años de antigüedad. | |
| 8) En una encuesta, cuyas alternativas son “sí” y “no”, participaron 10000 personas. Se quiere saber cuántas de ellas votaron por la primera opción. | |

- 9) Hacer un programa que registre 30 números en un array de una dimensión, y que muestre el cuadrado de los números registrados en las posiciones pares.
- 10) Hacer un programa que registre 50 números en un arreglo de una dimensión y que muestre los números registrados en las posiciones impares de forma decreciente.
- 11) Hacer un programa que registre 50 números en un array de una dimensión y que muestre los números registrados en las posiciones impares de forma decreciente, sin tomar en cuenta el intervalo entre 25 y 30.
- 12) Hacer un programa que registre 50 números en un array de una dimensión, que muestre los múltiplos de 5.
- 13) Se tiene un array de 7 elementos y se desea insertar uno nuevo.
- 14) Se tienen 8 elementos y se desea invertir esos elementos.
- 15) Se tienen 9 marcas de jeans y se desea insertar 2 marcas nuevas en las posiciones 2 y 4.
- 16) Escribir el algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T. Sea una tabla de dimensiones M, N leídas desde el teclado.
- 17) Inicializar una matriz de dos dimensiones con un valor constante dado K.

4.10 Problemas resueltos.

1- Programe un algoritmo que, dados los datos contenidos en un vector, calcule e imprima el máximo, el mínimo, la media y la desviación tipo.

Solución en lenguaje C

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

#define MAX 100 /* Máximo de elementos del vector */
/*Programa principal */
int main ()
{
    int i, n;
    float tabla[MAX];
    float maximo, minimo, suma, cuadrados, media, sigma;
    printf("Estadística sobre un vector de datos\n");
    printf("-----\n\n");

    do
    {
        printf("¿Cuántos datos desea introducir? (2..%i): ", MAX);
        scanf("%i", &n);
    } while ( (n < 2) || (n > MAX) );

    for (i=0; i < n; i++)
    {
        printf("Introduzca el número %i: ", i+1);
        scanf("%f", &tabla[i]);
    }

    /* Cálculo de las variables estadísticas */

    maximo = tabla[0];
    minimo = tabla[0];
    suma = tabla[0];
    cuadrados = tabla[0];
    for (i=1; i < n; i++)
    {
        if (tabla[i] > maximo)
            maximo = tabla[i];
        if (tabla[i] < minimo)
            minimo = tabla[i];
        suma += tabla[i];
        cuadrados += tabla[i]*tabla[i];
    }
    media = suma/n;
    sigma = sqrt((cuadrados/n) - (media*media));
    printf("El promedio es: %.2f\n", media);
    printf("La desviación típica es: %.2f\n", sigma);
}
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```

cuadrados = tabla[0] * tabla[0];
for (i=1; i<n; i++)
{
    suma += tabla[i];
    if (tabla[i] > maximo)
        maximo = tabla[i];
    else if (tabla[i] < minimo)
        minimo = tabla[i];
    cuadrados += tabla[i] * tabla[i];
}

media = suma / n;
sigma = sqrt(cuadrados/n - media*media);

printf("Resultados\n");
printf("-----\n");
printf("Máximo = %8.2f\n", maximo);
printf("Mínimo = %8.2f\n", minimo);
printf("Media = %8.2f\n", media);
printf("Desv. = %8.2f\n", sigma);

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_01_cap04;
CONST MAX = 100; { MÁximo de elementos del vector }

VAR i, n: integer;
    tabla: ARRAY[1..MAX] OF REAL;
    maximo, minimo, suma, cuadrados, media, sigma: REAL;

BEGIN
    WRITELN('Estadística sobre un vector de datos');
    WRITELN('-----');

    REPEAT
        WRITELN('¿Cuántos datos desea introducir? (2..', MAX, '): ');
        READLN(n);
    UNTIL ( (n >= 2) AND (n <= MAX) );

    FOR i:=1 TO n DO
        BEGIN
            WRITELN('Introduzca el número ', i);
            READLN(tabla[i]);
        END;

    { Cálculo de las variables estadísticas }

    maximo := tabla[1];
    minimo := tabla[1];
    suma := tabla[1];

```

```

cuadrados := tabla[1] * tabla[1];

FOR i:=2 TO n DO
  BEGIN
    suma := suma + tabla[i];
    IF (tabla[i] > maximo) THEN
      maximo := tabla[i]
    ELSE
      IF (tabla[i] < minimo) THEN
        minimo := tabla[i];
    cuadrados := cuadrados + (tabla[i] * tabla[i]);
  END;

media := suma / n;
sigma := sqrt(cuadrados/n - media*media);

WRITELN('Resultados');
WRITELN('-----');
WRITELN('Máximo = ', maximo:6:2);
WRITELN('Mínimo = ', minimo:6:2);
WRITELN('Media = ', media:6:2);
WRITELN('Desv. = ', sigma:6:2);

END.

```

- 2- Escriba un algoritmo que indique si una clave (o un valor determinado) existe o no en un arreglo de N posiciones. Se pide que el bucle termine cuando se encuentre la clave.

Solución en lenguaje C

```

#include <stdio.h>

#define MAX 100 /* Máximo de elementos del vector */

/*Programa principal */
int main ()
{
  int i, n;
  int clave;
  int tabla[MAX];

  printf("Localización de una clave en un vector\n");
  printf("-----\n\n");

  do
  {
    printf("¿Cuántos datos desea introducir? (2..%i): ", MAX);
    scanf("%i", &n);
  } while ( (n < 1) || (n > MAX) );

  for (i=0; i < n; i++)
  {
    printf("Introduzca el número %i: ", i+1);

```

```

        scanf("%i", &tabla[i]);
    }

    /* Solicitamos la clave */
    printf("\n¿Cuál es la clave?: ");
    scanf("%i", &clave);

    /* Búsqueda */
    i = 0;
    while ((i < n) && (tabla[i] != clave))
        i++;

    if (i < n)
        printf("\nClave encontrada en la posición %i\n", i+1);
    else
        printf("\nClave no encontrada\n");

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_02_cap04;

CONST MAX = 100; { MÁximo de elementos del vector }

VAR i, n, clave: integer;
    tabla: ARRAY[1..MAX] OF INTEGER;

BEGIN
    WRITELN('Localización de una clave en un vector');
    WRITELN('-----');

    REPEAT
        WRITELN('¿Cuántos datos desea introducir? (2..', MAX, '): ');
        READLN(n);
    UNTIL ( (n >= 1) AND (n <= MAX) );

    FOR i:=1 TO n DO
        BEGIN
            WRITELN('Introduzca el número ', i);
            READLN(tabla[i]);
        END;

    WRITELN;
    WRITELN('¿Cuál es la clave?: ');
    READLN(clave);

    { Búsqueda }
    i := 1;
    WHILE ((i <= n) AND (tabla[i] <> clave)) DO
        i := i+1;

    IF (i <= n) THEN

```

```

      WRITELN('Clave encontrada en la posición ', i)
ELSE
      WRITELN('Clave no encontrada');

END.

```

3- Desarrolle un algoritmo que calcule e imprima los n primeros números de la serie de Fibonacci definida como:

```

Fibo (0) = 1
Fibo (1) = 1
Fibo (i) = Fibo(i-1) + Fibo(i-2)    para i > 1

```

Se supone $n \geq 2$

Solución en lenguaje C

```

#include <stdio.h>

#define MAX 20 /* Máximo de elementos del vector */

/*Programa principal */
int main ()
{
    int i, n;
    long fibo[MAX];
    printf("Serie de Fibonacci\n");
    printf("-----\n\n");

    do
    {
        printf("¿Cuántos términos de la serie quiere calcular? (2 .. %i): ", MAX);
        scanf("%i", &n);
    } while ( (n < 2) || (n > MAX) );

    fibo[0] = 1;
    fibo[1] = 1;
    for (i=2; i<n; i++)
        fibo[i] = fibo[i-1] + fibo[i-2];

    printf("\nLa serie de Fibonacci es:\n");
    for (i=0; i<n; i++)
        printf("fibo(%i) = %li\n", i, fibo[i]);

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_03_cap04;
CONST MAX = 20; { MÁXIMO DE ELEMENTOS DEL VECTOR }
VAR i, n: INTEGER;
    fibo: ARRAY[0..MAX-1] OF LONGINT;

```

```

BEGIN

WRITELN('Serie de Fibonacci');
WRITELN('-----');
WRITELN;

REPEAT
  WRITE('¿Cuántos términos quiere calcular? (2 .. ', MAX, '): ');
  READLN(n);
UNTIL (n >= 2) AND (n <= MAX);

fib0[0] := 1;
fib0[1] := 1;
FOR i:=2 TO n-1 DO
  fibo[i] := fibo[i-1] + fibo[i-2];

WRITELN;
WRITELN('La serie de Fibonacci es: ');
FOR i:=0 TO n-1 DO
  WRITELN('fib0 (', i, ') = ', fibo[i]);

END.

```

4- Desarrolle un algoritmo que cargue una matriz con valores ingresados por teclado.

Solución en lenguaje C

```

#include <stdio.h>

#define F 3           /* Número de filas de la matriz */
#define C 3           /* Número de columnas de la matriz */

/*Programa principal */
int main ()
{
  int i,j;
  int matriz[F][C];

  printf("Rellenado de una matriz con valores ingresados por teclado\n");
  printf("-----\n\n");

  for (i=0; i<F; i++)
    for (j=0; j<C; j++)
    {
      printf("Introduzca el elemento [%i,%i]: ", i, j);
      scanf("%i", &matriz[i][j]);
    }

  /* Una linea de separación mejora la presentación */
  printf("\n");

  for (i=0; i<F; i++)
  {

```

```

        for (j=0; j<C; j++)
            printf("%i ", matriz[i][j]);
        printf("\n");
    }

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_04_cap04;

CONST F = 3;           { Número de filas de la matriz }
                      { Número de columnas de la matriz }

VAR i,j: INTEGER;
    matriz: ARRAY[1..F, 1..C] OF INTEGER;

BEGIN
    WRITELN('Rellenado de una matriz con valores ingresados por teclado');
    WRITELN('-----');
    WRITELN;
    FOR i:=1 TO F DO
        FOR j:=1 TO C DO
            BEGIN
                WRITE('Introduzca el elemento [', i, ',', j, '] : ');
                READLN(matriz[i,j]);
            END;
    { Una línea de separación mejora la presentación }
    WRITELN;
    FOR i:=1 TO F DO
        BEGIN
            FOR j:=1 TO C DO
                WRITE(matriz[i,j], ' ');
            WRITELN;
        END;
    END.

```

5- Desarrolle un algoritmo que cargue una matriz con la suma de los índices de fila y columna.

Solución en lenguaje C

```

#include <stdio.h>

#define F 3           /* Número de filas de la matriz */
#define C 3           /* Número de columnas de la matriz */

/*Programa principal */
int main ()

```

```

{
    int i,j;
    int matriz[F][C];

    printf("Rellenado de una matriz\n");
    printf("-----\n\n");

    for (i=0; i<F; i++)
        for (j=0; j<C; j++)
            matriz[i][j] = i+j;

    for (i=0; i<F; i++)
    {
        for (j=0; j<C; j++)
            printf("%i ", matriz[i][j]);
        printf("\n");
    }

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_05_cap04;

CONST F = 3;           { Número de filas de la matriz }
                     { Número de columnas de la matriz }

VAR i,j: INTEGER;
    matriz: ARRAY[1..F, 1..C] OF INTEGER;

BEGIN
    WRITELN('Rellenado de una matriz');
    WRITELN('-----');
    WRITELN;

    FOR i:=1 TO F DO
        FOR j:=1 TO C DO
            matriz[i,j] := i+j;

    FOR i:=1 TO F DO
        BEGIN
            FOR j:=1 TO C DO
                WRITE(matriz[i,j], ' ');
            WRITELN;
        END;
    END.

```

6- Diseñe un algoritmo que sume matrices. Las matrices deben ser introducidas por el usuario.

Solución en lenguaje C

```
#include <stdio.h>

#define F 3           /* Número de filas de la matriz */
#define C 3           /* Número de columnas de la matriz */

/* Usamos un tipo de usuario */
typedef int t_matriz[F][C];

/*Programa principal */
int main ()
{
    int i,j;
    t_matriz a, b, c;

    printf("Algoritmo para sumar matrices\n");
    printf("-----\n\n");

    for (i=0; i<F; i++)
        for (j=0; j<C; j++)
    {
        printf("Introduzca el elemento [%i,%i] de la primera matriz: ", i, j);
        scanf("%i", &a[i][j]);
    }

    /* Una linea de separación mejora la presentación */
    printf ("\n");

    for (i=0; i<F; i++)
        for (j=0; j<C; j++)
    {
        printf("Introduzca el elemento [%i,%i] de la segunda matriz: ", i, j);
        scanf("%i", &b[i][j]);
    }

    /* Una linea de separación mejora la presentación */
    printf ("\n");

    /* Calculamos la suma de las dos matrices */
    for (i=0; i<F; i++)
        for (j=0; j<C; j++)
            c[i][j] = a[i][j] + b[i][j];

    for (i=0; i<F; i++)
    {
        for (j=0; j<C; j++)
            printf("%i ", c[i][j]);
        printf("\n");
    }

    return 0;
}
```

Solución en lenguaje Pascal

```

PROGRAM alg_06_cap04;

CONST FIL = 3;           { Número de filas de la matriz }
    COL = 3;           { Número de columnas de la matriz }

{ Usamos un tipo de usuario }
TYPE t_matriz = ARRAY[1..FIL, 1..COL] OF INTEGER;

VAR i,j: INTEGER;
    a, b, c: t_matriz;

BEGIN

    WRITELN('Algoritmo que suma matrices ');
    WRITELN('-----');
    WRITELN;

    FOR i:=1 TO FIL DO
        FOR j:=1 TO COL DO
            BEGIN
                WRITE('Introduzca el elemento [', i, ',', j, '] de la primera matriz: ' );
                READLN(a[i,j]);
            END;

        (* Una línea de separación mejora la presentación *)
        WRITELN;

        FOR i:=1 TO FIL DO
            FOR j:=1 TO COL DO
                BEGIN
                    WRITE('Introduzca el elemento [', i, ',', j, '] de la segunda matriz: ' );
                    READLN(b[i,j]);
                END;

        (* Una línea de separación mejora la presentación *)
        WRITELN;

        FOR i:=1 TO FIL DO
            FOR j:=1 TO COL DO
                c[i,j] := a[i,j] + b[i,j];

        FOR i:=1 TO FIL DO
            BEGIN
                FOR j:=1 TO COL DO
                    WRITE(c[i,j], ' ');
                WRITELN;
            END;

        END.

```

7- Realice un algoritmo que trasponga matrices.

Solución en lenguaje C

```
#include <stdio.h>

#define FIL 3           /* Número de filas de la matriz */
#define COL 3           /* Número de columnas de la matriz */

typedef float t_matriz[FIL][COL];

/* Programa principal */
int main()
{
    int i, j;
    t_matriz matriz, traspuesta;

    printf("Algoritmo para trasponer matrices\n");
    printf("-----\n\n");

    for (i=0; i<FIL; i++)
        for (j=0; j<COL; j++)
    {
        printf("Introduzca el elemento [%i,%i] de la matriz: ", i, j);
        scanf("%f", &matriz[i][j]);
    }

    /* Cálculo de la matriz traspuesta */
    for (i=0; i<FIL; i++)
        for (j=0; j<COL; j++)
            traspuesta[i][j] = matriz[j][i];

    printf("\nLa matriz introducida es:\n");
    for (i=0; i<FIL; i++)
    {
        for (j=0; j<COL; j++)
            printf("%.2f ", matriz[i][j]);
        printf("\n");
    }

    printf("\nLa matriz traspuesta es:\n");
    for (i=0; i<FIL; i++)
    {
        for (j=0; j<COL; j++)
            printf("%.2f ", traspuesta[i][j]);
        printf("\n");
    }

    return 0;
}
```

Solución en lenguaje Pascal

```

PROGRAM alg_07_cap04;

CONST FIL = 3;           { Número de filas de la matriz }
    COL = 3;           { Número de columnas de la matriz }

{ Usamos un tipo de usuario }
TYPE t_matriz = ARRAY[1..FIL, 1..COL] OF REAL;

VAR i,j: INTEGER;
    matriz, traspuesta: t_matriz;

BEGIN

    WRITELN('Algoritmo para trasponer matrices');
    WRITELN('-----');
    WRITELN;

    FOR i:=1 TO FIL DO
        FOR j:=1 TO COL DO
            BEGIN
                WRITE('Introduzca el elemento [', i, ',', j, '] : ');
                READLN(matriz[i,j]);
            END;

    { Cálculo de la matriz traspuesta }
    FOR i:=1 TO FIL DO
        FOR j:=1 TO COL DO
            traspuesta[i,j] := matriz[j,i];

    WRITELN;
    WRITELN('La matriz introducida es: ');
    FOR i:=1 TO FIL DO
        BEGIN
            FOR j:=1 TO COL DO
                WRITE(matriz[i,j]:4:2, ' ');
            WRITELN;
        END;

    WRITELN;
    WRITELN('La matriz traspuesta es: ');
    FOR i:=1 TO FIL DO
        BEGIN
            FOR j:=1 TO COL DO
                WRITE(traspuesta[i,j]:4:2, ' ');
            WRITELN;
        END;

    END.

```

8- Cree un algoritmo que multiplique dos matrices. Las dimensiones deben ingresarse por teclado.

Solución en lenguaje C

```
#include <stdio.h>

#define FIL 5           /* Número de filas de la matriz */
#define COL 5           /* Número de columnas de la matriz */

typedef float t_matriz[FIL][COL];

/*Programa principal */
int main ()
{
    float aux;
    int i, j, k;
    int f1, c1, f2, c2;
    t_matriz a, b, c;

    printf("Algoritmo para multiplicar matrices\n");
    printf("-----\n\n");

    do
    {
        printf("\nIntroduzca el número de filas y columnas de la primera matriz: ");
        scanf ("%i %i", &f1, &c1);
    } while ( (f1 < 1) || (f1 > FIL) || (c1 < 1) || (c1 > COL) );

    for (i=0; i<f1; i++)
        for (j=0; j<c1; j++)
    {
        printf("Introduzca el elemento [%i,%i] de la matriz: ", i, j);
        scanf("%f", &a[i][j]);
    }

    do
    {
        printf("\nIntroduzca el número de filas y columnas de la segunda matriz: ");
        scanf ("%i %i", &f2, &c2);
    } while ( (f2 < 1) || (f2 > FIL) || (c2 < 1) || (c2 > COL) || (c1 != f2) );

    for (i=0; i<f2; i++)
        for (j=0; j<c2; j++)
    {
        printf("Introduzca el elemento [%i,%i] de la matriz: ", i, j);
        scanf("%f", &b[i][j]);
    }

    /* Cálculo del producto */
    for (i=0; i<f1; i++)
        for (j=0; j<c2; j++)
    {
        aux = 0;
        for (k=0; k<c1; k++)

```

```

        aux += a[i][k] * b[k][j];
        c[i][j] = aux;
    }

/* Imprimimos la matriz producto*/
printf("\nLa matriz resultado es:\n");
for (i=0; i<f1; i++)
{
    for (j=0; j<c2; j++)
        printf("%.2f ", c[i][j]);
    printf("\n");
}
return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_08_cap04;

CONST FIL = 5;           { Número de filas de la matriz }
    COL = 5;           { Número de columnas de la matriz }

{ Usamos un tipo de usuario }
TYPE t_matriz = ARRAY[1..FIL, 1..COL] OF REAL;

VAR aux: REAL;
    i,j,k: INTEGER;
    f1,c1,f2,c2: INTEGER;
    a, b, c: t_matriz;

BEGIN
    WRITELN('Algoritmo para multiplicar matrices');
    WRITELN('-----');
    WRITELN;

    WRITELN;
    REPEAT
        WRITE('Introduzca el número de filas y de columnas de la primera matriz: ');
        READLN(f1, c1);
    UNTIL ( (f1 >= 1) AND (f1 <= FIL) AND (c1 >= 1) AND (c1 <= COL) );

    FOR i:=1 TO f1 DO
        FOR j:=1 TO c1 DO
            BEGIN
                WRITE('Introduzca el elemento [', i, ',', j, '] de la matriz: ');
                READLN(a[i,j]);
            END;

    WRITELN;
    REPEAT
        WRITE('Introduzca el número de filas y de columnas de la segunda matriz: ');
        READLN(f2, c2);

```

```

UNTIL ( (f2 >= 1) AND (f2 <= FIL) AND (c2 >= 1) AND (c2 <= COL) AND (cl = f2) );
FOR i:=1 TO f2 DO
  FOR j:=1 TO c2 DO
    BEGIN
      WRITE('Introduzca el elemento [', i, ',', j, '] de la matriz: ' );
      READLN(b[i,j]);
    END;

{ Cálculo del producto }
FOR i:=1 TO f1 DO
  FOR j:=1 TO c2 DO
    BEGIN
      aux := 0;
      FOR k:=1 TO cl DO
        aux := aux + a[i,k] * b[k,j];
      c[i,j] := aux;
    END;

WRITELN;
WRITELN('La matriz resultado es: ');
FOR i:=1 TO f1 DO
  BEGIN
    FOR j:=1 TO c2 DO
      WRITE(c[i,j]:4:2, ' ');
    WRITELN;
  END;
END.

```

9- Escriba un algoritmo que cargue una cadena de caracteres, calcule su longitud y la muestre en sentidos normal y rebatido.

Solución en lenguaje C

```

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

#define MAX_LEN 50

int main()
{
  char s[MAX_LEN];
  int l, i;

  s[0] = 'H';
  s[1] = 'o';
  s[2] = 'l';
  s[3] = 'a';
  s[4] = ' ';
  s[5] = 'M';
  s[6] = 'u';

```

```

s[7] = 'n';
s[8] = 'd';
s[9] = 'o';
s[10] = '!';
s[11] = '\0'; /* Fin de cadena */

l = strlen(s);

printf("Cadena: %s\n", s);

printf("Longitud: %d\n", l);
/* Impresión normal */
for (i=0; i<l; i++)
    printf("s[%d] = %c\n", i, s[i]);

/* Impresión rebatida */
for (i=l-1; i>=0; i--)
    printf("s[%d] = %c\n", i, s[i]);

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_09_cap04;

CONST MAX_LEN = 50;

VAR s: STRING[MAX_LEN];
    l,i: INTEGER;

BEGIN
    s := 'Hola Mundo!';
    l := LENGTH(s);

    WRITELN('Cadena: ', s);
    WRITELN('Longitud: ', l);

    { Impresión normal }
    for i:=1 TO l DO
        WRITELN('s[' , i , '] = ', s[i]);

    { Impresión rebatida }
    for i:=l DOWNTO 1 DO
        WRITELN('s[' , i , '] = ', s[i]);

END.

```

- 10- Desarrolle un algoritmo que cargue una cadena de caracteres por teclado y la copie a otra cadena auxiliar. Para el caso de C, incluya el carácter '\0' y haga uso de la función más segura fgets() para la lectura de la cadena. En el caso de Pascal, también copie el valor de longitud almacenado en la posición 0 de la cadena.

Solución en lenguaje C

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

#define MAX_LEN 50

int main()
{
    char s1[MAX_LEN];
    char s2[MAX_LEN];
    int i,l;

    printf("Ingrese una cadena (máx %d caracteres): ", MAX_LEN-1);

    /* fgets() es más segura que scanf() para leer cadenas, ya que */
    /* permite evitar desbordamientos de buffer. */
    fgets(s1, MAX_LEN, stdin);

    l = strlen(s1);

    /* Nótese que también se copia el carácter de fin de cadena */
    for (i=0; i<l+1; i++)
        s2[i] = s1[i];

    printf("s1: %s\n", s1);
    printf("s2: %s\n", s2);

    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM alg_10_cap04;

CONST MAX_LEN = 50;

VAR s1, s2: STRING[MAX_LEN];
    l, i: INTEGER;

BEGIN
    WRITE('Ingrese una cadena (máx ', MAX_LEN-1, ' caracteres): ');
    READLN(s1);

    l := LENGTH(s1);

    { Nótese que también se copia el valor de longitud, que se }
    { almacena en la posición 0 de una cadena en Pascal.      }

    for i:=0 TO l DO
        s2[i] := s1[i];

    WRITELN('s1: ', s1);
    WRITELN('s2: ', s2);

END.
```



Verifique que la terminal tiene soporte para codificación ISO8851-1 (como sucede prácticamente en todos los sistemas operativos actuales), caso contrario no se verán de forma correcta en este problema y en otros que muestren en pantalla caracteres latinos (las vocales acentuadas y la letra ñ).

- 11- Programe un algoritmo que cargue dos cadenas de caracteres por teclado y las compare. Para el caso de C, utilice la función `strcmp()` para la comparación, y haga uso de la función más segura `fgets()` para la lectura de la cadena. En el caso de Pascal, utilice los operadores de comparación.

Solución en lenguaje C

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

#define MAX_LEN 50

int main()
{
    char s1[MAX_LEN];
    char s2[MAX_LEN];

    int res;

    printf("Ingrese una cadena (máx %d caracteres): ", MAX_LEN-1);
    fgets(s1, MAX_LEN, stdin);

    printf("Ingrese otra cadena (máx %d caracteres): ", MAX_LEN-1);
    fgets(s2, MAX_LEN, stdin);

    res = strcmp(s1, s2);

    if (res == 0)
        printf("Las cadenas son iguales.\n");
    else
        if (res < 0)
            printf("La primera cadena es menor que la segunda.\n");
        else
            printf("La primera cadena es mayor que la segunda.\n");

    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM alg_11_cap04;

CONST MAX_LEN = 50;

VAR s1, s2: STRING[MAX_LEN];

BEGIN

    WRITE('Ingrese una cadena (máx ', MAX_LEN-1, ' caracteres): ');
    READLN(s1);

    WRITE('Ingrese otra cadena (máx ', MAX_LEN-1, ' caracteres): ');
    READLN(s2);

    IF (s1 = s2) THEN
        WRITELN('Las cadenas son iguales.')

```

```

ELSE
  IF (s1 < s2) THEN
    WRITELN('La primera cadena es menor que la segunda.')
  ELSE
    WRITELN('La primera cadena es mayor que la segunda.');

END.

```

- 12- Cree un algoritmo que cargue dos cadenas de caracteres y las concatene. Para el caso de C, utilice la función `strcat()`. Para el caso de Pascal, haga uso del operador de concatenación '+'.

Solución en lenguaje C

```

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

#define MAX_LEN 100

typedef char t_cadena[MAX_LEN+1];

int main()
{
  t_cadena s1 = "Hola ";
  t_cadena s2 = "Mundo!";

  printf("s1: %s\n", s1);
  printf("s2: %s\n", s2);

  strcat(s1, s2);
  printf("concatenación: %s\n", s1);

  return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_12_cap04;
CONST MAX_LEN = 50;
TYPE t_cadena = STRING[MAX_LEN];
VAR s1, s2, concatenacion: t_cadena;
BEGIN
  s1 := 'Hola ';
  s2 := 'Mundo!';

  WRITELN('s1: ', s1);
  WRITELN('s2: ', s2);

  concatenacion := s1 + s2;
  WRITELN('concatenación: ', concatenacion);

END.

```

4.11 Contenido de la página Web de apoyo.....



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- Operaciones entre vectores.
- Operaciones entre matrices.
- Máximo y mínimo de un vector.



Autoevaluación.



Video explicativo (03:45 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas.*



Presentaciones. *

5

Complejidad algorítmica Métodos de ordenamiento y búsqueda

Contenido

5.1 Introducción	126
5.2 Complejidad computacional.....	126
5.3 Métodos de búsqueda.....	129
5.4 Métodos de ordenamiento.....	132
5.5 Mezcla de arreglos.....	134
5.6 Resumen.....	136
5.7 Problemas propuestos.....	137
5.8 Problemas resueltos.....	137
5.9 Contenido de la página Web de apoyo.....	147

Objetivos

- Presentar la complejidad algorítmica.
- Dominar los métodos de búsqueda y ordenamiento.
- Explicar la mezcla de arreglos.
- Analizar los algoritmos y los métodos de más eficiente implementación.
- Comparar la eficiencia de cada método de acuerdo con las distintas circunstancias.



En la página Web de apoyo encontrarás un breve comentario del autor sobre este capítulo.



La calidad de un algoritmo puede definirse según diferentes criterios, como el **tiempo**, el **tamaño**, o cualquier otra métrica que implique economizar recursos.



Complejidad computacional:

Es el grado de esfuerzo que requiere un algoritmo para encontrar la solución del problema.

5.1 Introducción.

El procesamiento de conjuntos de datos, ya sean arreglos simples, matrices u otras estructuras más complejas, genera necesidades adicionales. En ocasiones se requiere buscar un elemento determinado entre todos los valores que conforman el conjunto de datos, como la localización de una persona en una base de datos mediante su documento de identidad. También podría ser necesario ordenar esos datos, como la generación de un listado de alumnos de una universidad ordenado en forma ascendente por apellido. Estas cuestiones pueden ser triviales en un escenario donde el conjunto de datos a procesar es pequeño. Sin embargo, si ése no es el caso, el desempeño de un programa puede verse degradado si las búsquedas o los ordenamientos se llevan a cabo de manera ineficaz. Así, para evitar posibles “cuellos de botella”, existen métodos que permiten buscar un elemento en un conjunto de datos o aplicar un ordenamiento, manteniendo niveles de *performance* aceptables. En este capítulo se estudiarán algunos de ellos: El algoritmo de búsqueda binaria y los métodos de ordenamiento por burbujeo, selección e inserción.

Para comparar los desempeños de los diferentes algoritmos que se estudiarán a continuación es preciso definir parámetros, como el grado de complejidad de cada uno de ellos. Por esta razón, antes de estudiar los métodos de búsqueda y ordenamiento, se desarrollará una introducción a los conceptos de complejidad computacional y cotas asintóticas.

5.2 Complejidad computacional.

A lo largo de este libro nos dedicamos a construir algoritmos más o menos complejos que permitieran resolver problemas determinados. En este momento comienza a ser importante estudiar la calidad de esos algoritmos, ya no en busca de la solución a un problema, sino más bien de la mejor solución posible.

La calidad de un algoritmo puede definirse según diferentes criterios, como el **tiempo** (cuánto se tarda en encontrar la solución), el **tamaño** (cuánto ocupa en memoria), o cualquier otra métrica que implique economizar recursos. En este capítulo consideraremos el tiempo como medida de rendimiento; esto es, estamos interesados en diseñar algoritmos que encuentren la solución en el menor tiempo posible.

Dado un problema determinado (por ejemplo, buscar un dato en un arreglo), diferentes algoritmos podrían dedicar distintas cantidades de esfuerzo para resolverlo. El grado de esfuerzo que requiere un algoritmo para encontrar la solución del problema es lo que se denomina complejidad computacional. Los algoritmos se clasifican de acuerdo con la complejidad en diferentes categorías que permiten compararlos (las comparaciones son útiles, sobre todo cuando se cuenta con más de un algoritmo para resolver un problema).

Cuando una persona ejecuta una tarea, para concretarla deberá dedicar un esfuerzo mayor o uno menor en función de la complejidad que ésta tenga. Por ejemplo, es probable que el esfuerzo de una persona que controla boletos en un tren sea menor que el del encargado de construir un edificio. No obstante, además, la cantidad de trabajo también depende del tamaño del problema; el esfuerzo de quien controla boletos de tren será menor cuanto menor sea el número de pasajeros y, en el mismo sentido, el esfuerzo de quien construye un edificio tendrá grados diferentes según el tamaño de la obra. De manera análoga, el tiempo que requiere un algoritmo para resolver un problema es función del tamaño X del conjunto de datos para procesar: $f(X)$.

$f(X)$ determina el esfuerzo (en nuestro caso medido en tiempo) en función del tamaño de la entrada, e interesa obtener su orden de magnitud, o sea, poder mensurar con qué velocidad crece a medida que aumenta el tamaño X del conjunto de entradas. Por ejemplo, supongamos el siguiente algoritmo en lenguaje C para buscar un elemento en un arreglo lineal:

```

int buscar(int clave, int datos[], int n)
{
    int i;
    i = 0;

    while ( (i < n) && (datos[i] != clave) )
        i = i + 1;

    return i;
}

```

La función `buscar()` recibe una clave para buscar, un conjunto de datos y el tamaño de éste (n). Supongamos que toda operación aritmético-lógica o de asignación insume una unidad de tiempo. La primera operación que efectúa el algoritmo es la inicialización de i en 0 (una unidad de tiempo). La condición lógica de la estructura `while` realiza dos comparaciones, la indexación del arreglo `datos` y un `and` (cuatro unidades de tiempo). El cuerpo del `while` ejecuta una suma y una asignación (dos unidades de tiempo). En la última línea se retorna el resultado (una unidad de tiempo). Así, el tiempo total que insume el algoritmo puede calcularse como:

$$1 + k(4 + 2) + 1 \text{ unidades de tiempo}$$

donde k es la cantidad de iteraciones que ejecuta la estructura `while`. Si el elemento buscado se hallaba en la primera posición ($k = 1$), entonces el algoritmo insume 8 unidades de tiempo. En cambio, si la clave no se encontraba en el conjunto de datos ($k = n$), entonces el tiempo de ejecución fue $6n + 2$ unidades de tiempo. Así, queda claro que el tiempo real de ejecución depende del escenario: En el mejor caso, 8 unidades de tiempo; en el peor caso, $6n + 2$ unidades de tiempo. En adelante, analizaremos los algoritmos desde un punto de vista “conservador” y sólo consideraremos el peor caso.

Tomando el peor escenario, el algoritmo de búsqueda dado antes insume $6n + 2$ unidades de tiempo. Esto significa que su complejidad tiene un orden de magnitud de n : El tiempo requerido para obtener el resultado crece de manera lineal con la cantidad n de datos de entrada. ¿Esto significa que el algoritmo es bueno o es malo? En principio no significa nada; la magnitud de su complejidad sólo es útil cuando se compara contra la complejidad de otro algoritmo que resuelve el mismo problema y, de esta forma, permite concluir sobre las virtudes de cada cual y optar por uno u otro.

Al referirnos al orden de magnitud de un algoritmo lo hacemos con expresiones como “crece tan rápido como...”, “crece por lo menos tan rápido como...”, “crece en el orden de...”. Para expresar formalmente estas sentencias, existen los símbolos O , Ω , Θ , respectivamente, y que estudiaremos a continuación.

5.2.1 Cota superior asintótica – O .

Dada $f(x)$, que expresa la complejidad de un algoritmo en función del tamaño de las entradas, la expresión $f(x) \in O(g(x))$ indica que $f(x)$ no crece más de prisa que $g(x)$ (a lo sumo lo hace tan rápido como $g(x)$ o puede crecer con más lentitud que ésta). A $g(x)$ se la denomina *cota superior asintótica*. $O(g(x))$ es un conjunto que, en términos formales, puede describirse de la siguiente manera:

$$O(g(x)) = \left\{ f(x) : \text{existen } c, x_0 > 0 \text{ tales que } \forall x \geq x_0 : 0 \leq |f(x)| \leq c|g(x)| \right\}$$

Por ejemplo, $\sin(x) \in O(1)$ (orden de complejidad constante) o $x^2 + x \in O(x^2)$ (orden cuadrático).

Gráficamente:

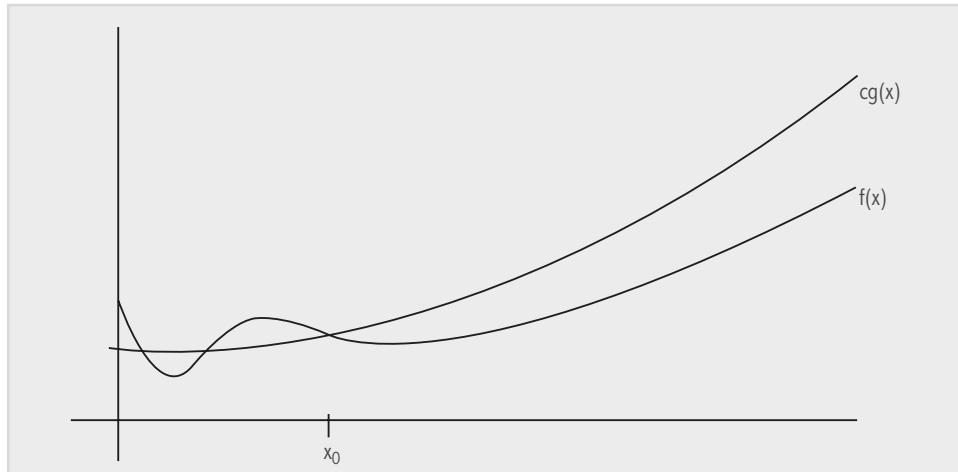


Fig. 5-1. Cota superior asintótica -O.

5.2.2 Cota inferior asintótica - Ω .

De manera análoga, $f(x) \in \Omega(g(x))$ indica que $f(x)$ crece más deprisa que $g(x)$ (a lo sumo crece a igual velocidad $g(x)$). En términos formales:

$$\Omega(g(x)) = \left\{ f(x) : \text{existen } c, x_0 \text{ constantes positivas tales que } \forall x \geq x_0: 0 \leq |f(x)| \leq c|g(x)| \right\}$$

Por ejemplo, $x^2 \in \Omega(x)$ (ya que x acota hacia inferior a x^2 cuando $x \rightarrow \infty$).

Gráficamente:

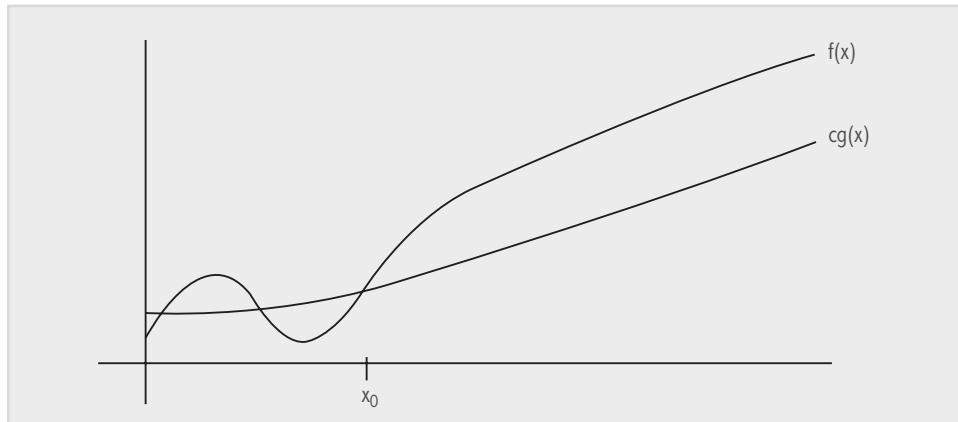


Fig. 5-2. Cota inferior asintótica - Ω .

5.2.3 Cota ajustada asintótica - Θ

Por último, $f(x) \in \Theta(g(x))$ indica que $f(x)$ y $g(x)$ crecen a la misma velocidad. Este conjunto se define como:

$$\Theta(g(x)) = \left\{ f(x) : \text{existen } c_1, c_2, x_0 \text{ constantes positivas tales que} \right. \\ \left. \forall x : x_0 \leq x : 0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x) \right\}$$

Por ejemplo, $2x^2 \in \Theta(x^2)$. Para probarlo, basta con encontrar un par de constantes c_1 y c_2 tales que $c_1 x^2 \leq x^2 \leq c_2 x^2$, como $c_1 = 0,5$ y $c_2 = 1,5$.

Gráficamente:

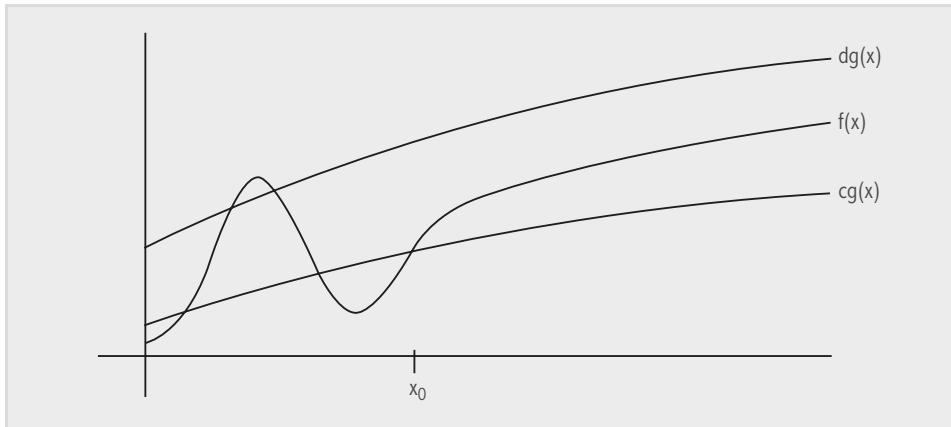


Fig. 5-3. Cota ajustada asintótica - Θ .

El empleo de cotas asintóticas para comparar algoritmos tiene sentido para entradas de tamaño considerable. En ocasiones, comparar algoritmos utilizando un número pequeño de datos de entrada puede conducir a conclusiones incorrectas sobre sus bondades.

5.3 Métodos de búsqueda.

Cuando se manipulan conjuntos de datos, la búsqueda de valores se convierte en una operación de vital importancia, cuya resolución, en ocasiones, no es trivial. Los métodos de búsqueda tienen por objeto la localización de un elemento (en un conjunto de datos) que se distingue por el valor de una **clave**. En lo sucesivo entenderemos por clave el valor, ya sea simple o no, que caracteriza en forma única al elemento. En esta sección se estudiarán dos de estos métodos, la búsqueda secuencial y la búsqueda binaria, aplicados en arreglos lineales.

5.3.1 Búsqueda secuencial.

Éste es el más simple de los algoritmos de búsqueda y su funcionamiento consiste en recorrer el arreglo, comparando cada elemento accedido contra la clave o el valor buscados. El recorrido termina al encontrarse la clave como un elemento del arreglo, o bien porque se completa la visita a todos los elementos del arreglo sin hallar la clave entre ellos. La búsqueda puede realizarse en arreglos desordenados u ordenados.

En el ejemplo siguiente se muestra una implementación (en lenguaje C) del algoritmo de búsqueda secuencial en un arreglo de enteros (nótese que si no se encuentra el elemento buscado, la función `buscar()` retorna 'n', el tamaño del arreglo):

Los métodos de búsqueda tienen por objeto la localización de un elemento (en un conjunto de datos) que se distingue por el valor de una **clave**.



Encuentre un simulador de búsqueda secuencial en la Web de apoyo.



Búsqueda secuencial.

Consiste en recorrer el arreglo, comparando cada elemento accedido contra la clave o el valor buscados. La búsqueda puede realizarse en arreglos desordenados u ordenados.

```

int buscar(int clave, int datos[], int n)
{
    int i;
    i = 0;
    while ( (i < n) && (datos[i] != clave) )
        i = i + 1;
    return i;
}

```

Si se asume que cada operación aritmético-lógica implica una unidad de tiempo, este algoritmo tiene un costo **para el peor caso** dado por la ecuación siguiente:

$$1 + n(4 + 2) + 1 = 2 + 6n \text{ unidades de tiempo}$$

La función $2 + 6n$ puede acotarse, por ejemplo, por la función lineal $10n$, para $n \geq 0,5$. Así, podemos concluir que la búsqueda secuencial tiene un orden de complejidad lineal; o sea que su costo se incrementa de manera proporcional con el tamaño del conjunto de entradas. Esto significa que si el conjunto de entradas es grande debería considerarse el uso de un algoritmo con mejor desempeño. Sin embargo, debido a la simpleza de su implementación, la búsqueda secuencial puede ser de gran utilidad cuando se aplica sobre un conjunto pequeño de datos.

Si se trata de un arreglo ordenado, la búsqueda secuencial puede optimizarse teniendo en cuenta la relación de orden entre el elemento visitado y la clave de búsqueda. En este caso, el proceso finaliza al encontrarse el elemento o bien al detectarse un elemento del arreglo mayor que el dato buscado (en el caso en que el orden sea ascendente). El siguiente es un ejemplo de implementación del algoritmo de búsqueda secuencial en un arreglo ordenado en forma ascendente:

```

int buscar(int clave, int datos[], int n)
{
    int i;
    i = 0;
    while ( (i < n) && (datos[i] != clave) )
    {
        if (clave < datos[i])
            return n;
        i++;
    }
    return i;
}

```

 **if** podría no ajustarse a los principios de la programación estructurada en ejemplos tales como el que figura a la izquierda. Como se observa en el ejemplo que tiene dos salidas: `return n` y `return`, según cumpla una condición u otra. El paradigma exige un solo punto de entrada y un solo punto de salida. Aún así, este tipo de recurso es correcto y se utiliza con frecuencia.

Búsqueda binaria.

Exige que el conjunto de datos de entrada se halle ordenado (de manera ascendente o descendente).

5.3.2 Búsqueda binaria.

Es un método más eficiente que el anterior para encontrar elementos en un arreglo y exige que el conjunto de datos de entrada se halle ordenado (de manera ascendente o descendente). El proceso comienza al comparar el elemento del arreglo ubicado en el centro del mismo contra el valor buscado. Si ambos coinciden, finaliza la búsqueda. En caso contrario, el valor buscado será mayor o menor que el central, y la búsqueda binaria continuará en el subarreglo inferior (la primera mitad

del arreglo) o en el superior (la otra mitad), para finalizar cuando se encuentre el elemento buscado o cuando el subarreglo no tenga más elementos. Veamos una implementación en lenguaje C:

```

int buscar(int clave, int datos[], int n)
{
    int inferior, superior, centro;
    inferior = 0;
    superior = n-1;

    while ( (inferior <= superior) )
    {
        centro = (inferior + superior) / 2; /* División entera */
        if (clave == datos[centro])
            return centro; /* Encontrado */
        else
            if (clave < datos[centro])
                superior = centro - 1; /* Buscar en la primera mitad */
            else
                inferior = centro + 1; /* Buscar en la segunda mitad */
    }

    /* Si no lo encuentra, retorna 'n' */
    return n;
}

```

En la primera iteración ($i = 0$) de la estructura while, el arreglo para procesar consta de n elementos. En la segunda iteración ($i = 1$), se procesa un subarreglo de $n/2$ elementos. En la siguiente ($i = 2$), el subarreglo es de tamaño $n/4 = n/2^2$. Así:

Primera iteración ($i = 0$)	\rightarrow	n elementos para procesar
Segunda iteración ($i = 1$)	\rightarrow	$n/2$ elementos para procesar
Tercera iteración ($i = 2$)	\rightarrow	$n/4 = n/2^2$ elementos para procesar
...		
i -ésima iteración	\rightarrow	$n/2^i$ elementos para procesar

En el **peor caso**, la i -ésima iteración del algoritmo es aquella en que el subarreglo para procesar contiene un solo elemento:

$$n/2^i = 1 \text{ elemento}$$

O también:

$$2^i = n$$

Aplicando \log_2 en ambos lados de la expresión:

$$\begin{aligned} \log_2(2^i) &= \log_2(n) \\ i &= \log_2(n) \end{aligned}$$

Dado que i es el número de iteraciones en el peor caso, podemos concluir que la complejidad de este algoritmo es de orden $\log_2(n)$, lo que crece más lento respecto del orden lineal de la búsqueda secuencial. Gráficamente:



Encuentre un simulador de búsqueda binaria en la Web de apoyo.

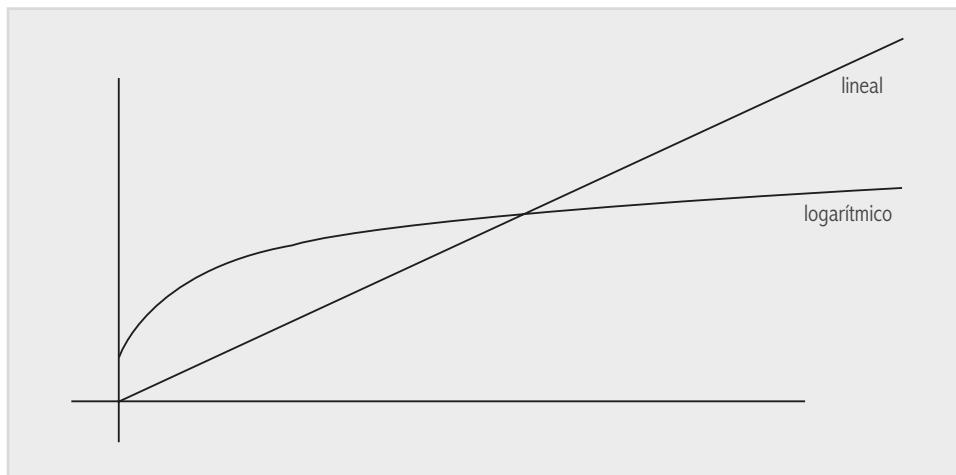


Fig. 5-4. Comparación entre búsqueda secuencial y búsqueda binaria.



Encuentre un simulador de ordenamiento por burbujeo en la Web de apoyo.



Ordenamiento por burbujeo

Este método implica comparar los elementos adyacentes en un arreglo e intercambiarlos en caso de que no se encuentren en el orden deseado.

Así, puede concluirse que la búsqueda binaria es mejor opción respecto de la búsqueda lineal, cuando el tamaño del conjunto de entrada (n) crece. Para valores pequeños de n , la búsqueda secuencial es una opción atractiva dada la simpleza de su implementación.

5.4 Métodos de ordenamiento.

A continuación se presentarán algunos de los algoritmos de ordenamiento más populares, aplicados a arreglos lineales.

5.4.1 Ordenamiento por burbujeo.

Este método implica comparar los elementos adyacentes en un arreglo e intercambiarlos en caso de que no se encuentren en el orden deseado. A continuación, se presenta la implementación en lenguaje C para orden ascendente:

```
void ordenar(int datos[], int n)
{
    int i, j, aux;
    for (i = n-1; i > 0; i--)
        for (j = 1; j <= i; j++)
            if (datos[j-1] > datos[j])
            {
                /* Intercambio */
                aux = datos[j];
                datos[j] = datos[j-1];
                datos[j-1] = aux;
            }
}
```

Se considera una “pasada” a la ejecución completa de la estructura `for` interna. Así, si el conjunto de datos se ordena de manera ascendente, en pasadas sucesivas se irán colocando los elementos mayores en las posiciones más altas del arreglo. Por tanto, el número de pasadas necesarias para que el arreglo resulte ordenado es tantas como elementos menos uno ($N-1$), ya que en la última pasada se colocarán los dos elementos más pequeños en sus posiciones finales.

La cantidad de iteraciones del bucle `for` más interno está condicionada por el valor de i . Cuando i vale $n - 1$, el bucle interno itera $n - 1$ veces. Cuando i vale $n - 2$, el bucle interno itera $n - 2$ veces. Por último, cuando i vale 1, el bucle interno itera una vez. Así, la cantidad total de iteraciones es:

$$(n - 1) + (n - 2) + \dots + 2 + 1 = n^2 - n$$

La función $n^2 - n$ está acotada por n^2 , lo que permite concluir que la complejidad de este algoritmo es de orden cuadrático.

5.4.2 Ordenamiento por selección.

Se basa en dos principios básicos:

1. Seleccionar el elemento más pequeño (o más grande) del arreglo.
2. Colocarlo en la posición más baja (o más alta) del arreglo.

Implementación:

```
void ordenar(int datos[], int n)
{
    int i, j, aux;

    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++)
            if (datos[i] > datos[j])
            {
                /* Intercambio */
                aux = datos[i];
                datos[i] = datos[j];
                datos[j] = aux;
            }
}
```



El ordenamiento por selección implica seleccionar el elemento más pequeño (o más grande) del arreglo y colocarlo en la posición más baja (o más alta) del mismo.



Encuentre un simulador de ordenamiento por selección en la Web de apoyo.

El orden de complejidad de este algoritmo también es cuadrático (su comportamiento es similar al de un ordenamiento por burbujeo).

5.4.3 Ordenamiento por inserción.

El ordenamiento por inserción es un método simple con buen desempeño para conjuntos de datos relativamente pequeños o aquellos donde existe, *a priori*, cierto grado de orden en los elementos. Su funcionamiento consiste en construir un arreglo ordenado tomando de a un elemento por vez. Por ejemplo, suponiendo el siguiente conjunto de datos:

25	7	46	33
0	1	2	3

Se toma el primer elemento, y se lo considera un arreglo ordenado con un único elemento:

25
0



El ordenamiento por inserción es un método simple con buen desempeño para conjuntos de datos pequeños o aquellos que cuentan con cierto grado de orden en sus elementos. Su funcionamiento permite construir un arreglo ordenado tomando de a un elemento por vez.

Luego se toma el segundo elemento, y se lo inserta de manera ordenada en el arreglo nuevo:

7	25
0	1



Encuentre un simulador de ordenamiento por inserción en la Web de apoyo.

Con ulterioridad se hace lo mismo con el tercer elemento:

7	25	46
0	1	2

Por último, se toma el cuarto valor y se lo inserta en el nuevo arreglo en la posición adecuada:

7	25	33	46
0	1	2	3

Lo más costoso de este algoritmo radica en insertar cada elemento en su posición correcta, ya que podría implicar desplazamientos de otros elementos en el arreglo para hacer espacio. Nótese que si el arreglo original estuviese ordenado por completo *a priori*, este método no debería implicar desplazamiento alguno (sólo comparaciones), y su complejidad será lineal. A continuación se muestra una posible implementación en lenguaje C:

```
void ordenar(int datos[], int n)
{
    int i, j;
    int temp;
    for (i = 1; i < n; i++)
    {
        if (datos[i] < datos[i-1])
        {
            temp = datos[i];
            for (j = i-1; j >= 0; j--) /* Desplazamiento */
            {
                datos[j+1] = datos[j];
                if (j == 0 || datos[j-1] <= temp)
                    break; /*encuentra la posición correcta*/
            }
            /* copia la entrada no ordenada a la posición correcta */
            datos[j] = temp;
        }
    }
}
```

Cuando se procesa cada elemento del arreglo de entrada es preciso buscar la posición correcta donde debe insertarse en el arreglo de salida. En la implementación mostrada esta búsqueda es secuencial; sin embargo, podría emplearse un algoritmo de búsqueda más eficiente (ordenamiento por inserción binaria).

El peor caso para este algoritmo se produce cuando la entrada está ordenada en sentido inverso. En ese escenario el orden de complejidad es cuadrático: $O(n^2)$.



La mezcla de arreglos consiste en generar un arreglo ordenado a partir de otros ordenados con anterioridad. Es decir que se aprovecha que ya existe un orden parcial en las entradas.

5.5 Mezcla de arreglos.

La mezcla de arreglos consiste en generar un arreglo ordenado a partir de otros ordenados con anterioridad. En lugar de concatenar los dos subarreglos en otro mayor y ordenar el conjunto, se aprovecha que ya existe un orden parcial en las entradas.

El mecanismo de mezcla consiste en comparar sendos elementos de cada uno de los arreglos de entrada y se coloca el más pequeño en el arreglo destino, tomando el elemento siguiente del arreglo correspondiente, hasta agotar todos los elementos de uno de ellos. Por último, se copiarán los elementos no procesados del otro arreglo. A continuación se presenta la implementación en lenguaje C:

```
#include <stdio.h>

#define M 4
#define N 5

int main()
{
    int arreglo1[] = {2, 3, 5, 8};
    int arreglo2[] = {1, 2, 6, 9, 10};
    int destino[M+N];
    int i, j, k;

    i = 0;
    j = 0;
    k = 0;

    /* Mezcla */
    while ( i < M && j < N )
    {
        if ( arreglo1[i] <= arreglo2[j] )
        {
            destino[k] = arreglo1[i];
            i++;
        }
        else
        {
            destino[k] = arreglo2[j];
            j++;
        }
        k++;
    }

    if (i == M)
    {
        /* 'arreglo1' no tiene más elementos,
           copiar 'arreglo2' */
        for (; j < N; j++)
        {
            destino[k] = arreglo2[j];
            k++;
        }
    }
    else if (j == N)
    {
        /* 'arreglo2' no tiene más elementos,
           copiar 'arreglo1' */
        for (; i < M; i++)
        {
            destino[k] = arreglo1[i];
            k++;
        }
    }
}
```

```

/* Visualización del arreglo resultante */
for (i=0; i<M+N; i++)
{
    printf("%d\n", destino[i]);
}

return 0;
}

```

5.6 Resumen.

Cuando se trabaja con conjuntos de datos, como vectores o matrices, aparecen dos tipos de operaciones nuevas: La búsqueda de un elemento y su ordenamiento en el conjunto. Si los conjuntos de datos con los que se opera poseen cantidades significativas de elementos, el desempeño de un programa podría verse degradado si las búsquedas o los ordenamientos se llevan a cabo de manera ineficaz. Por esta razón, en este capítulo se presentaron algunos de los algoritmos de búsqueda y ordenamiento más populares: La búsqueda secuencial y binaria, y los métodos de ordenamiento por burbujeo, selección e inserción.

Para comparar los desempeños de los diferentes algoritmos fue preciso definir parámetros, como el grado de complejidad de cada uno de ellos. Dado un problema determinado (por ejemplo, buscar un dato en un arreglo), diferentes algoritmos, para resolverlo, podrían dedicar distintas cantidades de esfuerzo. El grado de esfuerzo que requiere un algoritmo (por ejemplo, cantidad de tiempo) para encontrar la solución de un problema es lo que se denomina complejidad computacional. Así, los algoritmos se clasifican de acuerdo con la complejidad en diferentes categorías, lo que permite compararlos.

La búsqueda secuencial es el más simple de los algoritmos de búsqueda, y su funcionamiento consiste en recorrer el arreglo, comparando cada elemento accedido con la clave o el valor buscado. La búsqueda binaria es más eficiente que la secuencial, pero exige que el conjunto de datos se halle ordenado (de manera ascendente o descendente). El proceso comienza con la comparación del elemento del arreglo ubicado en el centro del mismo con el valor buscado. Si ambos coinciden, finaliza la búsqueda. En caso contrario, el valor buscado será mayor o menor estricto que el central, la búsqueda binaria continúa en el subarreglo inferior (la primera mitad del arreglo) o superior (la otra mitad), y finaliza cuando se encuentre el elemento buscado o bien cuando el subarreglo no tenga más elementos.

En cuanto a los métodos de ordenamiento, los tres algoritmos presentados (burbujeo, selección e inserción) tienen desempeños similares. Para estos métodos, si el arreglo a ordenar tiene n elementos, en el peor caso se deben realizar n^2 intercambios, y por esa razón se los clasifica como algoritmos de complejidad cuadrática: $O(n^2)$.

En algunos escenarios se cuenta con arreglos lineales previamente ordenados y se busca generar un nuevo arreglo como la unión de todos ellos, y con todos sus elementos ordenados. En lugar de concatenar los subarreglos en otro mayor y ordenar el conjunto, se aprovecha que ya existe un orden parcial en las entradas. A esta técnica se la denomina mezcla de arreglos.

5.7 Problemas propuestos.

- 1) Escribir el procedimiento de búsqueda binaria de forma recursiva.
- 2) Encontrar un elemento K en una lista de elementos x_1, x_2, \dots, x_n previamente clasificados en orden ascendente.
- 3) Igual que en el caso anterior, pero la lista de elementos está ordenada x_n, x_{n-1}, \dots, x_1 , en orden descendente.
- 4) Se ingresan 20 notas de un alumno. Publicar sus 8 notas mayores.
- 5) Ingresar "N" números; calcular el mayor.
- 6) Se ingresan 20 notas. Publicar las 5 menores.

- 7) Disponemos de un arreglo de 5 000 elementos de tipo entero. ¿Qué método de ordenación emplearía para obtener los 100 más pequeños? Realice un programa que justifique su hipótesis.
- 8) Escribir un procedimiento de ordenación por inserción binaria o dicotómica.
- 9) Escriba un programa que compare la rapidez de los distintos métodos de ordenación cuando el número total de elementos a ordenar aumenta sucesivamente con las potencias de 2. (Defina o utilice una función para medir intervalos de tiempo con las funciones GetDate y GetTime de Turbo Pascal).

5.8 Problemas resueltos.

- 1- Realice la localización de una cadena en un arreglo usando el algoritmo de búsqueda binaria.
La cadena a buscar (clave) debe ingresar como argumento por línea de comandos. El arreglo de datos debe estar ordenado con anterioridad (requisito de la búsqueda binaria).

Solución en lenguaje C

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

#define N 4

typedef char* t_cadena; /* Tipo cadena de caracteres */
typedef t_cadena t_arreglo_cadenas[N]; /* Tipo arreglo de cadenas */

/* Prototipo */
int buscar(t_cadena clave, t_arreglo_cadenas datos, int n);

/* Programa Principal */
int main(int argc, char *argv[])
{
    t_arreglo_cadenas datos = {"Ana", "Carlos", "Juan", "María"};
    int pos;

    if (argc != 2)
    {
        printf("Modo de uso: %s <cadena clave>\n", argv[0]);
        return 1;
    }

    /* Búsqueda */
    pos = buscar(argv[1], datos, N);

    if (pos != N)
    {
        printf("La cadena '%s' se encontró en la posición %d\n",
               argv[1], pos);
    }
}
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```

    }
else
{
    printf("La cadena '%s' no pudo encontrarse\n", argv[1]);
}
return 0;
}

int buscar(t_cadena clave, t_arreglo_cadenas datos, int n)
{
    int inferior, superior, centro, cmp;
    inferior = 0;
    superior = n-1;

    while ( (inferior <= superior) )
    {
        centro = (inferior + superior) / 2; /* División entera */
        cmp = strcmp(clave, datos[centro]);
        if (cmp == 0)
            return centro; /* Encontrado */
        else
            if (cmp < 0)
                superior = centro - 1; /* Buscar en la primera mitad */
            else
                inferior = centro + 1; /* Buscar en la segunda mitad */
    }

    /* Si no lo encuentra, retorna 'n' */
    return n;
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_01_cap05;
CONST N = 4;
TYPE t_cadena = STRING;
TYPE t_arreglo_cadenas = ARRAY[1..N] OF t_cadena;
FUNCTION buscar(clave: t_cadena; datos: t_arreglo_cadenas; n: INTEGER): INTEGER;
VAR inferior, superior, centro: INTEGER;
    encontrado: BOOLEAN;
BEGIN
    inferior := 1;
    superior := n;
    encontrado := FALSE;

    WHILE ( (inferior <= superior) AND (NOT encontrado) ) DO
        BEGIN
            centro := (inferior + superior) DIV 2; { División entera }

```

```

    IF (clave = datos[centro]) THEN
        encontrado := TRUE { Encontrado }
    ELSE
        IF (clave < datos[centro]) THEN
            superior := centro - 1 { Buscar en la primera mitad }
        ELSE
            inferior := centro + 1; { Buscar en la segunda mitad }
        END;

    IF (encontrado) THEN
        buscar := centro
    ELSE
        { Si no lo encuentra, retorna 0 }
        buscar := 0
    END;

VAR datos: t_arreglo_cadenas;
    pos: INTEGER;

{Programa Principal}
BEGIN

    IF (ParamCount <> 1) THEN
        BEGIN
            WRITELN('Modo de uso: ', ParamStr(0), ' <cadena clave>');
            EXIT;
        END;

    datos[1] := 'Ana';
    datos[2] := 'Carlos';
    datos[3] := 'Juan';
    datos[4] := 'María';

    { Búsqueda }
    pos := buscar(ParamStr(1), datos, N);

    IF (pos <> 0) THEN
        WRITELN('La cadena "', ParamStr(1), '" se encontró en la posición ', pos)
    ELSE
        WRITELN('La cadena "', ParamStr(1), '" no pudo encontrarse');

    END.

```

2- Haga un ordenamiento por burbujeo (en forma descendente) de un arreglo de números enteros que se genere de manera aleatoria.

Solución en lenguaje C

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10

```

```

/* Prototipo */
void ordenar(int datos[], int n);

/* Programa Principal */
int main()

{
    int datos[N];
    int i;

    /* Inicialización del generador de números aleatorios */
    srand ( time(NULL) );

    printf("Arreglo original: ");
    for (i=0; i<N; i++)
    {
        datos[i] = rand() % 100; /* Entre 0 y 99 */
        printf("%d ", datos[i]);
    }
    printf("\n");

    /* Ordenamiento */
    ordenar(datos, N);

    printf("Arreglo ordenado: ");
    for (i=0; i<N; i++)
        printf("%d ", datos[i]);
    printf("\n");

    return 0;
}

void ordenar(int datos[], int n)
{
    int i, j, aux;

    for (i = n-1; i > 0; i--)
        for (j = 1; j <= i; j++)
            if (datos[j-1] > datos[j])
            {
                /* Intercambio */
                aux = datos[j];
                datos[j] = datos[j-1];
                datos[j-1] = aux;
            }
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_02_cap05;
CONST N = 10;
type t_arreglo = ARRAY[1..10] OF INTEGER;

```

```

{ El arreglo de datos debe pasarse por referencia: VAR }
PROCEDURE ordenar(VAR datos: t_arreglo; n: INTEGER);

VAR i, j, aux: INTEGER;

BEGIN
  FOR i:=n DOWNTO 2 DO
    FOR j:=2 TO i DO
      IF (datos[j-1] > datos[j]) THEN
        BEGIN
          { Intercambio }
          aux := datos[j];
          datos[j] := datos[j-1];
          datos[j-1] := aux;
        END
      END;
    VAR datos: t_arreglo;
    i: INTEGER;

{ Programa Principal }
BEGIN

{ Inicialización del generador de números aleatorios }
RANDOMIZE;

WRITE('Arreglo original: ');
FOR i:=1 TO N DO
  BEGIN
    datos[i] := RANDOM(100); { Entre 0 y 99 }
    WRITE(datos[i], ' ');
  END;
WRITELN;

{ Ordenamiento }
ordenar(datos, N);

WRITE('Arreglo ordenado: ');
FOR i:=1 TO N DO
  WRITE(datos[i], ' ');

END.

```

3- Desarrolle un ordenamiento por selección (en forma ascendente) de un arreglo de caracteres. Luego contabilice la ocurrencia de cada carácter en una sola pasada.

Solución en lenguaje C

```

#include <stdio.h>

#define N 10

/* Prototipo */
void ordenar(char datos[], int n);
void estadisticas(char datos[], int n);

```

```

/* Programa Principal */
int main()
{
    char caracteres[N] = {'H', 'O', 'L', 'A', 'M', 'U', 'N', 'D', 'O', '!'};
    int i;

    printf("Arreglo original: ");
    for (i=0; i<N; i++)
        printf("%c ", caracteres[i]);
    printf("\n");

    /* Ordenamiento */
    ordenar(caracteres, N);

    printf("Arreglo ordenado: ");
    for (i=0; i<N; i++)
        printf("%c ", caracteres[i]);
    printf("\n\n");

    /* Estadísticas */
    estadisticas(caracteres, N);

    return 0;
}

void ordenar(char datos[], int n)
{
    int i, j;
    char aux;

    for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++)
            if (datos[i] > datos[j])
            {
                /* Intercambio */
                aux = datos[i];
                datos[i] = datos[j];
                datos[j] = aux;
            }
}

void estadisticas(char datos[], int n)
{
    int i, count;
    char car;

    car = datos[0];
    count = 0;
    for (i=0; i<N; i++)
    {
        if (datos[i] == car)
            count++;
        else
    }
}

```

```

        printf("El carácter '%c' aparece %d veces.\n", car, count);
        car = datos[i];
        count = 1;
    }
}
printf("El carácter '%c' aparece %d veces.\n", car, count);
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_03_cap05;

CONST N = 10;

{ El arreglo de datos debe pasarse por referencia: VAR }
PROCEDURE ordenar(VAR datos: STRING[N]; n: INTEGER);

VAR i, j: INTEGER;
    aux: CHAR;

BEGIN
    FOR i:=1 TO n-1 DO
        FOR j:=i+1 TO n DO
            IF (datos[i] > datos[j]) THEN
                BEGIN
                    { Intercambio }
                    aux := datos[j];
                    datos[j] := datos[i];
                    datos[i] := aux;
                END
            END;
    END;

PROCEDURE estadisticas(datos: STRING[N]; n: INTEGER);

VAR i, count: INTEGER;
    car: CHAR;

BEGIN
    car := datos[1];
    count := 0;
    FOR i:=1 TO n DO
        BEGIN
            IF (datos[i] = car) THEN
                count := count + 1
            ELSE
                BEGIN
                    WRITELN('El carácter ', car, '" aparece ', count, ' veces');
                    car := datos[i];
                    count := 1;
                END;
        END;
    END;
    WRITELN('El carácter ', car, '" aparece ', count, ' veces');
END;

```

```

VAR caracteres: STRING[N];
i: INTEGER;

{ Programa Principal }
BEGIN

{ Se usa una cadena, pero se la trata como un arreglo }
caracteres := 'HOLA MUNDO';

WRITE('Arreglo original: ');
FOR i:=1 TO N DO
  WRITE(caracteres[i], ' ');
WRITELN;

{ Ordenamiento }
ordenar(caracteres, N);

WRITE('Arreglo ordenado: ');
FOR i:=1 TO N DO
  WRITE(caracteres[i], ' ');
WRITELN;
WRITELN;

{ Estadísticas }
estadisticas(caracteres, N);

END.

```

- 4- Haga una Implementación optimizada del algoritmo de ordenamiento por burbujeo. La ejecución puede terminar en forma temprana si, en una pasada, se detecta que el arreglo está ordenado.

Solución en lenguaje C

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10

/* Prototipo */
void ordenar(int datos[], int n);

/* Programa Principal */
int main()
{
  int datos[N];
  int i;

  /* Inicialización del generador de números aleatorios */
  srand ( time(NULL) );

  printf("Arreglo original: ");
  for (i=0; i<N; i++)
  {
    datos[i] = rand() % 100; /* Entre 0 y 99 */
  }
}

```

```

        printf("%d ", datos[i]);
    }
    printf("\n");
    /* Ordenamiento */
    ordenar(datos, N);

    printf("Arreglo ordenado: ");
    for (i=0; i<N; i++)
        printf("%d ", datos[i]);
    printf("\n");

    return 0;
}

void ordenar(int datos[], int n)
{
    enum {FALSO, VERDADERO} ordenado;
    int i, j, aux, it = 0;

    ordenado = FALSO;
    for (i = n-1; i>0 && !ordenado; i--)
    {
        ordenado = VERDADERO;
        for (j = 1; j <= i; j++)
        {
            if (datos[j-1] > datos[j])
            {
                /* Intercambio */
                aux = datos[j];
                datos[j] = datos[j-1];
                datos[j-1] = aux;
                ordenado = FALSO;
            }
            it++; /* Contador de iteraciones */
        }
    }

    printf("Cantidad de iteraciones: %d\n", it);
}

```

Solución en lenguaje Pascal

```

PROGRAM alg_04_cap05;
CONST N = 10;
type t_arreglo = ARRAY[1..10] OF INTEGER;
{ El arreglo de datos debe pasarse por referencia: VAR }
PROCEDURE ordenar(VAR datos: t_arreglo; n: INTEGER);
VAR i, j, aux, it: INTEGER;
    ordenado: BOOLEAN;

```

```

BEGIN
  ordenado := FALSE;
  i := n;
  it := 0;

  WHILE ( (i >= 2) AND (NOT ordenado) ) DO
    BEGIN
      ordenado := TRUE;
      j := 2;
      WHILE (j <= i) DO
        BEGIN
          IF (datos[j-1] > datos[j]) THEN
            BEGIN
              { Intercambio }
              aux := datos[j];
              datos[j] := datos[j-1];
              datos[j-1] := aux;
              ordenado := FALSE;
            END;
          it := it + 1;
          j := j + 1;
        END;
      i := i - 1;
    END;
  WRITELN('Cantidad de iteraciones: ', it);
END;

VAR datos: t_arreglo;
  i: INTEGER;

{ Programa Principal }
BEGIN
  { Inicialización del generador de números aleatorios }
  RANDOMIZE;

  WRITE('Arreglo original: ');
  FOR i:=1 TO N DO
    BEGIN
      datos[i] := RANDOM(100); { Entre 0 y 99 }
      WRITE(datos[i], ' ');
    END;
  WRITELN;

  { Ordenamiento }
  ordenar(datos, N);

  WRITE('Arreglo ordenado: ');
  FOR i:=1 TO N DO
    WRITE(datos[i], ' ');

END.

```

5.9 Contenido de la página Web de apoyo.

El material marcado con asterisco (*) sólo está disponible para docentes.

**Mapa conceptual.****Simulación:**

- Búsqueda secuencial.
- Búsqueda binaria.
- Ordenamiento por burbujeo.
- Ordenamiento por selección.
- Ordenamiento por inserción.

**Autoevaluación.****Video explicativo** (03:02 minutos aprox.).**Código fuente de los ejercicios resueltos.****Evaluaciones propuestas.*****Presentaciones. ***

6

Estructuras y tablas

Contenido

6.1 Introducción	150
6.2 Declaración y uso de registros.....	150
6.3 Registros como parámetros de funciones.....	152
6.4 Registros jerárquicos.....	154
6.5 Uniones	155
6.6 Tablas	156
6.7 Resumen.....	159
6.8 Problemas propuestos.....	159
6.9 Problemas resueltos.....	160
6.10 Contenido de la página Web de apoyo.....	177

Objetivos

- Introducir la noción de tablas y registros y desarrollar su concepto e implementación como estructura de datos básica.
- Operar con tablas y registros.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.



El concepto de registro se implementa en lenguaje C mediante un tipo de dato conocido como **struct**. Por su parte, el soporte de registros en Pascal se brinda a partir del tipo **record**.



Encuentre un simulador sobre le uso de registros en la Web de apoyo.

6.1 Introducción.

En los capítulos anteriores se introdujo el concepto de arreglo y sus usos. Estos tipos de estructuras son en particular útiles para mantener colecciones de elementos de igual tipo, por eso se las denomina homogéneas. En ocasiones interesa almacenar en una misma estructura datos fuertemente vinculados entre sí y, además, de diferentes tipos (por ejemplo, nombre y apellido de una persona, su edad, número de cuenta bancaria y domicilio). En ese caso el tipo registro es el adecuado; se trata de una estructura heterogénea en cuanto a que es capaz de alojar datos de diferentes tipos. Cada “porción” de información en un registro se denomina campo, y se puede acceder a ella por el nombre de campo. En este sentido, el mecanismo de acceso a los datos de un registro es diferente a la indexación propia de los arreglos.

6.2 Declaración y uso de registros.

El concepto de registro se implementa en lenguaje C mediante un tipo de dato conocido como **struct**, que agrupa un conjunto de campos, que pueden ser variables de tipos intrínsecos o definidos por el usuario. Por su parte, el soporte de registros en Pascal se brinda a partir del tipo **record**. Es importante notar el alto grado de dependencia entre los campos (un registro representa una “entidad”, por ejemplo, un cliente, un alumno, un producto, etc.).

El siguiente fragmento de código define un tipo de dato estructurado llamado **t_libro**, que consta de cuatro campos, cada uno del tipo más adecuado para representar cada atributo, y que podría servir para mantener información sobre libros en una biblioteca:

```
typedef struct {
    char    titulo[20];
    char    autor[30];
    float   precio;
    int     edicion;
} t_libro;
```

Haciendo uso del tipo **t_libro** puede declararse una variable según se muestra a continuación:

```
t_libro libro;
```

Se puede acceder a cada uno de los campos que conforman una estructura, tanto para escritura como para lectura, como si se tratara de una variable convencional. Para esto debe especificarse el nombre de la variable registro, seguido del nombre del campo en cuestión, ambos separados con un punto:

```
strcpy(libro.titulo, "Rayuela");
strcpy(libro.autor, "Julio Cortázar");
libro.precio = 45;
libro.edicion = 1;
...
```

La única operación que se puede hacer con una variable tipo registro como tal es la asignación; o sea que se pueden copiar todos los campos de una variable registro a otra del mismo tipo, utilizando la sentencia de asignación, como se muestra en el ejemplo siguiente:

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

typedef struct {
    char     titulo[20];
    char     autor[30];
    float   precio;
    int      edicion;
} t_libro;

int main()
{
    t_libro libro1, libro2;

    strcpy(libro1.titulo, "Rayuela");
    strcpy(libro1.autor, "Julio Cortázar");
    libro1.precio = 45;
    libro1.edicion = 1;

    /* Asignación */
    libro2 = libro1;

    printf("Título: %s\n", libro2.titulo);
    printf("Autor: %s\n", libro2.autor);
    printf("Precio: %.2f\n", libro2.precio);
    printf("Edición: %d\n", libro2.edicion);

    return 0;
}
```

El tratamiento de registros en Pascal es análogo al mostrado para C. En primer lugar debe declararse un tipo registro, para lo que se emplea la palabra reservada `record`.

```
type
  t_libro = record
    titulo:     string[20];
    autor:     string[30];
    precio:     real;
    edicion:    integer;
  end;
```

Una vez declarado el tipo de dato pueden definirse variables, y acceder a cada uno de sus campos indicando el nombre de la variable registro y el nombre del campo en cuestión, ambos separados por un punto, como ya se mostró para el caso de C. Además, una variable registro R_1 puede asignarse a otra variable registro R_2 , siempre que R_1 y R_2 sean del mismo tipo. En el ejemplo siguiente se ilustran estos aspectos relativos a la manipulación de registros en Pascal:

```
program ejemplo;

type
  t_libro = record
    titulo:     string[20];
```

```

        autor:      string[30];
        precio:    real;
        edicion:   integer;
end;

var libro1, libro2: t_libro;

begin

    libro1.titulo  := 'Rayuela';
    libro1.autor   := 'Julio Cortázar';
    libro1.precio  := 45;
    libro1.edicion := 1;

    { Asignación }
    libro2 := libro1;

    writeln('Título: ', libro2.titulo);
    writeln('Autor: ', libro2.autor);
    writeln('Precio: ', libro2.precio:6:2);
    writeln('Edición: ', libro2.edicion);

end.

```

Es preciso aclarar que en C y en Pascal las estructuras (registros) no pueden compararse entre sí, aun si ambas variables son del mismo tipo.



Los registros también pueden pasarse como argumentos de funciones (y de procedimientos en el caso de Pascal). En el caso de C, este pasaje se realiza por copia o por referencia.

6.3 Registros como parámetros de funciones.

Como ocurre con otras variables estructuradas (arreglos lineales, matrices), los registros también pueden pasarse como argumentos de funciones (y de procedimientos, en el caso de Pascal). En el caso del lenguaje C, este pasaje se realiza por copia, aunque también puede utilizarse la dirección de memoria (puntero) donde se aloja la estructura si fuese necesario modificar alguno de sus campos dentro de la rutina (lo que también conocemos como pasaje por referencia).

El ejemplo siguiente es muy elemental, pero permite ilustrar cómo se implementan los pasajes por copia y por referencia en lenguaje C:

```

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

typedef struct {
    char    nombre[30];
    int     edad;
    long    dni;
    int     cod_carrera;
    char   nombre_carrera[20];
} t_alumno;

/* Prototipos */
void cargar_alumno(t_alumno* alumno);
void mostrar_alumno(t_alumno alumno);

int main()

```

```
{  
    t_alumno alumno;  
  
    cargar_alumno( &alumno );  
    mostrar_alumno( alumno );  
  
    return 0;  
}  
  
/* Nótese el pasaje de la referencia */  
void cargar_alumno(t_alumno* alumno)  
{  
    printf("Ingrese datos del alumno\n\n");  
  
    printf("Nombre: ");  
    fgets(alumno->nombre, 30, stdin);  
    if (alumno->nombre[strlen(alumno->nombre)-1] == '\n')  
        alumno->nombre[strlen(alumno->nombre)-1] = '\0';  
    else  
        while (getchar() != '\n'); /* Limpieza buffer de teclado */  
  
    printf("Edad: ");  
    scanf("%d", &(alumno->edad));  
    while (getchar() != '\n'); /* Limpieza buffer de teclado */  
  
    printf("DNI: ");  
    scanf("%ld", &(alumno->dni));  
    while (getchar() != '\n'); /* Limpieza buffer de teclado */  
  
    printf("Cód Carrera: ");  
    scanf("%d", &(alumno->cod_carrera));  
    while (getchar() != '\n'); /* Limpieza buffer de teclado */  
  
    printf("Nombre Carrera: ");  
    fgets(alumno->nombre_carrera, 20, stdin);  
    if (alumno->nombre_carrera[strlen(alumno->nombre_carrera)-1] == '\n')  
        alumno->nombre_carrera[strlen(alumno->nombre_carrera)-1] = '\0';  
    else  
        while (getchar() != '\n'); /* Limpieza buffer de teclado */  
}  
  
/* Nótese el pasaje por copia del registro */  
void mostrar_alumno(t_alumno alumno)  
{  
    printf("Datos del alumno\n\n");  
  
    printf("Nombre: %s\n", alumno.nombre);  
    printf("Edad: %d\n", alumno.edad);  
    printf("DNI: %ld\n", alumno.dni);  
    printf("Cód Carrera: %d\n", alumno.cod_carrera);  
    printf("Nombre Carrera: %s\n", alumno.nombre_carrera);  
}
```

Recuérdese que en C el pasaje por referencia en realidad es el pasaje de la dirección que ocupa la variable en memoria (lo que suele denominarse puntero). Para obtener la dirección de memoria de una variable se utiliza el operador '&'. Esto puede observarse en la función `main()` cuando se llama a la función `cargar_alumno()`:

```
cargar_alumno( &alumno );
...
```

Si se presta atención a la lista de parámetros formales de la función `cargar_alumno()` podemos ver lo siguiente:

```
void cargar_alumno(t_alumno* alumno)
{
    ...
}
```

Queda claro que esta función espera una dirección de memoria (lo que se expresa con el operador '*'). En particular, espera la dirección de una variable de tipo `t_alumno` (o sea, un puntero al registro). Cuando se trabaja sobre una variable que no es un registro sino un puntero a un registro, el acceso a alguno de los campos de la estructura es un poco más trabajoso. Por ejemplo, para acceder al campo `edad` del alumno, debería hacerse lo siguiente:

```
(*alumno).edad = 25;
```

Cuando escribimos `(*alumno)` estamos "desreferenciando" el puntero; o sea que a partir de su dirección en memoria estamos accediendo en forma indirecta a su contenido (esto aplica a punteros a cualquier tipo de dato, no sólo punteros a registros). Para ser más claros: En el ejemplo anterior `alumno` es un puntero (referencia) a una estructura, mientras que `(*alumno)` es la estructura en sí. Dado que es muy frecuente el uso de punteros a estructuras, en C encontramos una forma equivalente y más clara de hacer lo mismo:

```
alumno->edad = 25; /* Equivale a (*alumno).edad = 25 */
```

Todo lo referente a manipulación de punteros será tratado más adelante, pero en este punto es preciso aclarar un poco las cosas para seguir avanzando.

El pasaje de registros como parámetros en el caso de Pascal es análogo a C. La diferencia más significativa está en el pasaje por referencia que, en Pascal, se hace utilizando la palabra reservada `var`:

```
procedure cargar_alumno(var alumno: t_alumno);
```

En los ejemplos incluidos al final de este capítulo podrán encontrarse modelos de pasajes de registros como parámetros, tanto para C como para Pascal.

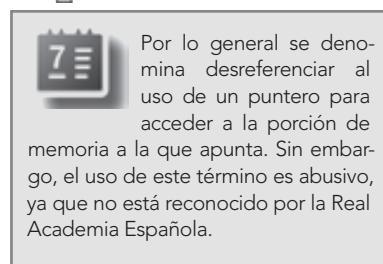
6.4 Registros jerárquicos.

Los campos de los registros pueden ser de cualquier tipo definido por el usuario, incluso también registros. Un registro con uno o más campos de tipo registro se denomina registro jerárquico. Por ejemplo, un registro que contenga datos de una persona, incluida su fecha de nacimiento, puede declararse de la siguiente forma:

```
typedef struct {
    int dia;
    int mes;
    int anio;
```



Por lo general se denombra desreferenciar al uso de un puntero para acceder a la porción de memoria a la que apunta. Sin embargo, el uso de este término es abusivo, ya que no está reconocido por la Real Academia Española.



```

} t_fecha;

typedef struct {
    char nombre[40];
    t_fecha fecha_nac;
    ...
} t_persona;

```

El acceso a los campos de un registro anidado en otro puede hacerse como se muestra a continuación:

```

t_persona persona;
persona.fecha_nac.dia = 11;
persona.fecha_nac.mes = 10;
persona.fecha_nac.anio = 1979;
...

```

Pascal también brinda soporte para registros jerárquicos en forma idéntica al lenguaje C.

6.5 Uniones.

En C existe un tipo de dato estructurado llamado unión que tiene un aspecto similar al de una estructura, salvo porque todos sus miembros (campos) comparten la misma porción de memoria (a diferencia de una estructura convencional, donde los campos se alojan en posiciones independientes). En una estructura de tipo unión, existe un espacio de memoria compartido por las variables que forman parte de ella, y el compilador le asignará a la unión tanta memoria como requiera el más grande de los campos. La declaración es similar al caso de las estructuras, utilizando la palabra reservada union:

```

typedef union {
    tipo componente_1;
    tipo componente_2;
    tipo componente_3;
    ...
    tipo componente_n;
} nombre_tipo_union;

```

Para dejar en claro el funcionamiento y el propósito de las uniones, veamos un ejemplo simple. Suponiendo una computadora con procesador de 32 bits, su memoria principal estará físicamente organizada como un arreglo, donde cada posición es un bloque de memoria de 32 bits (4 bytes), según se muestra en la figura siguiente:

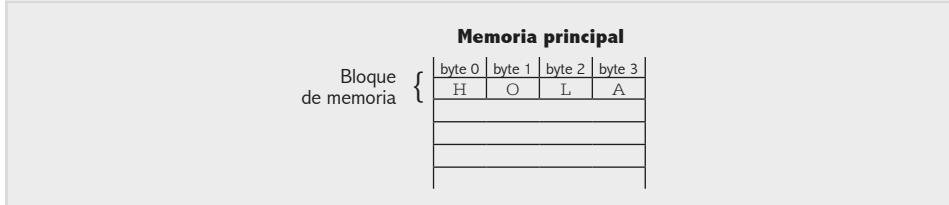
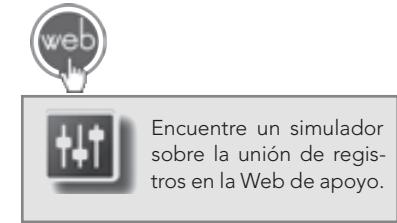


Fig. 6-1. Funcionamiento y propósito de las uniones.

O sea que un programa, durante su ejecución, podría almacenar en un bloque de memoria datos tales como cuatro caracteres (de 1 byte cada uno), un entero (4 bytes), 2 caracteres y un entero corto, entre otras posibilidades. Así, cada bloque de memoria podría modelarse como una unión de 4 bytes de longitud:

```
#include <stdio.h>

typedef union {
    int dato;      /* Asumimos enteros de 4 bytes = 32 bits */
    struct {
        char b0;
        char b1;
        char b2;
        char b3;
    } bytes;
} t_bloque_mem; /* Bloque de memoria de 4 bytes = 32 bits */

int main()
{
    t_bloque_mem linea_memoria;

    linea_memoria.bytes.b0 = 'H';
    linea_memoria.bytes.b1 = 'O';
    linea_memoria.bytes.b2 = 'L';
    linea_memoria.bytes.b3 = 'A';

    printf("La linea de memoria contiene: [%x]\n",
linea_memoria.dato);

    return 0;
}
```

El tipo `t_bloque_mem` es una unión que contiene dos campos: Un entero (`dato`) y una estructura anidada (`bytes`). Por tratarse de una unión, tanto el campo entero como el campo estructura se alojan en la misma porción de memoria (4 bytes en este caso). De esta forma, cuando se hace referencia a cada campo `b1`, `b2`, `b3` o `b4` de la estructura anidada, también se accede a cada byte del entero `dato`. En este ejemplo el uso de la unión facilita la tarea de tener que acceder a un byte en particular dentro del bloque de memoria, lo que podría ser una tarea frecuente en el funcionamiento interno de un sistema operativo.

Aunque de uso menos frecuente, Pascal también ofrece un tipo de dato cuyo comportamiento es similar al de las uniones de C: El registro variante.

6.6 Tablas.

El uso combinado de arreglos lineales y registros da lugar a un recurso de gran utilidad denominado tabla. Dado que un registro permite almacenar información relacionada con una entidad, un **registro de registros** permite manipular colecciones de estas entidades (por ejemplo, un listado de datos de personas, productos de un almacén, historias clínicas, etc.). El concepto de tabla es muy simple, y consiste en implementar un arreglo lineal donde cada componente del vector es un registro. Gráficamente puede interpretarse de la manera siguiente:

Tabla 6-1 - Representación de una tabla en la que se implementa un arreglo lineal donde cada componente del vector es un registro.

	<i>Nombre y apellido</i>	<i>Edad</i>	<i>DNI</i>	<i>COD.</i>	<i>Nombre carrera</i>
0	Juan Maidana	23	31 896 005	9	Lic. Sistemas
1					
2					

En esta figura se representa una tabla, en la que cada posición del arreglo es un registro con los campos siguientes: nombre y apellido, edad, DNI, código y nombre de carrera. Como se definió en el capítulo 4, un arreglo es una estructura homogénea en cuanto a que todos sus elementos son del mismo tipo. Esta afirmación todavía es válida aun en el caso de las tablas, ya que cada posición del arreglo en cuestión contiene un registro de tipo X (por ejemplo, t_alumno) y todas las posiciones del arreglo contienen registros del mismo tipo X.

A continuación se ilustra el uso de tablas mediante un programa simple que permite cargar y visualizar un listado de datos de alumnos (las funciones cargar_alumno() y mostrar_alumno() son las implementadas antes):

```
#define MAX_ALUMNOS 100

typedef struct {
    char    nombre[30];
    int     edad;
    long    dni;
    int     cod_carrera;
    char    nombre_carrera[20];
} t_alumno;

/* Tabla de alumnos */
typedef t_alumno t_tabla_alumnos[MAX_ALUMNOS];

/* Prototipos */
void cargar_alumno(t_alumno* alumno);
void mostrar_alumno(t_alumno alumno);

int main()
{
    t_tabla_alumnos tabla_alumnos;
    int i;

    /* Carga de la tabla */
    for (i=0; i<MAX_ALUMNOS; i++)
    {
        cargar_alumno( &tabla_alumnos[i] );
    }

    /* Visualización de la tabla */
    for (i=0; i<MAX_ALUMNOS; i++)
    {
        mostrar_alumno( tabla_alumnos[i] );
    }
}

return 0;
}
```

El acceso a un campo en un registro de la tabla se lleva a cabo indexando, en primer lugar, el arreglo, y luego referenciando el campo en cuestión:

```
tabla_alumnos[i].edad = 23;
```

El mismo ejemplo de manipulación de un listado de alumnos se presenta a continuación, ahora implementado el lenguaje Pascal:

```
program ejemplo_tabla;
const MAX_ALUMNOS = 100;
type t_alumno = record
    nombre: string[30];
    edad: integer;
    dni: longint;
    cod_carrera: integer;
    nombre_carrera: string[20];
end;

{ Tabla de alumnos }
t_tabla_alumnos = array[1..MAX_ALUMNOS] of t_alumno;

{ Nótese el pasaje de la referencia }
procedure cargar_alumno(var alumno: t_alumno);
begin
    ...
end;

{ Nótese el pasaje por copia del registro }
procedure mostrar_alumno(alumno: t_alumno);
begin
    ...
end;

var tabla_alumnos: t_tabla_alumnos;
    i: integer;

{ Programa Principal }
begin
    { Carga de la tabla }
    for i:=1 to MAX_ALUMNOS do
        begin
            cargar_alumno( tabla_alumnos[i] );
        end;

    { Visualización de la tabla }
    for i:=1 to MAX_ALUMNOS do
        begin
            mostrar_alumno( tabla_alumnos[i] );
        end;
end.
```

A esta altura del libro, el lector debería estar familiarizado con las estructuras homogéneas (arreglos lineales y matrices) y heterogéneas (registros y tablas) más importantes de todo lenguaje de programación estructurada. Estos tipos de datos constituyen las bases (el “abc”) que permiten la construcción de programas más o menos complejos, con la ayuda (imprescindible) de la creatividad y las buenas prácticas del programador.

6.7 Resumen.

En el capítulo 4 se presentaron las estructuras homogéneas vector y matriz. Estos tipos de datos son útiles para mantener colecciones de elementos de igual tipo (razón por la cual se las denomina homogéneas). En ocasiones interesa almacenar, en una misma estructura, datos de diferentes tipos (por ejemplo, nombre y apellido de una persona, su edad, número de cuenta bancaria y domicilio). En lenguajes como C y Pascal esto puede implementarse mediante un tipo estructurado heterogéneo llamado registro o estructura.

Cada “porción” de información en un registro se denomina campo, y se puede acceder a ella por el nombre de campo. En este sentido, el mecanismo de acceso a los datos de un registro es diferente a la indexación propia de los arreglos.

El concepto de registro se implementa en lenguaje C mediante un tipo de dato conocido como **struct**, que agrupa un conjunto de campos. Por su parte, el soporte de registros en Pascal se brinda a partir del tipo **record**. Es importante notar el alto grado de cohesión entre los campos (un registro representa una “entidad”, por ejemplo, un cliente, un alumno, un producto, etc.).

Los registros también pueden pasarse como argumentos de funciones (y de procedimientos, en el caso de Pascal). En C este pasaje se realiza por copia, aunque también puede utilizarse la dirección de memoria (puntero) donde se aloja la estructura si fuese necesario modificar alguno de sus campos dentro de la rutina (pasaje por referencia). El caso de Pascal es análogo a C; la diferencia más significativa está en el pasaje por referencia, que en Pascal se hace utilizando la palabra reservada **var**.

El uso combinado de arreglos lineales y registros da lugar a un tipo de estructura denominado tabla. Dado que un registro permite almacenar información relacionada con una entidad, un **arreglo de registros** permite manipular colecciones de estas entidades (por ejemplo, un listado de datos de personas, productos de un almacén, historias clínicas, etc.). El concepto de tabla es muy simple, y consiste en implementar un arreglo lineal donde cada componente del vector es un registro.

6.8 Problemas propuestos.

- 1) Las notas de 30 alumnos están almacenadas en un arreglo de registros. Los registros contienen apellido y nota, suponiendo apellidos no repetidos. Se pide ordenarlos alfabéticamente en forma ascendente.
- 2) Del caso anterior (se tiene un arreglo de registros de 30 alumnos), se pide ordenarlos alfabéticamente en forma descendente. Se pretende obtener las 5 notas más bajas, considerando que las notas pueden estar repetidas pero los apellidos no.
- 3) Ahora, calcular el promedio de la mejor nota y la peor.
- 4) Se dispone de una relación de personas de sexo masculino y femenino. Mostrar mediante un programa todas las combinaciones posibles de parejas que pueden contraer matrimonio, con el requisito de que ambas sean de distinto sexo y la diferencia de edad no supere los diez años.
- 5) Crear una agenda de registros de personas donde se almacenen su nombre, domicilio y número de teléfono, y en la que se puedan introducir altas, bajas, consultas, etc., mediante el menú de opciones correspondiente.
- 6) Se dispone de una relación de patentes de autos correspondientes a las distintas provincias de un país, dado que tienen multas de tráfico pendientes. Se pide generar un aplicativo que ordene y muestre las distintas patentes ordenadas primero por provincia, por importe de la multa, y posteriormente indique la provincia que figura a la cabeza de deudores.

6.9 Problemas resueltos.

- 1- Escriba un programa que reciba, por línea de comandos, los datos correspondientes a un pixel (coordenadas y componentes de color RGB). Con la función `parse_input()` “parsee” las cadenas de entrada. Con la función `invert_color()` invierta los componentes RGB. Observe las dos posibilidades de retornar datos desde una función: Por su nombre o pasando una referencia del parámetro.

Solución en lenguaje C

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

/* Estructura para almacenar las coordenadas */
/* y el color RGB de un pixel. */
typedef struct {
    float      x, y;    /* Coordenadas */
    unsigned char r,g,b; /* Componentes Red, Green, Blue */
} t_pixel;

/* Prototipos */
t_pixel parse_input(char*[]);
void invert_color(t_pixel*);

int main(int argc, char* argv[])
{
    t_pixel pixel;
    if (argc != 6)
    {
        printf("Modo de uso: %s <x> <y> <r> <g> <b>\n", argv[0]);
        return 1;
    }

    pixel = parse_input(argv);

    printf("\tCoord pixel: ( %.2f , %.2f )\n", pixel.x, pixel.y);
    printf("\tRGB:          ( %d , %d , %d )\n", pixel.r, pixel.g, pixel.b);

    /* El color RGB es invertido */
    invert_color(&pixel);

    printf("\tRGB invert:   ( %d , %d , %d )\n", pixel.r, pixel.g, pixel.b);

    return 0;
}

/* El nuevo registro es retornado a través del nombre de */
/* la función. */
t_pixel parse_input(char* argv[])
{
    t_pixel p;
    p.x = atof(argv[1]); /* atof() convierte una cadena de */
    p.y = atof(argv[2]); /* caracteres en un dato 'float' */
    p.r = atoi(argv[3]); /* atoi() convierte una cadena de */
    p.g = atoi(argv[4]); /* caracteres en un dato 'int' */
}
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```

p.b = atoi(argv[5]);
return p;
}

/* Obsérvese el pasaje de una referencia (puntero) para */
/* modificar el parámetro dentro de la función. */
void invert_color(t_pixel* pix)
{
    pix->r = 255 - pix->r; /* Equivale a: (*pix).r = ... */
    pix->g = 255 - pix->g; /* Equivale a: (*pix).g = ... */
    pix->b = 255 - pix->b; /* Equivale a: (*pix).b = ... */
}

```

Solución en lenguaje Pascal

```

PROGRAM cap06_ej01;

{ Estructura para almacenar las coordenadas }
{ y el color RGB de un pixel. }

TYPE t_pixel = RECORD
    x, y: REAL;      { Coordenadas }
    r,g,b: BYTE;     { Componentes Red, Green, Blue }
END;

{ El nuevo registro es retornado a través del nombre de }
{ la función. }

FUNCTION parse_input: t_pixel;

VAR p: t_pixel;
    errorflag : INTEGER;

BEGIN
    { La función val() intenta convertir la cadena recibida }
    { como primer parámetro en un valor numérico. }

    VAL(PARAMSTR(1), p.x, errorflag);
    VAL(PARAMSTR(2), p.y, errorflag);

    VAL(PARAMSTR(3), p.r, errorflag);
    VAL(PARAMSTR(4), p.g, errorflag);
    VAL(PARAMSTR(5), p.b, errorflag);

    parse_input := p;
END;

{ Obsérvese el pasaje de una referencia (puntero) para }
{ modificar el parámetro dentro de la función. }

PROCEDURE invert_color(VAR pix: t_pixel);

BEGIN
    pix.r := 255 - pix.r;
    pix.g := 255 - pix.g;
    pix.b := 255 - pix.b;
END;

```

```

VAR pixel: t_pixel;
{ Programa Principal }
BEGIN
  IF (PARAMCOUNT <> 5) THEN
    BEGIN
      WRITELN('Modo de uso: ', PARAMSTR(0), ' <x> <y> <r> <g> <b>');
      EXIT;
    END;
  pixel := parse_input();
  WRITELN(' Coord pixel: ( ', pixel.x:6:2, ' , ', pixel.y:6:2, ' )');
  WRITELN(' RGB:           ( ', pixel.r, ' , ', pixel.g, ' , ', pixel.b, ' )');
  { El color RGB es invertido }
  invert_color(pixel);
  WRITELN(' RGB invert:  ( ', pixel.r, ' , ', pixel.g, ' , ', pixel.b, ' )');
END.

```

2- Utilice registros “jerárquicos” para cargar datos relativos a un alumno y calcular el promedio de sus notas. Utilice la función fgets() y la limpieza del buffer de teclado.

Solución en lenguaje C

```

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

#define MAX_NOMBRE      51
#define MAX_MATERIAS   30

typedef struct {
    int dia, mes, anio;
} t_fecha;

typedef struct {
    long cod_materia;
    float nota;
    t_fecha fecha_aprob;
} t_materia;

typedef struct {
    long padron;
    char nombre[MAX_NOMBRE];
    t_materia materias[MAX_MATERIAS];
} t_alumno;

float promedio(t_materia[]);

int main()
{
    t_alumno alumno;
    int i;

```

```

printf("Ingrese los datos del alumno:\n\n");
printf("Padrón: ");
scanf("%ld", &alumno.padron);
while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Nombre y apellido (máx %d caracteres): ", MAX_NOMBRE-1);
fgets(alumno.nombre, MAX_NOMBRE, stdin);
if (alumno.nombre[strlen(alumno.nombre)-1] == '\n')
    alumno.nombre[strlen(alumno.nombre)-1] = '\0';
else
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

for (i=0; i<MAX_MATERIAS; i++)
{
    printf("\tCód materia: ");
    scanf("%ld", &alumno.materias[i].cod_materia);
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("\tNota: ");
    scanf("%f", &alumno.materias[i].nota);
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("\tFecha aprobación (DD MM AAAA): ");
    scanf("%d %d %d", &alumno.materias[i].fecha_aprob.dia,
          &alumno.materias[i].fecha_aprob.mes,
          &alumno.materias[i].fecha_aprob.anio );
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("\n");
}

printf("PROMEDIO DE NOTAS: %.2f\n\n", promedio(alumno.materias));

return 0;
}

float promedio(t_materia materias[])
{
    int i;
    float suma = 0.0;

    for (i=0; i<MAX_MATERIAS; i++)
        suma += materias[i].nota;

    return (suma / MAX_MATERIAS);
}

```

Solución en lenguaje Pascal

```

PROGRAM cap06_ej02;

CONST MAX_NOMBRE    = 50;
      MAX_MATERIAS = 30;

TYPE t_fecha = RECORD

```

```

        dia, mes, anio: INTEGER;
      END;

TYPE t_materia = RECORD
  cod_materia: LONGINT;
  nota: REAL;
  fecha_aprob: t_fecha;
END;

TYPE t_materias = ARRAY[1..MAX_MATERIAS] OF t_materia;

TYPE t_alumno = RECORD
  padron: LONGINT;
  nombre: STRING[MAX_NOMBRE];
  materias: t_materias;
END;

FUNCTION promedio(materias: t_materias): REAL;
VAR i: INTEGER;
  suma: REAL;

BEGIN
  suma := 0.0;
  FOR i:=1 TO MAX_MATERIAS DO
    suma := suma + materias[i].nota;
  promedio := suma / MAX_MATERIAS;
END;

VAR alumno: t_alumno;
  i: INTEGER;

{Programa Principal}
BEGIN

  WRITELN('Ingrese los datos del alumno:');
  WRITELN;

  WRITE('Padrón: ');
  READLN(alumno.padron);
  WRITE('Nombre y apellido (máx ', MAX_NOMBRE, ' caracteres): ');
  READLN(alumno.nombre);
  FOR i:=1 TO MAX_MATERIAS DO
    BEGIN
      WRITE(' Cód materia: ');
      READLN(alumno.materias[i].cod_materia);
      WRITE(' Nota: ');
      READLN(alumno.materias[i].nota);
      WRITE(' Fecha aprobación (DD MM AAAA): ');
      READLN(alumno.materias[i].fecha_aprob.dia,
             alumno.materias[i].fecha_aprob.mes,
             alumno.materias[i].fecha_aprob.anio);
    END;
  WRITELN;
END;

```

```

WRITELN('PROMEDIO DE NOTAS: ', promedio(alumno.materias):4:2);
END.

```

- 3- En una empresa de turismo se desea saber cuál es el país más visitado por sus clientes. Para ello se cuenta con una tabla de destinos y la cantidad de veces que fueron visitados. Se implementan funciones para la carga de datos y la búsqueda de máximo (que es secuencial). Cree el algoritmo necesario para tal fin. Utilice y observe la lectura de cadenas de caracteres mediante la función fgets(), más segura que scanf(). También haga la limpieza del buffer de teclado.

Solución en lenguaje C

```

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

#define MAX_DESTINOS 100

/* Tipo registro para almacenar un destino */
typedef struct {
    char pais[30];
    char medio[30];
    int cant_veces;
} t_destino;

/* Lista de destinos */
typedef t_destino t_vector[MAX_DESTINOS];

/* Prototipos */
void cargar_datos(t_vector, int);
t_destino buscar_mas_visitado(t_vector, int);

/* Programa Principal */
int main()
{
    t_vector lista_destinos;
    int cant_destinos;
    t_destino destino;

    printf("EMPRESA DE TURISMO\n");
    printf("~~~~~\n\n");

    printf("Ingrese la cantidad de destinos (países) a cargar: ");
    scanf("%d", &cant_destinos);
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    cargar_datos(lista_destinos, cant_destinos);
    destino = buscar_mas_visitado(lista_destinos, cant_destinos);

    printf("País más visitado es: %s\n", destino.pais);
    printf("Medio de transporte es: %s\n", destino.medio);
    printf("Cantidad de visitas es: %d\n", destino.cant_veces);

    return 0;
}

void cargar_datos(t_vector lista_destinos, int n)

```

```

{
    int i;

    for (i=0; i<n; i++)
    {
        printf("Ingrese el país: ");
        fgets(lista_destinos[i].pais, 30, stdin);
        if (lista_destinos[i].pais[strlen(lista_destinos[i].pais)-1] == '\n')
            lista_destinos[i].pais[strlen(lista_destinos[i].pais)-1] = '\0';
        else
            while (getchar() != '\n'); /* Limpieza buffer de teclado */

        printf("Ingrese el medio de transporte: ");
        fgets(lista_destinos[i].medio, 30, stdin);
        if (lista_destinos[i].medio[strlen(lista_destinos[i].medio)-1] == '\n')
            lista_destinos[i].medio[strlen(lista_destinos[i].medio)-1] = '\0';
        else
            while (getchar() != '\n'); /* Limpieza buffer de teclado */

        printf("Ingrese la cantidad de veces que eligieron este destino: ");
        scanf("%d", &(lista_destinos[i].cant_veces));
        while (getchar() != '\n'); /* Limpieza buffer de teclado */
    }
}

/* El registro cargado es retornado por el nombre de la función */
t_destino buscar_mas_visitado(t_vector lista_destinos, int n)
{
    int i, pos = 0;
    int mayor = lista_destinos[0].cant_veces;

    for (i=1; i<n; i++)
        if (lista_destinos[i].cant_veces > mayor)
    {
        mayor = lista_destinos[i].cant_veces;
        pos = i;
    }

    return lista_destinos[pos];
}

```

Solución en lenguaje Pascal

```

PROGRAM cap06_ej03;

CONST MAX_DESTINOS = 100;

{ Tipo registro para almacenar un destino }
TYPE t_destino = RECORD
    pais: STRING[30];
    medio: STRING[30];
    cant_veces: INTEGER;
END;

```

```
{ Lista de destinos }
TYPE t_vector = ARRAY[1..MAX_DESTINOS] OF t_destino;

PROCEDURE cargar_datos(VAR lista_destinos: t_vector; n: INTEGER);
VAR i: INTEGER;

BEGIN
  FOR i:=1 TO n DO
    BEGIN
      WRITE('Ingrese el país: ');
      READLN(lista_destinos[i].pais);
      WRITE('Ingrese el medio de transporte: ');
      READLN(lista_destinos[i].medio);
      WRITE('Ingrese la cantidad de veces que eligieron este destino: ');
      READLN(lista_destinos[i].cant_veces);
    END;
  END;

{ El registro cargado es retorna por el nombre de la función }
FUNCTION buscar_mas_visitado(lista_destinos: t_vector; n: INTEGER): t_destino;
VAR i, pos, mayor: INTEGER;

BEGIN
  pos := 1;
  mayor := lista_destinos[1].cant_veces;

  FOR i:=2 TO n DO
    IF (lista_destinos[i].cant_veces > mayor) THEN
      BEGIN
        mayor := lista_destinos[i].cant_veces;
        pos := i;
      END;
  END;

  buscar_mas_visitado := lista_destinos[pos];
END;

VAR lista_destinos: t_vector;
  cant_destinos: INTEGER;
  destino: t_destino;

{ Programa Principal }
BEGIN
  WRITELN('EMPRESA DE TURISMO');
  WRITELN('~~~~~');
  WRITELN;

  WRITE('Ingrese la cantidad de destinos (países) a cargar: ');
  READLN(cant_destinos);

  cargar_datos(lista_destinos, cant_destinos);
  destino := buscar_mas_visitado(lista_destinos, cant_destinos);

  WRITELN('País más visitado es: ', destino.pais);
  WRITELN('Medio de transporte es: ', destino.medio);
```

```

WRITELN('Cantidad de visitas es: ', destino.cant_veces);
END.

```

- 4- Cree un programa que permita cargar una planilla con datos de alumnos, donde cada uno de ellos disponga de 3 notas y una nota final (promedio de las otras) se inicialice la planilla (todos los registros con padrón igual a 0, lo que significa que ese registro está disponible), luego se lleve a cabo la carga de datos y, por último, se liste la planilla completa. Use la función fgets () y limpíe el buffer de teclado.

Solución en lenguaje C

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define MAX_ALUMNOS 100

/* Tipo de dato para almacenar tres notas por cada estudiante */
typedef float t_notas[3];

/* Tipo registro para almacenar un alumno */
typedef struct {
    long      padron;
    char      nombre[50];
    t_notas   notas;
    float     nota_final;
} t_alumno;

/* Lista de estudiantes */
typedef t_alumno t_planilla[MAX_ALUMNOS];

/* Dato enumerativo para retornar el "status" de una función */
typedef enum {OK, ERROR} t_status;

/* Prototipos */
void inicializar_planilla(t_planilla);
t_status ingresar_nuevo_alumno(t_planilla);
void listar_planilla(t_planilla);
float calcular_nota_final(t_alumno);

/* Programa Principal */
int main()
{
    t_planilla planilla_alumnos;
    t_status status;
    char opcion;

    /* Nótese que, dado que 'planilla_alumnos' es un arreglo, */
    /* se pasa un puntero a la primera posición (referencia) */
    /* en lugar de hacerse una copia de todos los elementos. */
    inicializar_planilla(planilla_alumnos);

    do {

```

```
status = ingresar_nuevo_alumno(planilla_alumnos);

if (status == ERROR)
{
    printf("Se ha producido un error en la carga de datos.\n");
    exit(1);
}

printf("Ingresar otro alumno? (s/n): ");
do {
    scanf("%c", &opcion);
} while ( toupper(opcion) != 'S' && toupper(opcion) != 'N' );

} while (toupper(opcion) == 'S');

printf("\n");

listar_planilla(planilla_alumnos);

return 0;
}

/* Inicialización del listado de estudiantes asignando un */
/* número de padrón vacío */
void inicializar_planilla(t_planilla listado)
{
    int i;
    for(i=0; i<MAX_ALUMNOS; i++)
        listado[i].padron = 0;
}

/* Alta de un nuevo estudiante en el listado. Retorna ERROR */
/* si no hubiese espacio suficiente. */
t_status ingresar_nuevo_alumno(t_planilla listado)
{
    int i = 0;

    /* Busca un lugar libre */
    while ((i < MAX_ALUMNOS) && (listado[i].padron != 0))
        i++;

    if (i == MAX_ALUMNOS)
    {
        printf("No hay espacio para un nuevo estudiante.");
        return ERROR;
    }

    printf("Ingrese los datos del nuevo estudiante:\n\n");

    printf("Padrón: ");
    scanf("%ld",&(listado[i].padron));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("Nombre: ");
    fgets(listado[i].nombre, 50, stdin);
```

```

if (listado[i].nombre[strlen(listado[i].nombre)-1] == '\n')
    listado[i].nombre[strlen(listado[i].nombre)-1] = '\0';
else
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Nota 1: ");
scanf("%f",&(listado[i].notas[0]));
while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Nota 2: ");
scanf("%f",&(listado[i].notas[1]));
while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Nota 3: ");
scanf("%f",&(listado[i].notas[2]));
while (getchar() != '\n'); /* Limpieza buffer de teclado */

listado[i].nota_final = calcular_nota_final(listado[i]);

return OK;
}

/* Muestra en pantalla la lista de datos de los estudiantes */
/* registrados. */
void listar_planilla(t_planilla listado)
{
    int i;
    for (i=0; i<MAX_ALUMNOS && listado[i].padron!=0; i++)
    {
        printf("Padrón:      %ld\n", listado[i].padron);
        printf("Nombre:     %s\n", listado[i].nombre);
        printf("Nota 1:      %.2f\n", listado[i].notas[0]);
        printf("Nota 2:      %.2f\n", listado[i].notas[1]);
        printf("Nota 3:      %.2f\n", listado[i].notas[2]);
        printf("Nota Final:  %.2f\n", listado[i].nota_final);
        printf("\n");
    }
}

/* Calcula la nota final como promedio de las tres notas del */
/* estudiante. */
float calcular_nota_final(t_alumno estudiante)
{
    float final = ( estudiante.notas[0] +
                    estudiante.notas[1] +
                    estudiante.notas[2] ) / 3 ;

    return final;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap06_ej04;
CONST MAX_ALUMNOS = 3;

```

```
{ Tipo de dato para almacenar tres notas por cada estudiante }
TYPE t_notas = ARRAY[1..3] OF REAL;

{ Tipo registro para almacenar un alumno }
TYPE t_alumno = RECORD
    padron:      LONGINT;
    nombre:      STRING[50];
    notas:       t_notas;
    nota_final:  REAL;
END;

{ Lista de estudiantes }
TYPE t_planilla = ARRAY[1..MAX_ALUMNOS] OF t_alumno;

{ Dato enumerativo para retornar el "status" de una función }
TYPE t_status = (OK, ERROR);

{ Inicialización del listado de estudiantes asignando un }
{ número de padrón vacío }
PROCEDURE inicializar_planilla(VAR listado: t_planilla);

VAR i: INTEGER;

BEGIN
    FOR i:=1 TO MAX_ALUMNOS DO
        listado[i].padron := 0;
END;

{ Calcula la nota final como promedio de las tres notas del }
{ estudiante. }
FUNCTION calcular_nota_final(estudiante: t_alumno): REAL;

VAR final: REAL;

BEGIN
    final := ( estudiante.notas[1] +
               estudiante.notas[2] +
               estudiante.notas[3] ) / 3 ;

    calcular_nota_final := final;
END;

{ Alta de un nuevo estudiante en el listado. Retorna ERROR }
{ si no hubiese espacio suficiente. }
FUNCTION ingresar_nuevo_alumno(VAR listado: t_planilla): t_status;

VAR i: INTEGER;

BEGIN
    i := 0;

    { Busca un lugar libre }
    WHILE ((i <= MAX_ALUMNOS) AND (listado[i].padron <> 0)) DO
        inc(i); {Equivale a i:=i+1}

    IF (i = MAX_ALUMNOS) THEN
        BEGIN
```

```
WRITELN('No hay espacio para un nuevo estudiante.');
ingresar_nuevo_alumno := ERROR;

END
ELSE
BEGIN

    WRITELN('Ingrese los datos del nuevo estudiante:');
    WRITELN;

    WRITE('Padrón: ');
    READLN(listado[i].padron);

    WRITE('Nombre: ');
    READLN(listado[i].nombre);

    WRITE('Nota 1: ');
    READLN(listado[i].notas[1]);

    WRITE('Nota 2: ');
    READLN(listado[i].notas[2]);

    WRITE('Nota 3: ');
    READLN(listado[i].notas[3]);

    listado[i].nota_final := calcular_nota_final(listado[i]);

END;
END;

{ Muestra en pantalla la lista de datos de los estudiantes }
{ registrados. }
PROCEDURE listar_planilla(listado: t_planilla);

VAR i: INTEGER;

BEGIN
    i:=1;

    WHILE (listado[i].padron <> 0) DO
        BEGIN
            WRITELN('Padrón: ', listado[i].padron);
            WRITELN('Nombre: ', listado[i].nombre);
            WRITELN('Nota 1: ', listado[i].notas[1]:4:2);
            WRITELN('Nota 2: ', listado[i].notas[2]:4:2);
            WRITELN('Nota 3: ', listado[i].notas[3]:4:2);
            WRITELN('Nota Final: ', listado[i].nota_final:4:2);
            WRITELN;

            inc(i); {Equivale a i:=i+1}
        END;
    END;

    VAR planilla_alumnos: t_planilla;
        status: t_status;
```

```

opcion: char;

{ Programa Principal }
BEGIN

{ El pasaje del arreglo es por referencia }
inicializar_planilla(planilla_alumnos);

REPEAT

    status := ingresar_nuevo_alumno(planilla_alumnos);
    IF (status = ERROR) THEN
        BEGIN
            WRITELN('Se ha producido un error en la carga de datos.');
            EXIT;
        END;

    WRITE('Ingresar otro alumno? (s/n): ');
    REPEAT
        READLN(opcion);
        UNTIL ( (UPCASE(opcion)='S') OR (UPCASE(opcion)='N') );
    UNTIL (UPCASE(opcion) = 'N');

    WRITELN;

    listar_planilla(planilla_alumnos);

END.

```

5- Inicialice un tablero para el juego de la "batalla naval". El tablero es una matriz bidimensional, donde cada casillero es un registro. La inicialización debería hacerse en forma aleatoria (tarea para el lector).

Solución en lenguaje C

```

#include <stdio.h>

#define COLS      20
#define FILAS     10

typedef enum {ACORAZADO, CRUCERO, PORTAAVIONES} t_tipo;
typedef enum {FALSE, TRUE} t_bool;

typedef struct {
    t_bool      ocupado;
    t_tipo      tipo;
    t_bool      hundido;
} t_casillero;

typedef t_casillero t_tablero[FILAS][COLS];

/* Prototipos */
void inicializar_tablero(t_tablero);
void mostrar_tablero(t_tablero);

int main()
{

```

```
t_tablero tablero;
inicializar_tablero(tablero);
mostrar_tablero(tablero);

return 0;
}

void inicializar_tablero(t_tablero tablero)
{
    int i,j;

    for (i=0; i<FILAS; i++)
        for (j=0; j<COLS; j++)
            tablero[i][j].ocupado = FALSE;

    /* 1 acorazado */
    tablero[4][18].ocupado = TRUE;
    tablero[4][18].tipo = ACORAZADO;
    tablero[4][18].hundido = FALSE;
    tablero[5][18].ocupado = TRUE;
    tablero[5][18].tipo = ACORAZADO;
    tablero[5][18].hundido = FALSE;
    tablero[6][18].ocupado = TRUE;
    tablero[6][18].tipo = ACORAZADO;
    tablero[6][18].hundido = FALSE;

    /* 2 cruceros */
    tablero[9][2].ocupado = TRUE;
    tablero[9][2].tipo = CRUCERO;
    tablero[9][2].hundido = FALSE;
    tablero[9][3].ocupado = TRUE;
    tablero[9][3].tipo = CRUCERO;
    tablero[9][3].hundido = FALSE;

    tablero[3][9].ocupado = TRUE;
    tablero[3][9].tipo = CRUCERO;
    tablero[3][9].hundido = FALSE;
    tablero[4][9].ocupado = TRUE;
    tablero[4][9].tipo = CRUCERO;
    tablero[4][9].hundido = FALSE;

    /* 3 portaaviones */
    tablero[1][1].ocupado = TRUE;
    tablero[1][1].tipo = PORTAAVIONES;
    tablero[1][1].hundido = FALSE;

    tablero[6][12].ocupado = TRUE;
    tablero[6][12].tipo = PORTAAVIONES;
    tablero[6][12].hundido = FALSE;

    tablero[8][19].ocupado = TRUE;
    tablero[8][19].tipo = PORTAAVIONES;
    tablero[8][19].hundido = FALSE;
}
```

```

void mostrar_tablero(t_tablero tablero)
{
    int i,j;

    printf("\t");
    for (i=0; i<COLS+2; i++)
        printf("-");
    printf("\n");

    for (i=0; i<FILAS; i++)
    {
        printf("\t|");
        for (j=0; j<COLS; j++)
        {
            if (!tablero[i][j].ocupado)
                printf(" ");
            else
                switch (tablero[i][j].tipo)
                {
                    case ACORAZADO:      printf("#");
                    break;
                    case CRUCERO:        printf("X");
                    break;
                    case PORTAAVIONES:   printf("@");
                    break;
                }
        }
        printf("|\n");
    }

    printf("\t");
    for (i=0; i<COLS+2; i++)
        printf("-");
    printf("\n");
}

```

Solución en lenguaje Pascal

```

PROGRAM cap06_ej05;

CONST COLS = 20;
CONST FILAS = 10;

TYPE t_tipo = (ACORAZADO, CRUCERO, PORTAAVIONES);

TYPE t_casillero = RECORD
    ocupado: BOOLEAN;
    tipo: t_tipo;
    hundido: BOOLEAN;
END;

TYPE t_tablero = ARRAY[1..FILAS, 1..COLS] OF t_casillero;
PROCEDURE inicializar_tablero(VAR tablero: t_tablero);

VAR i,j: INTEGER;

```

```

BEGIN
  FOR i:=1 TO FILAS DO
    FOR j:=1 TO COLS DO
      tablero[i,j].ocupado := FALSE;

    { 1 acorazado }
    tablero[4,18].ocupado := TRUE;
    tablero[4,18].tipo    := ACORAZADO;
    tablero[4,18].hundido := FALSE;
    tablero[5,18].ocupado := TRUE;
    tablero[5,18].tipo    := ACORAZADO;
    tablero[5,18].hundido := FALSE;
    tablero[6,18].ocupado := TRUE;
    tablero[6,18].tipo    := ACORAZADO;
    tablero[6,18].hundido := FALSE;

    { 2 cruceros }
    tablero[9,2].ocupado := TRUE;
    tablero[9,2].tipo    := CRUCERO;
    tablero[9,2].hundido := FALSE;
    tablero[9,3].ocupado := TRUE;
    tablero[9,3].tipo    := CRUCERO;
    tablero[9,3].hundido := FALSE;

    tablero[3,9].ocupado := TRUE;
    tablero[3,9].tipo    := CRUCERO;
    tablero[3,9].hundido := FALSE;
    tablero[4,9].ocupado := TRUE;
    tablero[4,9].tipo    := CRUCERO;
    tablero[4,9].hundido := FALSE;

    { 3 portaaviones }
    tablero[1,1].ocupado := TRUE;
    tablero[1,1].tipo    := PORTAAVIONES;
    tablero[1,1].hundido := FALSE;

    tablero[6,12].ocupado := TRUE;
    tablero[6,12].tipo    := PORTAAVIONES;
    tablero[6,12].hundido := FALSE;

    tablero[8,19].ocupado := TRUE;
    tablero[8,19].tipo    := PORTAAVIONES;
    tablero[8,19].hundido := FALSE;

  END;
  PROCEDURE mostrar_tablero(tablero: t_tablero);
  VAR i,j: INTEGER;
  BEGIN
    WRITE(' ');
    FOR i:=1 TO COLS+2 DO
      WRITE('-');
    WRITELN;

```

```

FOR i:=1 TO FILAS DO
BEGIN
    WRITE(' | ');
    FOR j:=1 TO COLS DO
    BEGIN
        IF (NOT tablero[i,j].ocupado) THEN
            WRITE(' ')
        ELSE
            CASE (tablero[i,j].tipo) OF
                ACORAZADO:      WRITE('#');
                CRUCERO:        WRITE('X');
                PORTAAVIONES:  WRITE('@');
            END;
        END;
    END;
    WRITELN(' | ');
END;

WRITE('   ');
FOR i:=1 TO COLS+2 DO
    WRITE('-');
WRITELN;
END;

VAR tablero: t_tablero;
{ Programa Principal }
BEGIN
    inicializar_tablero(tablero);
    mostrar_tablero(tablero);
END.

```

6.10 Contenido de la página Web de apoyo.



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- Uso de registros.
- Unión de registros.



Evaluaciones propuestas.*



Presentaciones. *



Autoevaluación.



Video explicativo (02:20 minutos aprox.).



Código fuente de los ejercicios resueltos.

7

Archivos

Contenido

7.1 Introducción	180
7.2 Tratamiento de archivos en lenguaje C	181
7.3 Tratamiento de archivos en lenguaje Pascal.....	187
7.4 Archivos de acceso directo.....	189
7.5 Operaciones entre archivos	191
7.6 Resumen.....	193
7.7 Problemas propuestos	194
7.8 Problemas resueltos.....	196
7.9 Contenido de la página Web de apoyo.....	213

Objetivos

- Introducir la necesidad de mantener los datos a lo largo del tiempo, mediante el uso de archivos.
- Conocer los problemas básicos de creación, lectura, modificación y crecimiento de archivos.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

7.1 Introducción.

Hasta el capítulo anterior, todos los programas y los ejemplos analizados tenían en común que los datos procesados (desde simples valores escalares, hasta arreglos y tablas) residían en memoria principal, comúnmente conocida como “memoria RAM” (*random access memory*), que es un dispositivo que no garantiza la persistencia de los valores a lo largo del tiempo. Cuando indicamos que la memoria no garantiza la persistencia a lo largo del tiempo, significa que ante ciertas circunstancias (ajenas al control del programa) un valor almacenado en memoria en un instante t_0 podría no encontrarse disponible en otro instante posterior $t_0 + \Delta t$. Esto podría producirse por alguna de las razones siguientes:

- Por un lado, la memoria principal “ pierde ” la información almacenada si se interrumpe el suministro de energía eléctrica (por ejemplo, cuando se apaga la computadora).
- Por otro lado, todos los datos que un programa mantiene durante su ejecución son desalojados de memoria principal cuando ese programa finaliza.

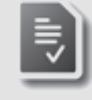
En la mayoría de los casos éste es un escenario poco deseable. Por ejemplo, si se dispone de una tabla con 1000 registros, no sería agradable para el usuario tener que volver a cargar todos esos datos cada vez que enciende la computadora; asimismo, si el programa genera como resultado un conjunto de informes y reportes, interesa tenerlos disponibles cada vez que se los necesite.

La solución al problema de la persistencia de datos consiste en utilizar otros medios de almacenamiento que, debido a sus características físicas, sean capaces de mantener información a lo largo del tiempo, más allá de factores externos. Ejemplos de esos dispositivos son los medios magnéticos (discos duros, cintas, discuetes), medios ópticos (CD-ROM, DVD-ROM) y algunas memorias electrónicas, entre otros. Así, es posible distinguir dos grandes “ dominios ” en cuanto al tipo de almacenamiento en una computadora:

- Memoria principal, donde se almacenan datos (entradas y resultados) e instrucciones de un programa en ejecución. No garantiza la persistencia ante situaciones que no pueden ser controladas por el programa.
- Memoria secundaria, donde se almacenan datos (entradas y resultados) e instrucciones de un programa para garantizar su persistencia, más allá de si el programa está en ejecución o no. Estos medios suelen tener grandes capacidades de almacenamiento.

Si la memoria secundaria es capaz de garantizar la existencia de los datos y, además, la capacidad de almacenamiento es considerablemente grande, ¿ cuál es la razón por la que toda computadora necesita una porción de memoria principal ? La respuesta radica en el tiempo de acceso; esto es, la “ demora ” (o latencia) entre el inicio de una operación de lectura y el momento en que el dispositivo de almacenamiento responde con el dato, o entre el inicio de una operación de escritura y el momento en que ese dato efectivamente se almacena. La latencia de la memoria principal es varios órdenes de magnitud inferior con respecto a los dispositivos de almacenamiento secundario; para una memoria con tecnología DDR2, el tiempo de acceso está en torno a los 10 ns (nanosegundos), mientras que para un disco duro de 160 Gb y 7200 RPM, la latencia puede tomar valores entre 10 y 20 ms (milisegundos). Esto significa que acceder a un disco duro es 1 millón de veces más lento que acceder a memoria principal. Para un programa en ejecución, pagar semejante costo en cada acceso a un dato es prohibitivo.

El almacenamiento de información en dispositivos secundarios se lleva a cabo mediante una estructura conocida como archivo, que puede interpretarse como una secuencia de datos. Estos datos pueden organizarse como una secuencia de bytes (*stream*) o como una secuencia de registros que, en cualquier caso, delimita su final con una marca especial denominada fin de archivo (*EOF, end of file*).



En la Web de apoyo encontrará un capítulo preliminar del libro *Sistemas Operativos*, de Silva, sobre archivos.

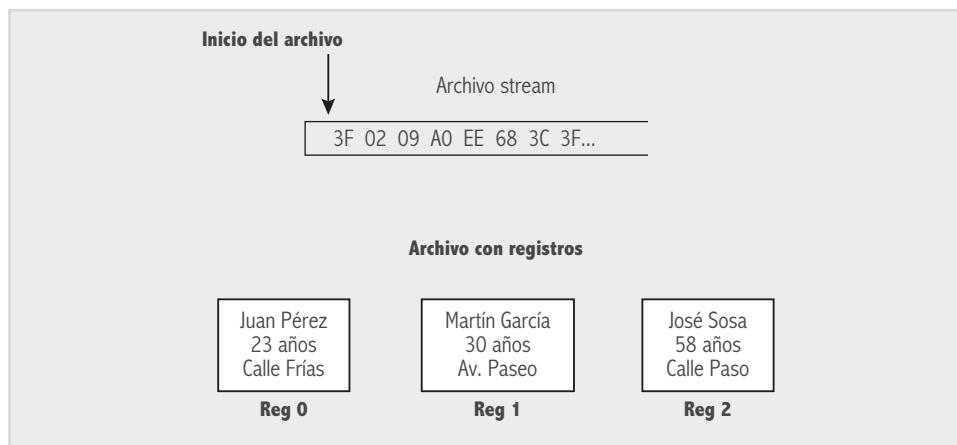


Fig. 7-1. Almacenamiento de información en dispositivos secundarios (archivos).

Así, el término archivo designa una colección de datos almacenados en un dispositivo secundario externo, en formato de texto (archivo de texto o ASCII) o en formato binario (archivo binario). Además, como se ilustrará en ejemplos sucesivos en este capítulo, la información contenida en un archivo puede procesarse de acuerdo con dos esquemas: Secuencial (el archivo se procesa en forma secuencial, en los casos típicos desde el inicio hasta el final) o directo (se accede de manera directa a un registro o byte específico, mediante el uso de un índice, en forma semejante a como se indexa un arreglo lineal).

En el lenguaje C la manipulación de archivos se implementa conforme al estándar ANSI, que define un conjunto completo de funciones de entrada/salida (E/S) para lectura y escritura de datos en un archivo, en formato texto o binario. Ésta es una aproximación de alto nivel, y que adoptaremos en este libro, conocida como sistema de archivos con *buffer*. Además, en sistemas operativos UNIX o derivados, es posible acceder a un archivo desde un programa C mediante un mecanismo de bajo nivel, implementado con llamadas a sistema, y comúnmente denominado sistema de archivos sin *buffer* o tipo UNIX. Este último caso no se considerará en este libro.

Por el lado de Pascal, el uso de archivos se implementa de una manera menos flexible aunque, para un curso de fundamentos de programación, más ordenada.

7.2 Tratamiento de archivos en lenguaje C.

En C, los archivos se organizan como secuencias de bytes, en principio sin estructura predeterminada alguna. Estas secuencias (o "flujos") se conocen como *streams*, y constituyen el mecanismo de abstracción de este lenguaje para acceder a dispositivos externos. Este mecanismo es tan importante que por "dispositivo externo" no sólo nos referimos a medios de almacenamiento, sino también a otros periféricos; así, una impresora puede interpretarse como un archivo de sólo escritura, una conexión de red entrante, como un archivo de sólo lectura, entre otros ejemplos. Los *streams* pueden ser secuencias binarias o de texto (ASCII, *American Standard Code for Information Interchange*).

Cada secuencia que se asocia con un archivo tiene una estructura de control de tipo FILE. Esta estructura se define en el archivo de cabecera stdio.h que, además, suministra los prototipos de las funciones de E/S y define los siguientes tipos:

```
size_t /* entero sin signo */
fpos_t /* entero sin signo */
FILE /* tipo estructura para manejar secuencias */
```



¿Qué es un buffer?

Un *buffer* es simplemente un mecanismo que se utiliza para "amortiguar" o compensar diferencias de velocidades entre dos partes en interacción. Puede ilustrarse con ejemplos de la vida cotidiana, como el sistema de amortiguación en un vehículo, que permite que los movimientos bruscos en las ruedas se traduzcan como movimientos más suaves en la carrocería. Un buzón es otro ejemplo de *buffer*, ya que los usuarios del servicio de correos pueden dejar allí las correspondencias que, con otra frecuencia, serán retiradas por el cartero. Sin el buzón como *buffer*, cada vez que una persona llevase su carta debería haber un cartero esperando para tomarla. En el caso de los archivos, el *buffer* compensa la velocidad de lectura/escritura del programa en ejecución con respecto a la velocidad de lectura/escritura del dispositivo físico asociado que, como se comentó antes, puede ser varios órdenes de magnitud superior para el caso de los discos duros.

No se debe menospreciar el concepto de *buffer*; por lo general es la solución a la mayoría de los problemas en computación.



Llamada a sistema (system call o syscall):

Puede verse como una rutina implementada dentro del sistema operativo y que puede ser invocada desde un programa de usuario. Es la "puerta de entrada" al sistema operativo, la interfaz entre éste y el usuario.

También se definen varias macros, de las cuales las siguientes son las más relevantes en cuanto al manejo de archivos:

```
EOF
SEEK_SET
SEEK_CUR
SEEK_END
```

EOF es el valor devuelto cuando una función intenta leer más allá del final del archivo. Las otras son usadas por `fseek()`, función que permite el acceso directo de datos en un archivo.

Un puntero a un archivo es una referencia a una estructura de datos en memoria que define los atributos y el estado actual del archivo, además de identificar el dispositivo físico con el que está vinculado el archivo. En un programa C cada archivo se identifica (y se accede) utilizando este puntero, cuya declaración se muestra a continuación:

```
FILE *f;
```

En los próximos párrafos se describirán en forma detallada las funciones ANSI para manipulación de archivos en C, que se encuentran en la unidad de biblioteca `stdio.h`.

7.2.1 Apertura de un archivo.

Antes de realizar cualquier operación sobre un archivo, se lo debe abrir. La operación de apertura implica, entre otras cosas, la asignación de recursos asociados al archivo (por ejemplo, memoria para el *buffer*) y la vinculación del archivo lógico (que se accede desde el programa) con el archivo físico (por ejemplo, un disco duro, una interfaz de red, etc.). Esta apertura se lleva a cabo mediante la función `fopen()`. La función `fopen()` retorna una dirección de memoria asociada con el archivo (un "puntero a un archivo"). Si éste no pudiese abrirse o crearse, `fopen()` retorna la constante `NULL`, que indica que la dirección de memoria no está definida. El concepto y manejo de punteros (direcciones de memoria) y el uso de la constante `NULL`, serán tratados con detenimiento en el capítulo 10. Hay diversas posibilidades de apertura de un archivo –sólo escritura, sólo lectura, lectura y escritura, etc.–, que pueden especificarse por medio del segundo argumento de esa función:

```
FILE *f;
f = fopen("mi_archivo.txt", "w");
if (f == NULL)
{
    printf("Error! No pudo crearse el archivo!");
    return 1; /* Cuando el programa termina con un valor distinto
               de cero estamos indicándole al sistema operativo
               una situación de anomalía */
}
...
```

El argumento “w” en la llamada a `fopen()` indica apertura para sólo escritura del archivo con nombre físico `mi_archivo.txt`. Además, si no se especifica lo contrario, el archivo es de texto. A continuación se enumeran los posibles valores que puede tomar el segundo parámetro de `fopen()`:



Encuentre un simulador sobre archivos de acceso secuencial en la Web de apoyo.

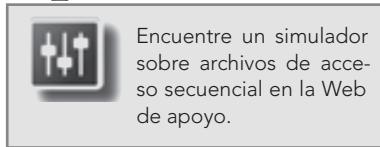


Tabla 7-1 - Posibles valores que puede tomar el segundo parámetro de fopen().

<i>Modo</i>	<i>Significado</i>
"r"	Abre un archivo de texto para lectura
"w"	Abre un archivo de texto para escritura. Si existe previamente, se crea uno vacío
"a"	Abre o crea un archivo para escritura al final de éste
"r+"	Abre un archivo de texto para lectura y escritura
"w+"	Crea un archivo de texto para lectura y escritura. Si existe previamente, se crea uno vacío
"a+"	Abre o crea un archivo de texto para lectura o escritura al final
"rb"	Abre un archivo binario para lectura
"wb"	Crea un archivo binario para escritura. Si existe previamente, se crea uno vacío
"ab"	Abre o crea un archivo binario para escritura al final de éste
"rb+"	Abre un archivo binario para lectura y escritura
"w+"	Crea un archivo binario para lectura y escritura. Si existe previamente, se crea uno vacío
"ab+"	Abre o crea un archivo binario para lectura y escritura al final.

7.2.2 Cierre de un archivo.

Un archivo debe cerrarse cuando se decide que ya no será utilizado. La operación de cierre es importante porque libera los recursos asignados al archivo y, sobre todo, porque fuerza la escritura en el medio externo de datos que podrían estar aún en el *buffer*. Esta operación está implementada por la función `fclose()`.

```
FILE *f;
...
fclose(f);
```

7.2.3 Funciones para manipulación de archivos.

`fopen()` abre un archivo y lo asocia a un dispositivo físico externo (en los casos típicos un medio de almacenamiento). Su prototipo es:

```
FILE *fopen(const char *nombre_arch, const char * modo);
```

`nombre_arch` es un puntero a una cadena de caracteres que representa un nombre válido del archivo y puede incluir una especificación de directorio.

`modo` es un puntero a una cadena de caracteres que determina cómo se abre el archivo.

`fclose()` cierra un archivo que fue previamente abierto mediante `fopen()`:

```
int fclose(FILE *pf);
```

El estándar ANSI define dos funciones equivalentes que escriben un carácter: `putc()` y `fputc()`. Estas dos funciones son idénticas y persisten sólo para preservar la compatibilidad con antiguas versiones de C.

La función `putc()` escribe caracteres en un archivo que se haya abierto con anterioridad para escritura.

```
int putc(int car, FILE *pf);
```

Si `putc()` tiene éxito, devuelve el carácter escrito. Si no, devuelve EOF (la marca de fin de archivo).

El estándar ANSI define dos funciones equivalentes que leen un carácter: `getc()` y `fgetc()`, para preservar la compatibilidad con versiones anteriores a C.

```
int getc(FILE *pf);
```

`getc()` devuelve una marca EOF cuando se alcanzó el final del archivo. Para leer un archivo de texto hasta que se encuentre la marca de fin de archivo, se puede usar el patrón siguiente:

```
do
{
    car = getc(f);
    ...
} while (car != EOF);

int fputs(const char *cad, FILE *pf);
```

`fputs()` escribe la cadena `cad` en el archivo `pf`. Si se produce un error devuelve EOF.

```
char *fgets(char * cad, int longitud, FILE *pf);
```

`fgets()` lee una cadena de la secuencia especificada hasta que se presente un carácter de salto de línea o hasta que se leyó `longitud - 1` caracteres. La cadena resultante acaba con el carácter fin de cadena '\0'. La función retorna un puntero a `cad` si se ejecutó bien o '\0' en caso de error.

```
void rewind(FILE *pf);
```

`rewind()` es una función que inicializa el indicador de posición al principio del archivo indicado por su argumento (esto es, rebobina el archivo).

```
int ferror(FILE *pf);
```

`ferror()` determina si se produjo un error en una operación sobre un archivo. Si hay un error de dicho tipo durante la última operación devuelve un valor distinto de 0, en caso contrario devuelve 0.

```
int remove(char *nombre_archivo);
```

`remove()` borra el archivo especificado. Si tiene éxito, devuelve 0, en caso contrario, un valor distinto de 0.

```
int fflush(FILE *pf);
```

`fflush()` escribe todos los datos almacenados en el *buffer* sobre el archivo asociado con `pf` (volcado del archivo). Al llamar a `fflush()` con un puntero nulo (NULL) se vacían los *buffers* de todos los archivos abiertos. Si tiene éxito, devuelve 0, en caso contrario, devuelve EOF.

Para leer y escribir los tipos de datos que ocupan más de un byte, el sistema de archivos de ANSI C suministra las funciones `fread()` y `fwrite()`, que permiten la lectura y la escritura de bloques de cualquier tipo de datos. Sus prototipos son:

```
size_t fread(void *buffer, size_t numero_bytes, size_t cuenta, FILE *pf);
```

`fread()` devuelve el número de elementos leídos. Este valor puede ser menor que `cuenta` si se encuentra el final del archivo o si se produce un error.

```
size_t fwrite(void *buffer, size_t numero_bytes, size_t cuenta, FILE *pf);
```

`fwrite()` devuelve el número de elementos escritos. Este valor será igual a `cuenta`, a menos que se produzca un error.

En la interfaz de estas funciones se tiene:

`buffer`: Puntero a una región de memoria donde escribir los datos leídos del archivo o puntero a la información que va a escribirse en el archivo;

`numero_bytes`: Número de bytes a leer o escribir, esto es, el tamaño de cada elemento a leer o escribir;

cuenta: Determina cuántos elementos, cada uno de tamaño `numero_bytes` de longitud, se van a leer o escribir;

`pf`: Puntero a una secuencia (previamente abierta con `fopen()`).

```
int fprintf(FILE *pf, const char *cadena_de_control, ...);
```

```
int fscanf(FILE *pf, const char *cadena_de_control, ...);
```

`fprintf()` y `fscanf()` se comportan como `printf()` y `scanf()`, respectivamente, pero operando sobre el archivo `pf`.

```
FILE *fopen(const char *nombre_archivo, const char *modo, FILE *secuencia);
```

`fopen()` asocia una secuencia con un archivo nuevo. Puede usarse para asociar una secuencia estándar (`stdin`, `stdout` o `stderr`) con un archivo nuevo. Estas tres secuencias se abren por omisión con la ejecución de todo programa C (`stdin` es entrada estándar, `stdout` es salida estándar y `stderr` es la salida de errores).

En el ejemplo siguiente se crea un archivo de texto con nombre físico `mi_archivo.txt` y nombre lógico `f`. Los caracteres ingresados por el usuario a través del teclado (`stdin`) se escriben en el archivo. Con posterioridad el archivo vuelve a abrirse, pero para lectura. Se lee todo su contenido y se lo muestra en pantalla.

```
#include <stdio.h>
int main()
{
    FILE *f; /* Puntero al archivo */
    char car;
    /* Se crea un archivo de texto para escritura */
    f = fopen("mi_archivo.txt", "w");
    if (f == NULL)
    {
        printf("Error! No pudo crearse el archivo!");
        return 1;
    }
    /* Se leen caracteres desde teclado (stdin) y */
    /* se escriben en el archivo abierto antes. */
    do
    {
        car = getc(stdin);
        fprintf(f, "%c", car);
    } while (car != '.');
    fclose(f);
    /* Se vuelve a abrir el archivo, pero ahora para lectura. */
    f = fopen("mi_archivo.txt", "r");
    if (f == NULL)
    {
```

```

        printf("Error! No pudo abrirse el archivo!");
        return 1;
    }
    /* Se lee todo el archivo y se lo muestra en pantalla. */
    do
    {
        car = getc(f);
        printf("%c", car);
    } while (car != EOF);
    fclose(f);
    return 0;
}

```



Un puntero es una variable escalar, simple, que almacena un número entero que representa una dirección de memoria.

7.2.4 Archivos como parámetros de funciones.

Desde el punto de vista del programa, un archivo no es más que un puntero a FILE, como se muestra en la declaración siguiente:

```
FILE *arch; /* Puntero al archivo */
```

Un puntero es una variable escalar, simple, que almacena un número entero que representa una dirección de memoria. Por lo tanto, el pasaje de un archivo como parámetro de una función es equivalente al pasaje de cualquier dato simple, como se muestra en el ejemplo siguiente:

```

#include <stdio.h>
void mostrar_archivo(FILE *f)
{
    char c;
    rewind(f);
    while ( !feof(f) )
    {
        c = fgetc(f);
        printf("%c", c);
    }
}
int main()
{
    FILE *arch; /* Puntero al archivo */
    /* Se abre un archivo de texto para escritura */
    arch = fopen("mi_archivo.txt", "r");

```

```

if (arch == NULL)
{
    printf("Error! No pudo abrirse el archivo!\n");
    return 1;
}

/* Llamada a la función */
mostrar_archivo(arch);
/* Se cierra el archivo */
fclose(arch);
return 0;
}

```

Como puede apreciarse, el pasaje de un archivo como parámetro no reviste complejidad adicional alguna.

7.3 Tratamiento de archivos en lenguaje Pascal.

Para declarar una variable archivo en Pascal es necesario indicar el tipo de elemento que lo compone, esto es, definir la naturaleza de sus registros. Esta declaración puede hacerse directamente en la sección de declaración de variables, o declarar con anterioridad el tipo de archivo en la sección de tipos y con posterioridad las variables correspondientes en la sección de variables. Así, la sentencia:

```
var archivo_enteros: file of integer;
```

es equivalente a:

```
type t_archivo_enteros = file of integer;
var archivo_enteros: t_archivo_enteros;
```

7.3.1 Apertura de un archivo.

Antes de la apertura, es necesario indicar en forma explícita a qué archivo físico estará vinculada la variable de archivo lógico, para lo cual se utiliza el procedimiento `assign()`. Después de esta “vinculación”, cualquier operación sobre la variable (archivo lógico) afectará al archivo físico correspondiente. El formato del procedimiento es:

```
assign(var_arch, nom_arch)
```

donde `var_arch` es una variable de tipo archivo y `nom_arch` es una cadena de caracteres que contiene el nombre del archivo físico asociado (incluidas, eventualmente, expresión de unidad de disco y ruta de acceso).

Luego de la asignación se procede a la apertura o la creación del archivo. El procedimiento `reset()` abre un archivo para lectura posicionando el puntero a registro en el primer registro. También puede utilizarse el procedimiento `reset()` sobre archivos que ya estén abiertos para lectura, y el resultado de la operación será reposicionar el puntero de lectura en el primer registro. El formato de este procedimiento es:

```
reset(var_arch)
```



Antes de la apertura, es necesario indicar en forma explícita a qué archivo físico estará vinculada la variable de archivo lógico, para lo cual se utiliza el procedimiento `assign()`.



El procedimiento `reset()` abre un archivo para lectura posicionando el puntero a registro en el primer registro. También puede utilizarse sobre archivos que ya estén abiertos para lectura, y el resultado de la operación será reposicionar el puntero de lectura en el primer registro.



A menudo cuando un programa intenta acceder a un archivo, no se sabe con certeza si éste existe (en cuyo caso debe abrirse) o no (y, por lo tanto, debe crearse), por lo que el uso de la función `reset()` o `rewrite()` no es tan evidente. En escenarios como éste, suele utilizarse la construcción de código siguiente:

```
{$I-}
reset(mi_archivo);
{$I+}
if (iorestart <> 0) then
begin
{El archivo no pudo abrirse.}
  rewrite(mi_archivo);
end;
...
```

El uso de las directivas `{$I-}` y `{$I+}` permite indicar al compilador que se ignoren los posibles errores relacionados con entrada/salida que podrían producirse. De esta manera, si el archivo en cuestión no existe, la llamada a `reset()` no aborta la ejecución del programa. Para conocer el resultado de la ejecución de `reset()`, puede consultarse el contenido de la variable predefinida `iorestart` (I/O result). Si esta variable tiene un valor diferente a 0, significa que se produjo un error en la última operación de entrada/salida, en cuyo caso deberá llamarse a `rewrite()` para crear el archivo.

donde `var_arch` es la variable archivo.

Por su parte, el procedimiento `rewrite()` abre el archivo para escritura (destruyendo el contenido previo si el archivo ya existiera). El formato de este procedimiento es:

```
rewrite(var_arch)
```

7.3.2 Cierre de un archivo.

El cierre de un archivo (y la liberación de sus recursos asociados) se realiza con el procedimiento `close()`, cuya sintaxis es la siguiente:

```
close(var_arch)
```

donde `var_arch` es la variable archivo. Recuérdese que el cierre de un archivo es importante, además, porque permite escribir datos que pudiesen haber quedado en el *buffer*.

7.3.3 Funciones y procedimientos para manipulación de archivos.

`read()` se utiliza para introducir el contenido de un registro del archivo en una variable de memoria definida del mismo tipo de dato que el registro leído. Su prototipo es:

```
read(var_arch, nom_reg)
```

`write()` escribe en un registro del archivo el contenido de una variable de memoria definida del mismo tipo. Su prototipo es:

```
write(var_arch, nom_reg)
```

El tratamiento de una estructura repetitiva, como es el caso de un archivo secuencial, requiere detectar la marca de fin de archivo, con el objeto de evitar errores en tiempo de ejecución al intentar leer datos que no existen. En cada iteración del bucle se lee un registro del archivo, avanzando (en forma automática) el puntero de lectura una posición hacia el final. En la lectura del último registro el puntero a registro se posiciona sobre la marca de fin de archivo (EOF, *end of file*) y establece la función lógica `eof()` en verdadero. Esta función tiene el formato siguiente:

```
eof(var_arch)
```

Dado que un archivo al comienzo puede estar vacío, el tratamiento típico debe tener la forma siguiente:

```
while (not eof(mi_archivo)) do
begin
  read(mi_archivo, reg);
  ...
end;
```

7.3.4 Archivos como parámetros de procedimientos y funciones.

El pasaje de parámetros de tipo archivo se hará siempre por referencia. En ningún caso es posible pasarlo por valor; por tanto, deben ir siempre precedidos por la palabra reservada `var`, como se muestra en los ejemplos siguientes:

```
procedure procesar(var arch: t_archivo_alumnos);
...
function obtener_deuda(var arch_cli: t_archivo_clientes);
...
```

7.4 Archivos de acceso directo.

Hasta aquí, el tratamiento sobre archivos se llevó a cabo de forma secuencial, procesando todos los elementos, de a uno por vez, desde el inicio hasta el final. Sin embargo, éste podría no ser siempre el caso y, en determinadas situaciones, interesaría acceder a un punto específico del archivo para procesar un dato en particular o escribir en una posición determinada. Este tipo de acceso “directo” es soportado tanto por C como por Pascal.

La declaración de un archivo de acceso directo es idéntica a la de otros archivos y sólo se distingue de ellos por las funciones de posicionamiento en registro.

Un puntero del archivo (puntero a registro) memoriza el número del registro o byte correspondiente para una operación de lectura o escritura, como se muestra en la figura siguiente:

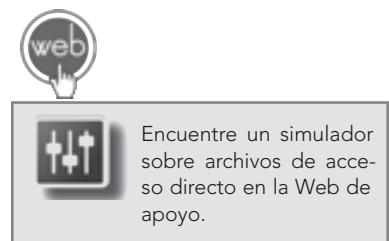
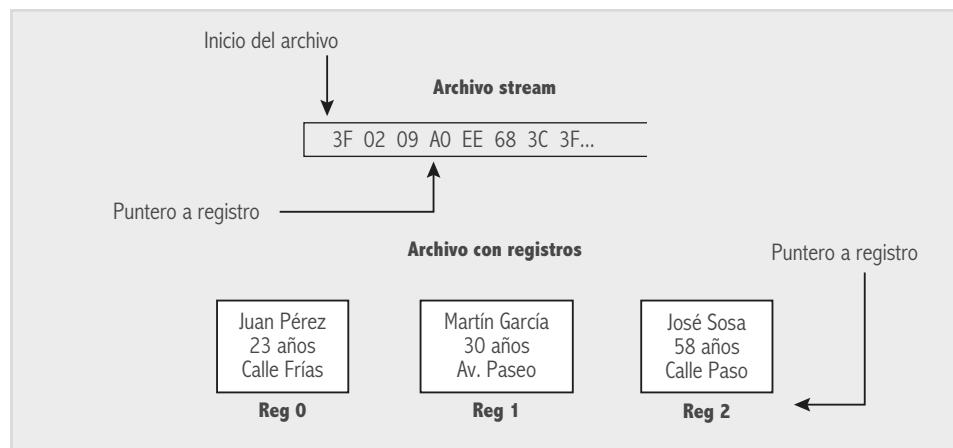


Fig. 7-2. Puntero del archivo que memoriza el byte correspondiente para una operación de lectura o escritura.

Cuando un archivo se abre, su puntero indica el primer registro. Después de cada operación de lectura o escritura este puntero se incrementa en uno, y pasa a apuntar al siguiente registro o byte. Es importante resaltar que sólo pueden accederse en forma directa los archivos de formato binario; en cambio, los de formato texto o ASCII siempre son de acceso secuencial.

7.4.1 Archivos de acceso directo en lenguaje C.

Para posicionarse en un registro o byte de un archivo de acceso directo, se utiliza la función de biblioteca `fseek()`:

```
int fseek(FILE *pf, long numbytes, int origen);
```

Esta función cambia el valor del puntero a registro. Sirve para tener acceso directo a un registro o byte en particular dentro del archivo.

`pf`: Puntero al archivo devuelto por una llamada a `fopen()`.

`numbytes`: Un entero largo, que es el número de bytes a partir de `origen` que supondría la nueva posición. Este valor podría ser negativo, si se desea retroceder el puntero.

`origen`: Alguna de las siguientes macros:

- `SEEK_SET` `numbytes` es relativo al principio de archivo.
- `SEEK_CUR` `numbytes` es relativo a la posición actual.
- `SEEK_END` `numbytes` es relativo al fin del archivo.



Para situarse a `numbytes` desde el principio del archivo, `origen` debe ser igual a `SEEK_SET`; para situarse a partir de la posición actual, se usa `SEEK_CUR`, y para situarse a partir del final del archivo, se usa `SEEK_END`.

El programa siguiente lee un archivo de texto carácter por carácter, y lo convierte a mayúsculas. Cuando lee un carácter, el puntero a registro avanza automáticamente a la posición siguiente; así, antes de escribir el carácter en mayúsculas es preciso retroceder el puntero a registro una posición. Por último, se lleva el puntero al inicio, y se muestra el contenido del archivo por salida estándar.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    FILE *f; /* Puntero al archivo */
    char car;
    /* Se abre el archivo para lectura y escritura. */
    f = fopen("mi_archivo.txt", "r+");
    if (f == NULL)
    {
        printf("Error! No pudo abrirse el archivo!");
        return 1;
    }
    /* Se lee todo el archivo y se convierte a mayúsculas. */
    while ( (car = getc(f)) != EOF )
    {
        /* Se convierte el carácter a mayúsculas. */
        car = toupper(car);
        /* Se mueve el puntero a registro una */
        /* posición hacia atrás. */
        fseek(f, -1, SEEK_CUR);
        /* Se escribe el carácter en mayúsculas. */
        /* Esta escritura también avanza el puntero. */
        putc(car, f);
    }
    /* Se lleva el puntero a la primera posición. */
    fseek(f, 0, SEEK_SET);
    /* Se lee todo el archivo y se lo muestra en pantalla. */
    while ( (car = getc(f)) != EOF )
    {
        printf("%c", car);
    }
    printf("\n");
    fclose(f);
    return 0;
}
```



En Pascal, el procedimiento `seek()` permite seleccionar un registro específico del archivo por su número de registro, para su uso en una operación de lectura o escritura.

7.4.2 Archivos de acceso directo en lenguaje Pascal.

En Pascal, el procedimiento `seek()` permite seleccionar un registro específico del archivo por su número de registro, para su uso en una operación de lectura o escritura. Su formato es:

```
seek(var_arch, num_reg)
```

Donde nom_arch es el archivo lógico, y num_reg es el número de registro al que se intenta acceder (En C los registros de un archivo se enumeran en forma consecutiva a partir de 0 : 0,1,2...).

Además, Pascal brinda funcionalidades adicionales para el tratamiento de archivos directos. Por ejemplo, la función filesize() devuelve el tamaño del archivo en formato de entero largo, indicando el número de registros almacenados. Esta función toma el nombre del archivo como único argumento. Si se trata de un archivo vacío, retorna el valor 0. Ejemplo:

```
tamanio := filesize(arch_clientes);
```

La función filepos() devuelve en formato de entero largo el número del registro actual (o sea, el valor actual del puntero a registro). Toma como argumento la variable archivo:

```
reg_actual := filepos(arch_clientes);
```

Algunos ejemplos adicionales que suelen ser muy útiles en la práctica cuando se trabaja con archivos de acceso directo son:

```
seek(arch, 0);           { primer registro }
seek(arch, filesize(arch)-1); { último registro }
seek(arch, filesize(arch)); { añade al final }
```

7.5 Operaciones entre archivos.

Los archivos, vistos como conjuntos de registros, admiten operaciones como apareo, mezcla, intersección y unión. Trataremos cada una de estas técnicas, considerando con más detalle las operaciones de **apareo** y **mezcla**.

En lo que sigue se supondrá que los archivos están ordenados en forma ascendente y, además, que todos tienen igual tipo de registro.

7.5.1 Apareo.

La primera de estas operaciones, el apareo, consiste en la actualización de un archivo (MAESTRO) a partir de otro archivo (NOVEDADES). O sea que el archivo MAESTRO se debe actualizar a partir de los registros presentes en NOVEDADES, y debe obtenerse un tercer archivo, NUEVO MAESTRO. La estructura de los registros del archivo NOVEDADES coincide con la del archivo MAESTRO, además, incluye un campo que indica la operación a realizar: Alta, baja o modificación. Recuérdese que tanto MAESTRO como NOVEDADES están ordenados en forma ascendente.

El algoritmo que se utiliza para aparear archivos consiste en tomar el primer registro de MAESTRO y el primer registro de NOVEDADES. Comparar los campos por los cuales los archivos se encuentran ordenados (por ejemplo, el número de legajo si cada registro representara a empleados de una empresa), si el registro considerado menor está en::

- MAESTRO: Se graba este registro en el archivo NUEVO MAESTRO, y se avanza en MAESTRO Una posición.
- NOVEDADES: Si es un alta, entonces se graba este registro en NUEVO MAESTRO. Si es baja, hay que devolver un mensaje de error, ya que se está intentando borrar del archivo MAESTRO un registro que no existe. Igualmente, si es una modificación deberá devolverse un error, porque no podemos modificar un registro que no existe en el archivo MAESTRO. Cualquiera sea la operación, luego se avanza una posición en el archivo NOVEDADES.



Apareo:

El apareo de archivos implica la actualización de un archivo tomando en consideración los registros de un segundo archivo.



Alta: generación de un nuevo registro.
Baja: eliminación de un registro existente.
Modificación: actualización de un registro existente.
Es frecuente referirse a estas tres operaciones como ABM.

- **MAESTRO Y NOVEDADES:** Si es un alta, deberá informarse el error, ya que se está intentando dar de alta un registro que ya existe en el archivo MAESTRO. Si se trata de una baja, no se hace nada. Y si fuese una modificación, entonces se debe grabar el registro leído del archivo NOVEDADES en el archivo NUEVO MAESTRO. Después de cualquiera de estas operaciones, se avanza una posición en ambos archivos.

Cuando alguno de los dos archivos (MAESTRO o NOVEDADES) finaliza (o llega a EOF), entonces deberá copiarse el resto del archivo que aún contiene registros en NUEVO MAESTRO.

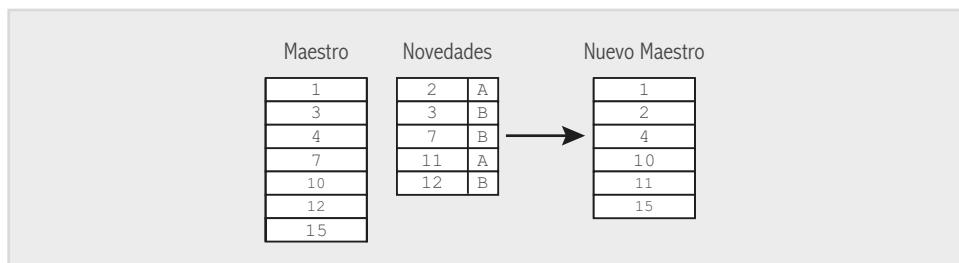


Fig. 7-3. Apareo.

7.5.2 Mezcla.

La mezcla consiste en obtener un archivo nuevo (ordenado) a partir de dos o más archivos también ordenados. El archivo que se obtiene contendrá todos los registros de los archivos que pretendemos mezclar.

Suponiendo que contamos con dos archivos, que llamaremos ARCHIVO1 y ARCHIVO2, y un archivo de salida llamado MEZCLA, el algoritmo es el siguiente:

Se toma el primer registro de cada uno de los archivos que están siendo mezclados. Comparando los campos por los cuales los archivos se encuentran ordenados, si el registro considerado menor está en:

- ARCHIVO1: Se graba este registro en MEZCLA, y se avanza una posición.
- ARCHIVO2: Se graba este registro en MEZCLA, y se avanza una posición.
- ARCHIVO1 y ARCHIVO2: Se graban ambos registros en el archivo MEZCLA, y se avanzan ambos archivos de entrada.

Cuando alguno de los archivos, ARCHIVO1 o ARCHIVO2, se haya procesado por completo, entonces se copia el resto del archivo que aún contenga registros en MEZCLA. Por último, MEZCLA contiene todos los registros de ARCHIVO1 y de ARCHIVO2, ordenados en forma ascendente.

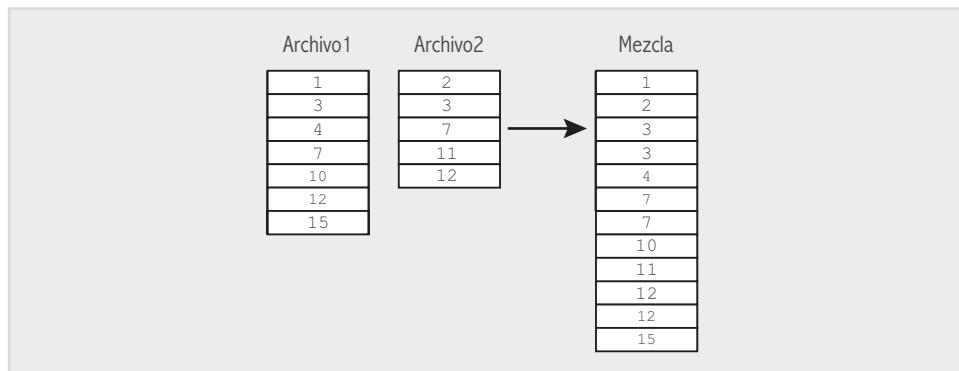


Fig. 7-4. Mezcla.

7.5.3 Intersección.

Como con conjuntos, la intersección de archivos consiste en obtener un archivo (ordenado) a partir de dos archivos, o más, ordenados en forma ascendente, que contenga sólo aquellos registros que son comunes a los archivos de entrada. Debido a que, en la práctica, es una operación poco usada (respecto del apareo y la mezcla), nos conformaremos con dar la definición.

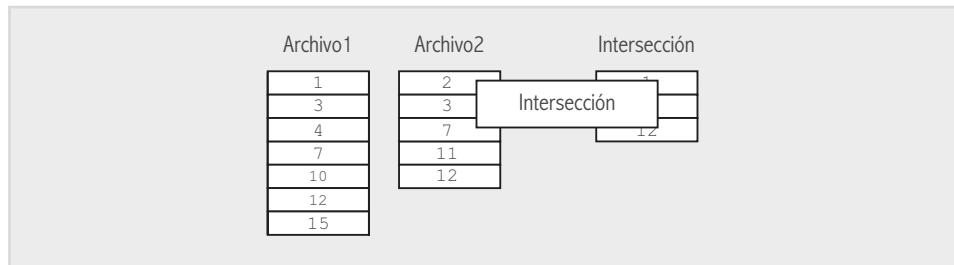


Fig. 7-5. Intersección.

7.5.4 Unión.

A partir de dos archivos o más, se pretende obtener uno nuevo que contenga todos los registros de los archivos anteriores. Nótese que, a diferencia de la mezcla, en la unión no se admiten registros repetidos en el archivo de salida.

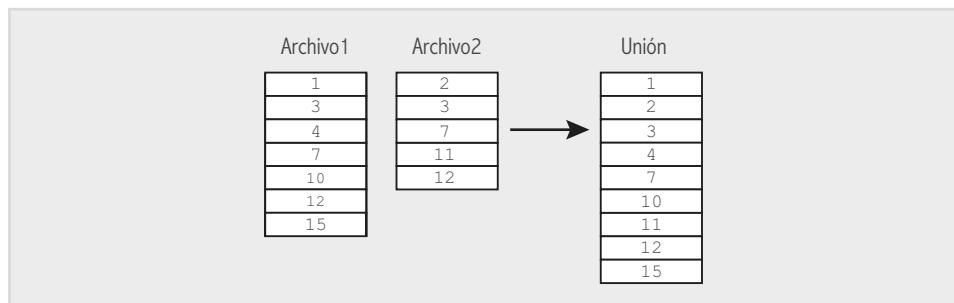


Fig. 7-6. Unión.

7.6 Resumen.

La memoria principal de una computadora no garantiza la persistencia de los datos que almacena a lo largo del tiempo; esto significa que un valor almacenado en memoria en un instante t_0 podría no encontrarse disponible en otro instante posterior $t_0 + \Delta t$. Sin embargo, en ocasiones es necesario mantener datos almacenados a través del tiempo. Por ejemplo, si se dispone de una tabla con 1 000 registros, no sería agradable para el usuario tener que volver a cargar todos esos datos cada vez que enciende la computadora; o si el programa genera como resultado un conjunto de informes y comunicaciones, interesa tenerlos disponibles cada vez que se los necesite.

La solución al problema de la persistencia de datos consiste en utilizar otros medios de almacenamiento que, debido a sus características físicas, sean capaces de mantener información a lo largo del tiempo, independientemente de factores externos. Ejemplos de esos dispositivos son los medios magnéticos (discos duros, cintas, discuetes), los medios ópticos (CD-ROM, DVD-ROM) y algunas memorias electrónicas, entre otros.

El almacenamiento de información en dispositivos secundarios se lleva a cabo mediante una estructura conocida como archivo, que puede ser interpretado como una secuencia de datos.



Intersección:

Generar un archivo con los datos comunes a dos o más archivos, conservando el mismo orden.



Unión:

A partir de dos archivos, o más, se pretende obtener uno nuevo que contenga todos los registros de los archivos anteriores. No se admiten registros repetidos en el archivo de salida.

Estos datos pueden organizarse como una secuencia de bytes (*stream*) o como una secuencia de registros que, en cualquier caso, delimita su final con una marca especial denominada fin de archivo (EOF, *end of file*).

Así, el término archivo designa una colección de datos almacenados en un dispositivo secundario externo, en formato de texto (archivo de texto o ASCII) o en formato binario (archivo binario). Además, la información contenida en un archivo puede procesarse de acuerdo con dos esquemas: Secuencial (el archivo se procesa de manera secuencial, típicamente desde el inicio hasta el final) o directo (se accede en forma directa a un registro o byte específico, mediante el uso de un índice).

En C, los archivos se organizan como secuencias de bytes, en principio sin ninguna estructura predeterminada. Estas secuencias (o "flujos") se conocen como *streams*, y constituyen el mecanismo de abstracción de este lenguaje para acceder a dispositivos externos. Por el lado de Pascal, el uso de archivos se implementa de una manera menos flexible aunque, para un curso de fundamentos de programación, más ordenada.

Algunas de las operaciones que pueden aplicarse sobre un archivo son las siguientes:

- **Apertura:** Se establece una vinculación entre el programa y la porción del medio externo donde residen los datos. Luego de la apertura es posible escribir datos en el archivo externo (si es una apertura para escritura) o leer datos de él (si la apertura es de sólo lectura).
- **Cierre:** Esta operación es importante porque libera los recursos asignados al archivo y, principalmente, porque fuerza la escritura en el medio externo de datos que podrían estar aún en la memoria principal (*buffer*).
- **Escritura:** Y lectura de datos.

Además, es posible aplicar operaciones entre archivos. El apareo y la mezcla son algunas de las más comunes. La primera de ellas consiste en la actualización de un archivo (MAESTRO) a partir de otro archivo (NOVEDADES), que generan como resultado un tercer archivo, NUEVO MAESTRO. Por su parte, la mezcla consiste en obtener otro archivo, a partir de dos o más archivos ordenados de manera ascendente, que incluya a todos los anteriores, y que también esté ordenado en forma ascendente (de manera análoga al concepto de mezcla en arreglos lineales).

7.7 Problemas propuestos.

- 1) El departamento de personal de una universidad tiene registros del nombre, el sexo y la edad de cada uno de los profesores que trabajan en ella. Escriba un programa que calcule e imprima los siguientes datos (primero debe llenar el archivo de datos):

- a. Edad promedio del grupo de profesores.
- b. Nombre del profesor más joven del grupo.
- c. Nombre del profesor de más edad.
- d. Número de profesores con edad mayor que la del promedio.
- e. Número de profesores con edad menor que la del promedio.

Nota: Cada una de las opciones del programa debe manejarse por medio de un menú, además de la opción para ingresar por primera vez los datos de los profesores.

- 2) Al momento de su ingreso en el hospital, a un paciente se le solicitan los siguientes datos: nombre, edad, sexo, domicilio (calle, número, ciudad), teléfono, obra social (este campo tendrá el valor verdadero si el paciente tiene O.S., y falso si no la posee).

Escriba un programa que pueda llevar a cabo las siguientes operaciones (primero debe llenar el archivo de datos):

- Listar los nombres de todos los pacientes hospitalizados.
- Obtener el porcentaje de pacientes hospitalizados en las siguientes categorías:
 - i. Niños: Hasta 13 años.
 - ii. Jóvenes: Mayores de 13 años y menores de 30 años.
 - iii. Adultos: Mayores de 30 años.
- Obtener el porcentaje de hombres y mujeres hospitalizados.

- Dado el nombre de un paciente, listar todos los datos relacionados con él.
- Calcular el porcentaje de pacientes que poseen obra social.

Nota: Cada una de las opciones del programa, debe ser manejada por medio de un menú, además de la opción para ingresar por primera vez los datos de los pacientes.

- 3) Una inmobiliaria tiene información sobre departamentos en alquiler. De cada uno se conoce:

- Número: Es un entero que identifica al departamento.
- Extensión: Superficie del departamento, en metros cuadrados.
- Estado: Excelente, bueno, regular, malo.
- Precio de alquiler: Es un valor real.
- Disponible: Verdadero si está disponible para el alquiler, y falso si ya está alquilado.

Diariamente acuden muchos clientes a la inmobiliaria solicitando información. Escriba un programa que sea capaz de realizar las siguientes operaciones sobre la información disponible (primero debe llenar el archivo con la información de los departamentos):

- Liste los datos de los departamentos disponibles que tengan un precio inferior o igual que un cierto valor P (que debe ser leído).
- Liste los datos de los departamentos disponibles que tengan una superficie mayor o igual que un cierto valor dado E (que debe ser leído) y una ubicación excelente.
- Liste el monto de la renta de todos los departamentos alquilados.
- Llega un cliente que solicita alquilar un departamento. Si existe un departamento con una superficie mayor o igual que la deseada, con un precio y un estado que se ajustan a las necesidades del cliente (estos parámetros deben ser leídos), el departamento se alquila. Además, se deben actualizar los datos que correspondan (disponible de verdadero a falso).
- Se vence un contrato de alquiler, si no se renueva, actualizar los datos que corresponda (disponible de falso a verdadero).
- Se ha decidido aumentar los valores del alquiler en un $n\%$. Actualizar los precios de los alquileres de los departamentos no alquilados.

Nota: Cada una de las opciones del programa debe ser manejada por medio de un menú, además de la opción para ingresar por primera vez los datos de los departamentos.

- 4) Una compañía distribuye N productos a distintos comercios de la ciudad. Para ello almacena en un archivo toda la información relacionada con su mercadería:

- Número: Permite identificar a cada producto.
- Descripción del producto.
- Stock: Cantidad que se mantiene del producto.
- Stock mínimo: Cantidad mínima del producto a mantener en bodega.
- Precio unitario.

Escribir un programa que pueda llevar a cabo las siguientes operaciones:

- Venta de un producto: Se deben actualizar los campos que correspondan y verificar que el nuevo stock no esté por debajo del mínimo.
- Reaprovisionamiento de un producto: Se deben actualizar los campos que correspondan.
- Actualizar el precio de un producto.
- Informar sobre un producto: Se deben proporcionar todos los datos relacionados con un producto.

Nota: Cada una de las opciones del programa debe manejarse por medio de un menú, además de la opción para ingresar por primera vez los datos de los productos.

- 5) Un supermercado necesita de su ayuda para la creación de un programa que automatice el proceso de ventas. El dueño del supermercado necesita que cada uno de sus productos esté codificado (para lo cual puede usar la clasificación de familias de productos, por ejemplo, fideos, y luego una clasificación de un producto particular dentro de esta familia, por ejemplo, ravióles marca "XX" de 500 g) y que se mantenga la información de relevancia de éstos, junto a su precio. La idea es que cuando el cliente pase por la caja, la cajera ingrese el código del producto o su nombre, la cantidad que el cliente lleva de ese producto y automáticamente el sistema le entregue el precio por cada producto y le genere la factura final, donde, además de aparecer el monto de la compra, se lea la cantidad de artículos que el cliente lleva.

Asimismo, se necesita generar un listado de los productos más vendidos y los montos vendidos en un día.

Defina qué información almacenará de los productos para dar respuesta a los requerimientos.

- 6) La biblioteca de una universidad le ha solicitado el desarrollo de un programa que optimice el préstamo de libros de sus dependencias. Para desarrollarlo, puede usar como referencia el proceso que actualmente se utiliza en la biblioteca, pero no

debe olvidar que los libros están clasificados de manera general para usarse sólo en las dependencias de la biblioteca o para prestarse y utilizarlos fuera de ella; además, cada libro posee una cantidad máxima de días de préstamo (que depende de la especialidad y las políticas de la biblioteca), pertenece a un área y especialidad, y posee características particulares.

Además, los alumnos que pueden solicitar un libro pertenecen a una carrera y se debe registrar qué libro se le prestó a cada uno y cuándo lo devolvió, para, en caso contrario, sancionarlo con suspensión de uso de la biblioteca.

También se precisa obtener información de los alumnos que están con devoluciones pendientes, saber qué libro es el que más se prestó y cuál es la especialidad más consultada.

Defina qué información almacenará de los libros y los alumnos para dar respuesta a los requerimientos.

- 7) Un banco le ha encargado la creación de un sistema que permita automatizar los giros de dinero de sus arcas (use como referencia las operaciones en cajeros automáticos). Para esta

tarea se le informó que cada cliente debe poseer una clave secreta, además de una cuenta corriente y una línea de sobregiro (esta última con un monto que el banco define). Un cliente puede hacer las operaciones de giro, depósito o consulta de saldo de cualquiera de las cuentas, además de la opción de traspaso de fondos desde la cuenta corriente hacia la línea de sobregiro o cambio de clave secreta.

Asimismo, se desea obtener la información de todos los clientes y saber cuál es el monto que el banco posee si se suman todos los saldos de las cuentas corrientes de todos los clientes del banco.

Defina qué información almacenará de los clientes (además de la ya mencionada) para dar respuesta a los requerimientos, además de la opción para ingresar por primera vez los datos de los clientes.

7.8 Problemas resueltos.

- 1- Cree un archivo secuencial llamado ENTEROS.DAT cuyos elementos sean números enteros.

El programa debe:

- Abrir el archivo para lectura/escritura en modo binario.
- Cargar datos en el archivo.
- Leer el primer registro.
- Imprimir ese registro.
- Cerrar el archivo.

Es preciso aclarar que, en este problema resuelto, utilizamos la palabra “registro” para referirnos a una variable atómica (ya que, en este caso, los elementos del archivo son números enteros).

Solución en lenguaje C

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

#define EOFP -1

int main()
{
    FILE *fp;
    int numero, extraido, n;

    if ((fp=fopen("enteros.dat","wb+")) == NULL)
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```
{  
    printf("No se puede abrir el archivo.\n");  
    return 1;  
}  
  
do  
{  
    printf("Ingrese un entero (%d para terminar): ", EOF);  
    scanf("%d", &numero);  
    if (numero != EOF)  
    {  
        n = fwrite(&numero, sizeof(numero), 1, fp);  
        if (n != 1)  
        {  
            printf("Error escribiendo en el archivo.\n");  
            fclose(fp);  
            return 1;  
        }  
    }  
} while (numero != EOF);  
  
rewind(fp);  
  
n = fread(&extraido, sizeof(extraido), 1, fp);  
if (n != 1)  
{  
    printf("Error leyendo del archivo.\n");  
    fclose(fp);  
    return 1;  
}  
printf("%d ", extraido);  
fclose(fp);  
  
return 0;  
}
```

Solución en lenguaje Pascal

```
PROGRAM cap07_ej01;  
  
CONST EOF = -1;  
  
TYPE t_arch = FILE OF INTEGER;  
  
VAR fp: t_arch;  
    numero, extraido: INTEGER;  
  
{ Programa Principal }
```

```
BEGIN

ASSIGN(fp,'enteros.dat');

{$I-}
REWRITE(fp);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
  WRITELN('No se puede abrir el archivo.');
  EXIT;
END;

REPEAT
  WRITE('Ingrese un entero (', EOFF, ' para terminar): ');
  READLN(numero);
  IF (numero <> EOFF) THEN
    BEGIN
      {$I-}
      WRITE(fp, numero);
      {$I+}
      IF (IORESULT <> 0) THEN
        BEGIN
          WRITELN('Error escribiendo en el archivo.');
          CLOSE(fp);
          EXIT;
        END;
      END;
    END;
  UNTIL (numero = EOFF);

RESET(fp);

{$I-}
READ(fp, extraido);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
  WRITELN('Error leyendo del archivo.');
  CLOSE(fp);
  EXIT;
END;

WRITELN(extraido);
CLOSE(fp);

END.
```

2- Cree un programa que abra un archivo binario (o lo cree si no existe), cuyos registros contengan los campos:

- Nombre y apellido (hasta 40 caracteres).
- DNI (hasta 10 caracteres).
- Padrón (hasta 10 caracteres).

A modo ilustrativo, sólo se operará con un registro de datos.

Solución en lenguaje C

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

#define EOF -1

typedef struct {
    char nombre[40+1];
    char dni[10+1];
    char padron[10+1];
} t_estudiante;

int main()
{
    FILE *fp;
    t_estudiante estudiante;

    if ((fp=fopen("estudiantes.dat","wb+")) == NULL)
    {
        printf("No se puede abrir el archivo.\n");
        return 1;
    }

    /* Carga de datos */
    printf("Nombre: ");
    gets(estudiante.nombre);
    printf("DNI: ");
    gets(estudiante.dni);
    printf("Padrón: ");
    gets(estudiante.padron);

    /* Escritura del registro en el archivo */
    fwrite(&estudiante, sizeof(t_estudiante), 1, fp);

    rewind(fp);

    /* Lectura del primer registro */
    fread(&estudiante, sizeof(t_estudiante), 1, fp);

    printf("Nombre: %s\n", estudiante.nombre);
    printf("DNI: %s\n", estudiante.dni);
```

```

printf("Padrón: %s\n", estudiante.padron);
fclose(fp);

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap07_ej02;

CONST EOF = -1;

TYPE t_estudiante = RECORD
    nombre: STRING[40];
    dni:     STRING[10];
    padron: STRING[10];
END;

t_arch = FILE OF t_estudiante;

VAR fp: t_arch;
    estudiante: t_estudiante;

{ Programa Principal }
BEGIN

ASSIGN(fp,'estudiantes.dat');

{$I-}
REWRITE(fp);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo.');
    EXIT;
END;

{ Carga de datos }
WRITE('Nombre: ');
READLN(estudiante.nombre);
WRITE('DNI: ');
READLN(estudiante.dni);
WRITE('Padrón: ');
READLN(estudiante.padron);

{ Escritura del registro en el archivo }
WRITE(fp, estudiante);

RESET(fp);

```

```
{ Lectura del primer registro }
READ(fp, estudiante);

WRITELN('Nombre: ', estudiante.nombre);
WRITELN('DNI: ', estudiante.dni);
WRITELN('Padrón: ', estudiante.padron);

CLOSE(fp);

END.
```

- 3- Cree un programa que haga un archivo llamado NUMEROS.DAT cuyos elementos sean números enteros. Luego, almacene el valor 250 en la posición 10, muestre el contenido del archivo modificado y, por último, cierre el archivo en cuestión.

Solución en lenguaje C

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

int main()
{
    FILE *fp;
    int num = 250;
    int i;

    /* Abre el archivo para lectura/escritura en modo binario. */
    if ((fp=fopen("numeros.dat","wb+")) == NULL)
    {
        printf("No se puede abrir el archivo.\n");
        return 1;
    }

    /* Rellena el archivo con números enteros. */
    for (i=0; i<20; i++)
        fwrite(&i, sizeof(int), 1, fp);

    /* Ubica el puntero a registro en la décima posición. */
    fseek(fp, 9*sizeof(int), SEEK_SET);

    /* Escribe 250 en la décima posición. */
    fwrite(&num, sizeof(int), 1, fp);

    /* Ubica el puntero a registro en la primera posición. */
    fseek(fp, 0, SEEK_SET);

    /* Imprime de manera secuencial todo el archivo. */
    while ( !feof(fp) )
    {
        fread(&num, sizeof(int), 1, fp);
        printf("%d ", num);
    }
}
```

```

        printf("%d  ", num);
    }
    printf("\n");

    /* Cierra el archivo. */
    fclose(fp);

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap07_ej03;

TYPE t_arch = FILE OF INTEGER;

VAR fp: t_arch;
    num, i: INTEGER;

{ Programa Principal }
BEGIN

ASSIGN(fp,'numeros.dat');

{$I-}
REWRITE(fp);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo.');
    EXIT;
END;

{ Rellena el archivo con números enteros. }
FOR i:=0 TO 20-1 DO
    WRITE(fp, i);

{ Ubica el puntero a registro en la décima posición. }
SEEK(fp, 9);

{ Escribe 250 en la décima posición. }
WRITE(fp, 250);

{ Ubica el puntero a registro en la primera posición. }
SEEK(fp, 0);

{ Imprime secuencialmente todo el archivo. }
WHILE (NOT EOF(fp)) DO
BEGIN

```

```
    READ(fp, num);
    WRITE(num, ' ');
END;
WRITELN;

{ Cierra el archivo. }
CLOSE(fp);

END.
```

4- Cree un archivo binario, cuyos registros contengan los siguientes campos:

- Código (entero).
- Descripción (cadena de caracteres).
- Precio unitario (real).

El programa debe permitir la carga de datos en el archivo. Luego, leer y mostrar el último registro válido del archivo (anterior al EOF). Con ulterioridad se debe posicionar sobre la marca de fin de archivo (EOF) y permitir agregar datos nuevos al final. Por último, mostrar el contenido del archivo completo.

Solución en lenguaje C

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

#define N 100

typedef struct {
    int    codigo;
    char   descripcion[50];
    float  precio_unit;
} t_producto;

/* Prototipos */
t_producto cargar_producto();
void imprimir_producto(t_producto);
int main()
{
    FILE *stock;
    t_producto buffer;
    int i;

    if ((stock=fopen("stock.dat","wb+")) == NULL)
    {
        printf("No se puede abrir el archivo.\n");
        return 1;
    }
```

```
/* Carga de datos en el archivo */
for (i=0; i<N; i++)
{
    buffer = cargar_producto();
    fwrite(&buffer, sizeof(t_producto), 1, stock);
}

/* Lectura del último registro (registro N-1) */
fseek(stock, (N-1)*sizeof(t_producto), SEEK_SET);
fread(&buffer, sizeof(t_producto), 1, stock);
imprimir_producto(buffer);

/* Posicionamiento sobre EOF (para agregar un registro al final) */
fseek(stock, 0, SEEK_END);
buffer = cargar_producto();
fwrite(&buffer, sizeof(t_producto), 1, stock);

/* Listado secuencial del archivo */
rewind(stock);
while (!feof(stock))
{
    fread(&buffer, sizeof(t_producto), 1, stock);
    imprimir_producto(buffer);
}
fclose(stock);

return 0;
}

t_producto cargar_producto()
{
    t_producto buffer;

    printf("DATOS DEL PRODUCTO\n");
    printf(" Código: ");
    scanf("%d", &(buffer.codigo));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf(" Descripción: ");
    fgets(buffer.descripcion, 50, stdin);
    if (buffer.descripcion[strlen(buffer.descripcion)-1] == '\n')
        buffer.descripcion[strlen(buffer.descripcion)-1] = '\0';
    else
        while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf(" Precio unitario: ");
    scanf("%f", &(buffer.precio_unit));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

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

```
    return buffer;
}

void imprimir_producto(t_producto producto)
{
    printf("DATOS DEL PRODUCTO\n");
    printf(" Código: %d\n", producto.codigo);
    printf(" Descripción: %s\n", producto.descripcion);
    printf(" Precio unitario: %f\n", producto.precio_unit);
    printf("\n");
}
```

Solución en lenguaje Pascal

```
PROGRAM cap07_ej04;

CONST N = 100;

TYPE t_producto = RECORD
    codigo: INTEGER;
    descripcion: STRING[50];
    precio_unit: REAL;
END;

t_arch = FILE OF t_producto;

FUNCTION cargar_producto: t_producto;
VAR buffer: t_producto;

BEGIN
    WRITELN('DATOS DEL PRODUCTO');
    WRITE(' Código: ');
    READLN(buffer.codigo);
    WRITE(' Descripción: ');
    READLN(buffer.descripcion);
    WRITE(' Precio unitario: ');
    READLN(buffer.precio_unit);
    WRITELN;

    cargar_producto := buffer;
END;

PROCEDURE imprimir_producto(producto: t_producto);

BEGIN
    WRITELN('DATOS DEL PRODUCTO');
    WRITELN(' Código: ', producto.codigo);
    WRITELN(' Descripción: ', producto.descripcion);
    WRITELN(' Precio unitario: ', producto.precio_unit:6:2);

```

```

WRITELN;
END;

VAR stock: t_arch;
    buffer: t_producto;
    i: INTEGER;

{ Programa Principal }
BEGIN

ASSIGN(stock,'stock.dat');

{$I-}
REWRITE(stock);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo.');
    EXIT;
END;

{ Carga de datos en el archivo }
FOR i:=1 TO N DO
BEGIN
    buffer := cargar_producto();
    WRITE(stock, buffer);
END;

{ Lectura del último registro (registro N-1) }
SEEK(stock, FILESIZE(stock)-1);
READ(stock, buffer);
imprimir_producto(buffer);

{ Posicionamiento sobre EOF (para agregar un registro al final) }
SEEK(stock, FILESIZE(stock));
buffer := cargar_producto();
WRITE(stock, buffer);

{ Listado secuencial del archivo }
RESET(stock);
WHILE (NOT EOF(stock)) DO
BEGIN
    READ(stock, buffer);
    imprimir_producto(buffer);
END;

CLOSE(stock);

END.

```

5- Tome un archivo de texto, haga que el programa lo encripte en otro archivo y grabe cada carácter como el código ASCII que lo representa. Los nombres de los archivos de entrada y salida se deben recibir como primer y segundo argumento por línea de comandos, respectivamente.

Solución en lenguaje C

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *in, *out;
    char car;

    if (argc != 3)
    {
        printf("Modo de uso: %s <arch entrada> <arch salida>\n", argv[0]);
        return 1;
    }

    if ((in=fopen(argv[1],"r")) == NULL)
    {
        printf("No se puede abrir el archivo de entrada.\n");
        return 1;
    }

    if ((out=fopen(argv[2], "w")) == NULL)
    {
        printf("No se puede crear el archivo de salida.\n");
        fclose(in);
        return 1;
    }

    rewind(in);

    while (!feof(in))
    {
        car = getc(in);
        fprintf(out, "%d", car);
    }

    fclose(in);
    fclose(out);

    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM cap07_ej05;
TYPE t_arch = TEXT;
```

```

VAR in_file, out_file: t_arch;
    car: CHAR;

{ Programa Principal }
BEGIN

    IF (PARAMCOUNT <> 2) THEN
        BEGIN
            WRITELN('Modo de uso: ', PARAMSTR(0), ' <arch entrada> <arch salida>');
            EXIT;
        END;

ASSIGN(in_file,PARAMSTR(1));
ASSIGN(out_file,PARAMSTR(2));

{$I-}
RESET(in_file);
{$I+}
IF (IORESULT <> 0) THEN
    BEGIN
        WRITELN('No se puede abrir el archivo de entrada.');
        EXIT;
    END;

{$I-}
REWRITE(out_file);
{$I+}
IF (IORESULT <> 0) THEN
    BEGIN
        WRITELN('No se puede crear el archivo de salida.');
        EXIT;
    END;

WHILE (NOT EOF(in_file)) DO
    BEGIN
        READ(in_file, car);
        { ORD() retorna el valor ordinal (ASCII) del carácter. }
        WRITE(out_file, ORD(car));
    END;

CLOSE(in_file);
CLOSE(out_file);

END.

```

- 6- En un depósito de mercaderías se utiliza un archivo (MERC.DAT) para almacenar información referida a cada producto. Al final del día se debe actualizar ese archivo para dejar constancia de las operaciones realizadas en cada jornada. Esta información se halla en un archivo llamado NOVE.DAT.

Cada registro del archivo MERC.DAT contiene los campos siguientes:

- Código de producto.
- Descripción.
- Unidades en existencia.

Para los registros del archivo NOVE.DAT los campos son:

- Código de producto.
- Cantidad vendida o comprada (negativo significa venta, y positivo, compra).

Este programa actualiza el archivo MERC.DAT a partir del archivo NOVE.DAT, y devuelve el resultado en el archivo NUEVO.DAT. Todos los archivos están ordenados de manera ascendente por código de producto.

Aclaración: no se proveen los archivos MERC.DAT y NOVE.DAT (debido a que son binarios). Deberán ser generados antes de la ejecución de este programa.

Solución en lenguaje C

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

typedef struct {
    int     codigo;
    char   descripcion[30];
    int     existencia;
} t_producto;

typedef struct {
    int     codigo;
    int     cant_transaccion;
} t_novedad;

typedef enum {FALSO, VERDADERO} t_bool;

void apareo(FILE *, FILE *, FILE *);

int main()
{
    FILE *fmerc, *fnov, *fnuevo;

    if ((fmerc=fopen("merc.dat","rb")) == NULL)
    {
        printf("No se puede abrir el archivo fuente.\n");
        return 1;
    }

    if ((fnov=fopen("nove.dat","rb")) == NULL)
    {
        printf("No se puede abrir el archivo de novedades.\n");
        fclose(fmerc);
        return 1;
    }
```

```
if ((fnuevo=fopen("nuevo.dat","wb")) == NULL)
{
    printf("No se puede abrir el archivo destino.\n");
    fclose(fmerc);
    fclose(fnov);
    return 1;
}

apareo(fmerc, fnov, fnuevo);

fclose(fmerc);
fclose(fnov);
fclose(fnuevo);

return 0;
}

void apareo(FILE* fmerc, FILE* fnov, FILE* fnuevo)
{
    t_producto producto;
    t_novedad novedad;

    rewind(fmerc);

    while (!feof(fmerc))
    {
        fread(&producto, sizeof(t_producto), 1, fmerc);

        rewind(fnov);
        while (!feof(fnov))
        {
            fread(&novedad, sizeof(t_novedad), 1, fnov);

            if (producto.codigo == novedad.codigo)
            {
                /* Actualizar */
                producto.existencia += novedad.cant_transaccion;
            }
        }

        if (fwrite(&producto, sizeof(t_producto), 1, fnuevo) != 1)
        {
            printf("Error de escritura de los datos!");
            return;
        }
    }
}
```

Solución en lenguaje Pascal

```
PROGRAM cap07_ej06;

TYPE t_producto = RECORD
    codigo:          INTEGER;
    descripcion:    STRING[30];
    existencia:     INTEGER;
  END;

t_novedad = RECORD
    codigo:          INTEGER;
    cant_transaccion: INTEGER;
  END;

t_arch_productos = FILE OF t_producto;
t_arch_novedades = FILE OF t_novedad;

PROCEDURE apareo(VAR fmerc: t_arch_productos; VAR fnov: t_arch_novedades; VAR fnuevo: t_arch_productos);

VAR producto: t_producto;
    novedad: t_novedad;

BEGIN
  RESET(fmerc);

  WHILE (NOT EOF(fmerc)) DO
    BEGIN
      READ(fmerc, producto);

      RESET(fnov);

      WHILE (NOT EOF(fnov)) DO
        BEGIN
          READ(fnov, novedad);

          IF (producto.codigo = novedad.codigo) THEN
            BEGIN
              { Actualizar }
              producto.existencia := producto.existencia +
novedad.cant_transaccion;
            END;
        END;

        {$I-}
        WRITE(fnuevo, producto);
        {$I+}

        IF (IORESULT <> 0) THEN
          BEGIN
            WRITELN('Error de escritura de los datos!');
          END;
    END;
  END;
END;
```

```

        EXIT;
    END;

    END;

END;

VAR fmerc, fnuevo: t_arch_productos;
fnov: t_arch_novedades;

{ Programa Principal }

BEGIN

ASSIGN(fmerc,'merc.dat');
ASSIGN(fnov,'nove.dat');
ASSIGN(fnuevo,'nuevo.dat');

{$I-}
RESET(fmerc);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo fuente.');
    EXIT;
END;

{$I-}
RESET(fnov);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo de novedades.');
    CLOSE(fmerc);
    EXIT;
END;

{$I-}
RESET(fnuevo);
{$I+}
IF (IORESULT <> 0) THEN
BEGIN
    WRITELN('No se puede abrir el archivo destino.');
    CLOSE(fmerc);
    CLOSE(fnov);
    EXIT;
END;

apareo(fmerc, fnov, fnuevo);

CLOSE(fmerc);
CLOSE(fnov);
CLOSE(fnuevo);

END.

```

7.9 Contenido de la página Web de apoyo.



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Lecturas adicionales:

Archivos (versión preliminar de 20 páginas), de Martín Silva, será parte del libro *Sistemas Operativos* de Alfaomega Grupo Editor. Agradecemos a su autor por permitir que su escrito sea parte de las lecturas complementarias de esta obra.



Autoevaluación.



Video explicativo (04:21 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas. *



Presentaciones. *

8

Claves e índices

Contenido

8.1 Introducción	216
8.2 Claves.....	216
8.3 Índices	218
8.4 Índices y archivos.....	219
8.5 Resumen.....	221
8.6 Problemas propuestos.....	222
8.7 Problemas resueltos.....	223
8.8 Contenido de la página Web de apoyo.....	245

Objetivos

- Desarrollar el concepto de clave a los fines de identificar un dato dentro de un medio de almacenamiento masivo.
- Introducir el concepto de índice, su creación y tratamiento, a los fines del acceso a datos.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.



Nos referimos a objeto como entidad o cosa (un alumno, un empleado, un producto). No confundir con el concepto objeto del paradigma de programación orientado a objetos (que excede los contenidos de este libro).

8.1 Introducción.

En la vida cotidiana, cuando necesitamos referirnos a una persona, lo hacemos por su nombre o apellido, con frases como: "¡hola Juan!", "señor Martínez, adelante por favor", "Cristina, pase al pizarrón". De manera similar, y en otros contextos, muchas veces debemos identificarnos con el número de documento, un número de legajo, un código. Éstos son algunos ejemplos de los "mecanismos" que existen y utilizamos para identificarnos, para que cada uno pueda distinguirse o reconocerse dentro del conjunto de personas con las que convivimos.

La identificación de objetos aplica a diferentes escenarios, no sólo para distinguir a una persona en una población sino, en general, para representar individuos o elementos dentro de un conjunto (cada producto dentro de un stock de mercaderías, cada registro en un archivo, etc.).

A la porción de información que empleamos para referirnos a un "objeto" se la denomina clave. Así, el nombre y el apellido de una persona, un número de documento, un código de producto, la placa o la matrícula de un automóvil son algunos ejemplos de claves que usamos en forma habitual. Claro que no todas ellas permiten una identificación única del objeto; por ejemplo, en un aula dos o más estudiantes podrían llamarse "Cristina", tal que cuando se ordena "Cristina, pase al pizarrón" es preciso brindar información adicional para desambiguar esa situación. En la primera parte de este capítulo se tratarán los diferentes tipos de claves y sus usos.

Continuando con ejemplos cotidianos, utilicemos este libro que tenemos en nuestras manos, que de hecho tiene muchas páginas. Si necesitamos buscar un tópico determinado, sería esperable que hagamos uso del índice, ya que se trata de un mecanismo simple para acceder de manera directa a la página que contiene el tema en cuestión.

Cuando se trabaja con cantidades importantes de datos, como podría ser la manipulación de archivos grandes, puede ser una estrategia muy interesante construir un mecanismo similar al índice de un libro, tal que brindando cierta información relacionada con un registro, podemos acceder a ese registro en forma directa, sin necesidad de buscar de manera secuencial entre todos los datos. El uso de índices en ese contexto es una técnica explorada y explotada desde los albores de la informática y que en el presente encuentra su campo más prolífico en el mundo de las bases de datos. En la segunda mitad de este capítulo se introducirá el concepto de índice y se tratarán sus nociones fundamentales.

8.2 Claves.

Una clave es una porción de información perteneciente a un objeto que permite identificarlo.

Para acceder a un elemento (dato) es necesario contar con información (clave) que represente en forma única a ese elemento.

Como se indicó en la introducción de este capítulo, una clave es una porción de información perteneciente a un objeto que permite identificarlo. Esta "identificación" puede ser, en algunos casos, única; esto es, la clave representa un objeto y sólo ese objeto (dos o más de ellos no pueden compartir la misma clave). Por ejemplo, dado un arreglo llamado `foo`, podríamos acceder a la décima posición del mismo para leer el dato alojado allí mediante la sentencia `foo[9]` (asumiendo que el arreglo comienza en la posición 0); en este caso, la posición del elemento lo identifica de forma única y podría, en principio, considerarse una clave potencial. De manera análoga, en un archivo directo podemos posicionarnos en un registro en particular (utilizando funciones como `fseek()` o `seek()`), pero sólo en un registro.

De esta manera, la primera conclusión que podemos extraer es que para acceder a un elemento (dato) en particular es necesario contar con información (clave) que represente en forma única a ese elemento. Se indica que la relación entre el dato y su clave es uno a uno.

Una clave única es para un dato lo que el número de documento es para las personas en el contexto de la población de un país. En el ámbito de una universidad, una buena clave es el número de padrón, registro o legajo de cada alumno. En el seno de una familia, el nombre de

cada integrante podría ser útil para distinguir a sus miembros (siempre que no haya dos personas con igual nombre). De esta manera puede observarse que la eficacia de una clave para convertirse en unívoca depende del contexto donde se la aplique (el nombre de una persona podría distinguirla de otros individuos dentro de su familia, pero es probable que no lo haga dentro de la población de su país).

Las claves no siempre son datos atómicos. O sea que una clave podría estar conformada por varios atributos, en cuyo caso se la denomina clave compuesta. En una base de datos de comercio electrónico podrían almacenarse los datos de clientes de diferentes partes del mundo. En un escenario como ése, utilizar el número de documento de la persona para identificarla de manera unívoca no sería correcto, ya que nadie puede asegurar que individuos de diferentes países no puedan tener igual número de documento. En ese caso, la desambiguación se logra construyendo una clave compuesta por nacionalidad y número de documento (ya que, si de algo podemos estar seguros, es que el número de documento no se repetirá para una misma nacionalidad).

En el ámbito de las bases de datos, la información se organiza en tablas; cada una de ellas con una estructura similar a la de un arreglo de registros (ya desarrollado en el capítulo 6). Supongamos una tabla en una base de datos, con información sobre estudiantes universitarios, como se muestra a continuación:

Tabla 8-1 - Tabla en una base de datos, con información sobre estudiantes universitarios.

# Legajo	Nombre y apellido	# Documento de identidad	Dirección	Fecha de nacimiento
76 054	Rinaldi, Jorge	25 982 901	Colombres 1243, 4° "D"	11/10/1983
76 055	Frías, Andrea	26 562 777	Av. Gaona 49, 6° "H"	03/05/1981
76 056	Rodríguez, Pablo	24 872 662	Carlos Calvo 345, 1° "A"	28/06/1983
76 057	Rinaldi, María	27 021 392	Colombres 1243, 4° "D"	14/10/1985
...

A partir de este conjunto de datos es nuestra responsabilidad (como programadores o administradores de la base) elegir uno o más campos que identifiquen de manera única cada registro de la tabla. En primera instancia podríamos optar por el número de legajo, ya que cada uno de ellos corresponde a un único alumno, y un alumno no puede poseer más de uno (relación uno a uno). Otra posibilidad es la adopción del número de documento de identidad, que también es capaz de identificar en forma unívoca a cada alumno. En cambio, no sería correcto utilizar el nombre y el apellido por la simple razón de que puede haber dos alumnos o más llamados de igual forma. Este mismo argumento se aplica para la dirección y la fecha de nacimiento, que tampoco pueden seleccionarse como claves unívocas. Sin embargo, en principio podría construirse una clave compuesta con los campos nombre y apellido, dirección y fecha de nacimiento (asumiendo que en un mismo domicilio no viven dos personas con igual nombre y apellido nacidas el mismo día, argumento que es suficientemente débil como para optar por esta combinación de datos como clave).

El número de legajo es un dato atómico (posiblemente implementado como un entero largo), a diferencia de la clave nombre y apellido + dirección + fecha de nacimiento. Además, en la tabla puede verse que los registros se encuentran ordenados en forma ascendente por número de legajo, lo que facilitaría el acceso a un registro en particular utilizando, por ejemplo, una búsqueda binaria.

No hay una “bala de plata” a la hora de buscar una clave, pero existen “buenas prácticas”. Para establecer la clave, deben analizarse los datos con los que se trabaja y considerar, por ejemplo, por medio de qué campo se accederá a los registros de manera habitual (por ejemplo, para la tabla de alumnos mostrada antes, el acceso será más frecuente por número de legajo, con lo que este campo sería más efectivo como clave respecto del número de documento de identidad).



Tipos de claves:

1. Clave candidata.
2. Clave primaria.
3. Clave secundaria.

Hay diferentes tipos de claves. Nombraremos los principales:

Clave candidata: Identificador único (simple o compuesto) que, si se le quita alguno de sus campos, deja de ser clave. Por esta razón, se la considera una clave mínima.

En el proceso de selección de una clave única para un conjunto de datos, se parte por reconocer las claves candidatas. De entre todas ellas se escogerá la más adecuada.

Clave primaria: Es la candidata que seleccionamos para representar los registros de un conjunto de datos. Esa selección debe hacerse bajo ciertos criterios, como podrían ser:

- La clave por la que se accederá a los registros con mayor frecuencia.
- La clave con menor cantidad de campos (en el mejor caso, un dato atómico, como el número de legajo).
- La clave por la que los registros mantengan un orden.

Clave secundaria: Esta es muy particular, en el sentido de que no es única; dos registros o más en la tabla podrían tener igual clave secundaria. La utilidad de este tipo de claves radica en que permiten “agrupar” datos, algo que en la práctica se requiere con frecuencia. Por ejemplo, cada día el sistema de bases de datos podría consultar cuáles son los alumnos nacidos ese día, con el fin de enviarles un mensaje de salutación (en este caso, la fecha de nacimiento se emplearía como clave secundaria). Es importante entender que, más allá de la cantidad de claves secundarias (si las hay), de todas maneras debe haber clave primaria.

En esta primera parte del capítulo se ofrece una introducción muy elemental al concepto de clave. El tema es por demás importante, ya que de la elección de una buena clave depende el funcionamiento futuro de la base de datos.



Encuentre un simulador sobre organización de un archivo por un índice determinado en la Web de apoyo.

8.3 Índices.

Para comprender el concepto de índice podemos partir de un ejemplo simple de la vida diaria. Supongamos que debemos prepararnos para un examen de física elemental, y uno de los tópicos en cuestión es “las leyes de Newton”. Para ese fin conseguimos un manual de física y mecánica clásica, que tiene unas 1 300 páginas. ¿Cómo encontramos en un libro de ese tamaño el tema “leyes de Newton”? Es probable que el lector utilice el índice del manual para conocer el número de página donde encontrar el tema. En cambio, si no contáramos con el índice, deberíamos buscar de manera secuencial página por página, técnica poco eficiente para libros de gran volumen.

Siempre que necesitemos consultar un libro en busca de algún tema en particular, el sentido común nos lleva a utilizar el índice que, en general, forma parte de las primeras o las últimas páginas (de manera que el lector pueda acceder a él con facilidad). En el esquema siguiente pueden apreciarse cuáles son los pasos a seguir para consultar un tema determinado en un libro:

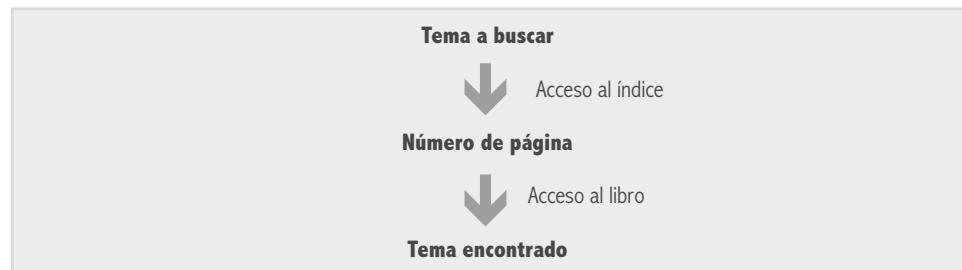


Fig. 8-1. Pasos que se deben seguir para consultar un tema determinado en un libro.

Hasta aquí podemos concluir que el índice de un libro permite acceder en forma directa a una página, sin necesidad de recorrerlo en forma secuencial página por página.

Supongamos que el manual de física y mecánica clásica pertenece a la biblioteca de la universidad. Cuando vamos a solicitarlo, la persona encargada de la biblioteca accederá a una carpeta (o sistema informático) donde están registradas todas las obras disponibles. En principio, todas ellas podrían estar ordenadas por su título y acompañadas con la información del número de estante correspondiente. Así, utilizando el título del libro en cuestión, sabríamos en qué estante podemos encontrarlo. Sin embargo, éste podría no ser siempre el caso y, en lugar de proporcionar el título del libro, damos el nombre y el apellido de su autor. En ese caso la biblioteca dispondrá de otra carpeta donde las obras estén registradas, ordenadas por el nombre de su autor y acompañadas por el número de estante.

Cada una de estas “carpetas” también es un índice, que permite acceder en forma directa a un estante determinado de la biblioteca en busca de un libro. Así, también podemos concluir que un índice no sólo sirve para acceder en forma directa al elemento buscado, sino que también nos permite “ver” el conjunto de elementos ordenado según diferentes criterios (título de la obra, nombre de su autor, ISBN, etc.).

8.4 Índices y archivos.

De la misma manera que los índices permiten acceder a una página de un libro o a un estante en una biblioteca, también pueden emplearse para el acceso directo a una posición específica en un archivo de datos. Esto es útil en particular cuando se manipulan archivos grandes, ya que, como se indicó en el capítulo 7, la latencia en el acceso a un medio de almacenamiento externo suele ser muy elevada.

En el diagrama siguiente se ilustra la manera en que están organizados un archivo y su índice; en este ejemplo el índice se ordenó por el número de legajo de cada alumno, asumiendo que ésta es la clave primaria.

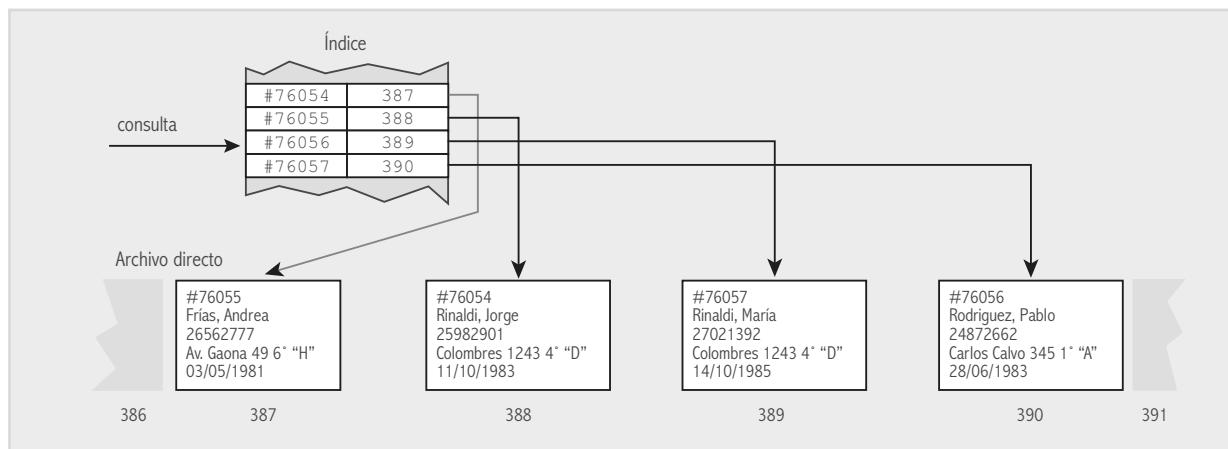


Fig. 8-2. Modo en que están organizados un archivo y su índice.

Como se puede apreciar en la figura, un índice es una estructura muy simple, donde cada entrada consiste en la clave de acceso y la posición en el archivo donde se aloja el registro buscado. Así, un índice puede construirse utilizando un arreglo lineal de registros.

El procedimiento de construcción de un índice se denomina indexación y a ese archivo se lo llama archivo indexado. En los casos típicos el índice suele construirse recorriendo en forma secuencial el



Un índice se utiliza para acceder en forma directa a un registro determinado en el archivo indexado (que debe ser de organización y acceso directo). Los archivos secuenciales no pueden ser indexados.

archivo en cuestión e insertando por cada registro una nueva entrada clave + posición. Es razonable realizar esta tarea sólo una vez, ya que los recorridos secuenciales de archivos son costosos.

Un índice se utiliza, básicamente, para acceder en forma directa a un registro determinado en el archivo indexado. Por esta razón, el archivo en cuestión debe ser de organización y acceso directo. Expresado de otra manera, los archivos secuenciales no pueden ser indexados.

Dado que el índice contiene una cantidad mínima de información (suficiente para acceder a cada registro) se intenta mantenerlo en memoria principal durante la ejecución de la aplicación. Así, cuando se busca un dato según una clave de acceso, esta búsqueda se lleva a cabo en memoria principal, sobre el índice, antes de acceder en forma directa al archivo. Si se toma en cuenta que en los casos típicos los tiempos de acceso a memoria principal son seis órdenes de magnitud menores respecto del acceso a memoria secundaria, es clara la ventaja en el uso de índices para realizar búsquedas.

Dado que el contenido de un archivo se altera con el agregado o la eliminación de registros, el índice que lo "indexa" también se afectará. En particular si se agrega un registro en el archivo, deberá introducirse una nueva entrada clave + posición en el índice. Por otro lado, si se elimina un registro del archivo, será necesario buscar y eliminar la entrada correspondiente.

El índice puede estar ordenado por clave de acceso. De esta manera, dada una clave de acceso, se puede aplicar una búsqueda binaria sobre el índice y recuperar la posición del registro. Nótese la ventaja de ejecutar una búsqueda binaria sobre un índice en memoria principal respecto de hacerlo directamente sobre el archivo. Sin embargo, podría optarse por mantener el índice sin orden particular alguno. En ese caso, dada una clave de acceso, puede emplearse una búsqueda secuencial sobre el índice, que siempre será más ventajosa que acceder al archivo.

No se debe olvidar que, como se comentó en el ejemplo de la "biblioteca", es posible construir más de un índice para acceder a un mismo conjunto de datos, por medio de diferentes claves de acceso. Para el archivo de estudiantes, además del índice por número de legajo (clave primaria), también podríamos contar con un índice por número de documento de identidad.

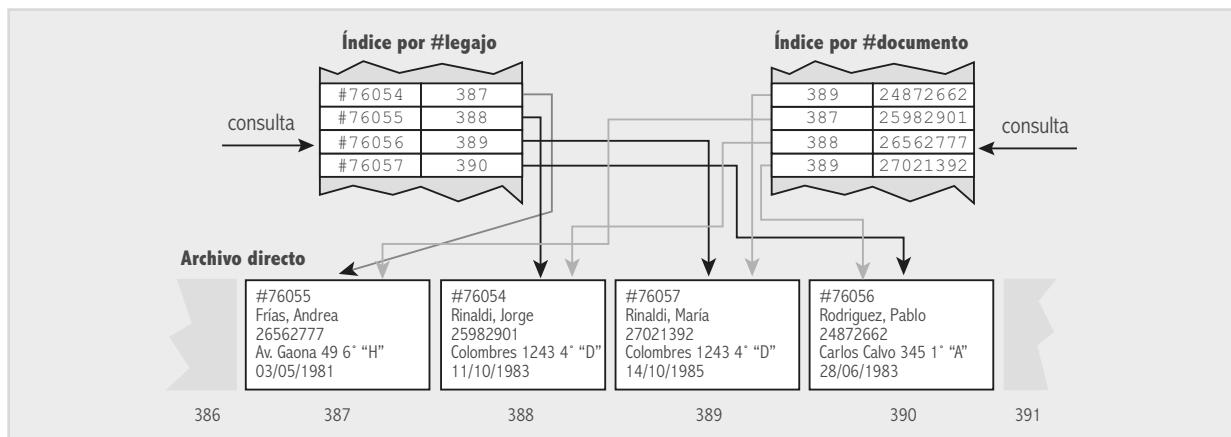


Fig. 8-3. Uso de más de un índice para acceder a un mismo conjunto de datos por medio de diferentes claves de acceso.

Debido a que en la práctica suelen construirse varios índices para acceder a archivos y que, además, estos índices pueden adquirir tamaños considerables, comienzan a surgir problemas relacionados con la forma de mantener los índices en memoria. Esto llevó al uso de técnicas que permiten, por ejemplo, mantener los índices en memoria de manera parcial (por ejemplo,

sólo las partes usadas más recientemente). Cuando se requiere acceder a una porción del índice que no está en memoria principal, entonces se la debe ir a buscar a memoria secundaria y cargarla en memoria principal, desalojando otra parte del índice. En esa situación el mundo “ideal” de los índices deja de ser tan ideal; sin embargo, para archivos grandes, la ventaja del indexado (aun en esas situaciones) sigue siendo indiscutible. Las técnicas empleadas para el tratamiento de índices que no caben en memoria no son alcance de este libro.

8.4.1 Índices primarios y secundarios.

De acuerdo con la clave de acceso indexada, los índices pueden clasificarse en:

- **Índices primarios:** Este tipo de índices tiene como característica que la clave por la que se indexa es primaria, o sea, unívoca. Debe quedar claro que en un índice primario cada entrada tiene una sola referencia al archivo de datos, ya que no puede haber dos registros o más con igual clave.
- **Índices secundarios:** En este caso se indexa por una clave secundaria. Esta última no necesariamente es unívoca y, por lo tanto, es posible que en el índice haya varias referencias para una misma clave.

Para un conjunto de datos en particular solo puede existir un índice primario, mientras que pueden convivir varios índices secundarios simultáneamente. Debe quedar claro que la clave primaria es única, por lo tanto el índice primario también lo es.

8.4.2 Eliminación y agregado de registros.

Es lógico que el archivo indexado cambie su morfología en la medida en que se eliminan o se añadan registros. De la misma manera, los índices que lo indexen también se verán alterados. Si se elimina un registro en el archivo, entonces también deberá eliminarse la entrada correspondiente en los índices. De manera análoga, cuando se agregue un nuevo registro en el archivo, deberá crearse una entrada en cada uno de los índices.

En el caso de eliminación de registros, esto puede hacerse a nivel físico (borrado físico), quitando el registro en cuestión del archivo. Sin embargo, cuando se utilizan índices, existe la posibilidad de hacer una eliminación a nivel lógico (borrado lógico), que consiste en quitar la entrada en el índice y modificar un campo booleano en el registro en cuestión que indique que el mismo ya no es válido (pero físicamente el registro no se elimina del archivo). Dado que el acceso al archivo se hace por medio del índice, es suficiente con borrar la entrada para indicar que ese registro ya no es válido. El borrado lógico tiene la ventaja de ser más rápido que el físico, ya que no requiere el acceso a memoria secundaria (sólo se modifica el índice en memoria principal). Sin embargo, requiere que en algún momento de la vida del programa (por ejemplo, cuando se sale de él) se borren físicamente todos los registros que antes fueron borrados en forma lógica. A esta tarea a menudo se la conoce como “limpieza del hogar” (*housekeeping*).

En el caso de agregado de registros, debe crearse una nueva entrada en cada uno de los índices y, además, éstos deben reordenarse.

8.5 Resumen

Al procesar conjuntos de datos podría ser necesario que todos los elementos que forman parte de ese conjunto puedan identificarse de manera unívoca. A la porción de información que empleamos para referirnos a un dato en particular dentro de un conjunto se la denomina clave y, como un ejemplo cotidiano, podemos indicar que el número de documento de cada persona dentro de un país es una clave unívoca.



Clasificación de índices:

1. Índices primarios.
2. Índices secundarios.

Sobre un conjunto de datos que pueden distinguirse en forma única mediante claves es posible construir índices. El concepto de índice en programación es análogo al de índice en un libro: Cuando se busca un tópico en particular, se accede al índice para obtener el número de página donde se desarrolla ese tema; luego, se puede acceder directamente a la página en cuestión, y evitar el recorrido secuencial por todo el libro. Por ejemplo, en la manipulación de archivos grandes podría ser muy conveniente construir un mecanismo similar al índice de un libro, tal que brindando cierta información relacionada con un registro (clave), podamos acceder a ese registro en forma directa, sin necesidad de buscar de manera secuencial entre todos los datos.

El procedimiento de construcción de un índice se denomina indexación y a ese archivo se lo llama archivo indexado. Dado que el índice contiene una cantidad mínima de información (suficiente para acceder a cada registro), se intenta mantenerlo en memoria principal durante la ejecución de la aplicación. Así, cuando se busca un dato según una clave de acceso, esta búsqueda se lleva a cabo en memoria principal, sobre el índice, antes de acceder directamente al archivo. Teniendo en cuenta que los tiempos de acceso a memoria principal son mucho menores respecto del acceso a memoria secundaria, es clara la ventaja en el uso de índices para realizar búsquedas.

Además, es posible construir más de un índice para acceder a un mismo conjunto de datos, por medio de diferentes claves de acceso. Todos estos índices pueden estar ordenados y, sin embargo, no se requiere que el conjunto de datos al que indexan lo esté. Ésta es otra de las características importantes de los índices: Permiten "ver" a un conjunto de datos como si estuviese ordenado según diferentes criterios.

8.6 Problemas propuestos.

- 1) Se tiene un archivo de registros, con el nombre: datos.dat; con la siguiente estructura:

```
Type
  Registro= Record
    logico:Boolean;
    Num : Array[3..6] of integer;
    SubReg: Record
      Valores: Array[1..3] of integer;
      Nombre : Array[1..10] of char
      End;
    Clave:Char
    End;
```

Desarrollar un programa que permita agregar registros al final del archivo datos.dat. Los nuevos registros deben ingresarse por teclado; se debe proveer un método para finalizar el ingreso de datos.

- 2) Se tienen dos archivos de registros de igual estructura. En cada archivo la secuencia de código está ordenada en forma ascendente. Los códigos presentes en un archivo no están presentes en el otro y viceversa.

La estructura del archivo es:

código : Entero;
nombre : String de hasta 16 caracteres;
valores: Arreglo de 10 enteros;

Diseñar un programa que genere un archivo con todos los registros de los dos archivos anteriores, y cuya secuencia de códigos esté ordenada de manera ascendente.

- 3) Generar una aplicación que realice la mezcla de archivos e implemente un índice sobre un campo dado.
- 4) Programar una aplicación que indexe un archivo con, por lo menos, dos índices.
- 5) Programar una aplicación que integre todos los métodos de ordenamiento e implemente un índice.
- 6) Programar una aplicación que aplique los métodos de búsqueda en arreglos indexados.
- 7) Generar una aplicación que realice la mezcla de archivos indexados.

8.7 Problemas resueltos.

1- Se cuenta con un archivo directo llamado ALUMNOS.DAT con información de alumnos de una universidad. El archivo se abre para lectura y se indexa por medio de su clave primaria (#legajo). El índice se mantiene en memoria principal.

Se ofrece la posibilidad al usuario de buscar alumnos por legajo. La búsqueda binaria se aplica sobre el índice y se accede en forma directa (`seek`) al archivo para recuperar el registro.

Se asume que el archivo está ordenado de manera ascendente por su clave primaria (#legajo). Si éste no fuese el caso, el índice debería ordenarse antes de aplicar una búsqueda binaria sobre él.

Aclaración: no se provee el archivo ALUMNOS.DAT (debido a que es binario). Deberá ser generado antes de la ejecución de este programa).

Solución en lenguaje C

```
#include <stdio.h>

#define TAMANIO_INDICE 5

typedef struct {
    int dia, mes, anio;
} t_fecha;

typedef struct {
    long legajo;
    char nomyap[30+1];
    char documento[10+1];
    char direccion[40+1];
    t_fecha fecha_nac;
} t_alumno;

/* Estructura de cada entrada en el índice */
typedef struct {
    long clave;          /* Clave de acceso (legajo) */
    long posicion;      /* Posición del registro en el archivo */
} t_entrada_indice;

typedef t_entrada_indice t_indice[TAMANIO_INDICE];

typedef enum {FALSO, VERDADERO} t_bool;

void indexar(FILE* arch, t_indice indice);
t_bool buscar(FILE* arch, t_indice indice, long legajo, t_alumno* alumno);
void mostrar_datos(t_alumno alumno);

int main()
{
    FILE* alumnos;
    t_indice indice;
    long legajo;
    t_alumno alumno;
    t_bool encontrado;
    int i;
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```
if ((alumnos=fopen("alumnos.dat","rb")) == NULL)
{
    printf("No se puede abrir el archivo.\n");
    return 1;
}

/* Indexación del archivo */
indexar(alumnos, indice);

do
{
    printf("Ingrese un legajo a buscar (-1 para terminar): ");
    scanf("%ld", &legajo);

    if (legajo != -1)
    {
        /* Búsqueda binaria sobre el índice */
        encontrado = buscar(alumnos, indice, legajo, &alumno);
        if (encontrado)
            mostrar_datos(alumno);
        else
            printf("El legajo %ld no se encuentra en el archivo.\n", legajo);
    }
    printf("\n");
} while (legajo != -1);

fclose(alumnos);

return 0;
}

void indexar(FILE* arch, t_indice indice)
{
    t_alumno alumno;
    int posicion, i = 0;

    rewind(arch);

    do
    {
        posicion = ftell(arch); /* Posición actual del puntero a registro */
        fread(&alumno, sizeof(t_alumno), 1, arch);
        if (!feof(arch))
        {
            indice[i].clave      = alumno.legajo;
            indice[i].posicion   = posicion;
            i++;
        }
    }
```

```
    } while (!feof(arch));
}

t_bool buscar(FILE* arch, t_indice indice, long legajo, t_alumno* alumno)
{
    long inferior, superior, centro;
    t_bool encontrado = FALSO;

    inferior = 0;
    superior = TAMANIO_INDICE-1;

    while ( (inferior <= superior) && !encontrado )
    {
        centro = (inferior + superior) / 2; /* División entera */

        if (indice[centro].clave == legajo)
        {
            encontrado = VERDADERO;
        }
        else
        {
            if (legajo < indice[centro].clave)
                superior = centro - 1; /* Buscar en la primera mitad */
            else
                inferior = centro + 1; /* Buscar en la segunda mitad */
        }
    }

    if (encontrado)
    {
        fseek(arch, indice[centro].posicion, SEEK_SET);
        fread(alumno, sizeof(t_alumno), 1, arch);
    }

    return encontrado;
}

void mostrar_datos(t_alumno alumno)
{
    printf("Legajo: %ld\n", alumno.legajo);
    printf("Nombre y apellido: %s\n", alumno.nomyap);
    printf("Documento de identidad: %s\n", alumno.documento);
    printf("Dirección: %s\n", alumno.direccion);
    printf("Fecha de nacimiento: %d/%d/%d\n", alumno.fecha_nac.dia, alumno.fecha_nac.mes, alumno.fecha_nac.anio);
}
```

Solución en lenguaje Pascal

```

PROGRAM cap08_ej01;

CONST TAMANIO_INDICE = 5;

TYPE t_fecha = RECORD
    dia, mes, anio: INTEGER;
END;

t_alumno = RECORD
    legajo: LONGINT;
    nomyap: STRING[30];
    documento: STRING[10];
    direccion: STRING[40];
    fecha_nac: t_fecha;
END;

{ Estructura de cada entrada en el indice }
t_entrada_indice = RECORD
    clave: LONGINT; { Clave de acceso (legajo) }
    posicion: LONGINT; { Posición del registro en el archivo }
END;

t_indice = ARRAY[1..TAMANIO_INDICE] OF t_entrada_indice;

t_arch_alumnos = FILE OF t_alumno;

PROCEDURE indexar(VAR arch: t_arch_alumnos; VAR indice: t_indice);

VAR alumno: t_alumno;
    posicion, i: INTEGER;

BEGIN
    i := 1;
    RESET(arch);

    WHILE (NOT EOF(arch)) DO
        BEGIN
            posicion := FILEPOS(arch);
            READ(arch, alumno);

            indice[i].clave := alumno.legajo;
            indice[i].posicion := posicion;
            INC(i);
        END;
    END;

    FUNCTION buscar(VAR arch: t_arch_alumnos; VAR indice: t_indice; legajo:

```

```
LONGINT; VAR alumno: t_alumno): BOOLEAN;

VAR inferior, superior, centro: LONGINT;
    encontrado: BOOLEAN;

BEGIN

    encontrado := FALSE;
    inferior := 1;
    superior := TAMANIO_INDICE;

    WHILE ( (inferior <= superior) AND (NOT encontrado) ) DO
        BEGIN
            centro := (inferior + superior) DIV 2; { División entera }

            IF (indice[centro].clave = legajo) THEN
                BEGIN
                    encontrado := TRUE;
                END
            ELSE
                BEGIN
                    IF (legajo < indice[centro].clave) THEN
                        superior := centro - 1 {Buscar en la primera mitad}
                    ELSE
                        inferior := centro + 1; { Buscar en la segunda mitad }
                END;
        END;

        IF (encontrado) THEN
            BEGIN
                SEEK(arch, indice[centro].posicion);
                READ(arch, alumno);
            END;

        buscar := encontrado;
    END;
END;

PROCEDURE mostrar_datos(alumno: t_alumno);

BEGIN
    WRITELN('Legajo: ', alumno.legajo);
    WRITELN('Nombre y apellido: ', alumno.nomyap);
    WRITELN('Documento de identidad: ', alumno.documento);
    WRITELN('Dirección: ', alumno.direccion);
    WRITELN('Fecha de nacimiento: ', alumno.fecha_nac.dia, '/',
            alumno.fecha_nac.mes, '/',
            alumno.fecha_nac.anio);
END;

VAR alumnos: t_arch_alumnos;
    indice: t_indice;
```

```

legajo:      LONGINT;
alumno:      t_alumno;
encontrado:  BOOLEAN;
i:           INTEGER;

{ Programa Principal }
BEGIN

ASSIGN(alumnos, 'alumnos_pas.dat');
RESET(alumnos);

{ Indexación del archivo }
indexar(alumnos, indice);

REPEAT
  WRITE('Ingrese un legajo a buscar (-1 para terminar): ');
  READLN(legajo);

  IF (legajo <> -1) THEN
    BEGIN
      { Búsqueda binaria sobre el índice }
      encontrado := buscar(alumnos, indice, legajo, alumno);
      IF (encontrado) THEN
        mostrar_datos(alumno)
      ELSE
        WRITELN('El legajo ', legajo, ' no se encuentra en el archivo.');
    END;
    WRITELN;
  UNTIL (legajo = -1);

CLOSE(alumnos);

END.

```

2- Se cuenta con el mismo archivo ALUMNOS.DAT del caso anterior. Éste se abre para lectura y se indexa mediante su clave primaria (#legajo). El índice se mantiene en memoria principal.

Se ofrece la posibilidad al usuario de añadir o eliminar registros del archivo, lo que impacta en el tamaño del índice. Al final se muestra el contenido del índice.

No se asume orden particular alguno en el archivo. Por lo tanto, luego de la indexación, el índice debe ordenarse.

Solución en lenguaje C

```

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

#define MAX_TAMANIO_INDICE 100

```

```
typedef struct {
    int dia, mes, anio;
} t_fecha;

typedef struct {
    long legajo;
    char nomayap[30+1];
    char documento[10+1];
    char direccion[40+1];
    t_fecha fecha_nac;
} t_alumno;

/* Estructura de cada entrada en el índice */
typedef struct {
    long clave;      /* Clave de acceso (legajo) */
    long posicion;   /* Posición del registro en el archivo */
} t_entrada_indice;

typedef t_entrada_indice t_indice[MAX_TAMANIO_INDICE];

typedef enum {FALSO, VERDADERO} t_bool;

void indexar(FILE* arch, t_indice indice, long* tamano_indice);
void cargar_alumno(FILE* arch, t_indice indice, long* tamano_indice);
void eliminar_alumno(FILE* arch, t_indice indice, long* tamano_indice);
void mostrar_indice(t_indice indice, long tamano_indice);
void ordenar_indice(t_indice indice, long tamano_indice);
long buscar(long legajo, t_indice indice, long tamano_indice);
int main()
{
    FILE* alumnos;
    t_indice indice;
    long tamano_indice;
    int opcion;

    if ((alumnos=fopen("alumnos.dat","r+b")) == NULL)
    {
        printf("No se puede abrir el archivo.\n");
        return 1;
    }

    /* Indexación del archivo */
    indexar(alumnos, indice, &tamano_indice);

    do
    {
        printf("1 -> Cargar nuevo alumno\n");
        printf("2 -> Eliminar alumno existente\n");
        printf("3 -> Salir\n");
    }
```

```
scanf("%d", &opcion);

switch (opcion)
{
    case 1: cargar_alumno(alumnos, indice, &tamanio_indice);
              break;
    case 2: eliminar_alumno(alumnos, indice, &tamanio_indice);
              break;
}
} while (opcion != 3);

mostrar_indice(indice, tamanio_indice);

fclose(alumnos);

return 0;
}

void indexar(FILE* arch, t_indice indice, long* tamanio_indice)
{
    t_alumno alumno;
    int posicion, i = 0;

    rewind(arch);

    do
    {
        posicion = ftell(arch); /* Posición actual del puntero a registro */
        fread(&alumno, sizeof(t_alumno), 1, arch);

        if (!feof(arch))
        {
            indice[i].clave      = alumno.legajo;
            indice[i].posicion   = posicion;
            i++;
        }
    } while (!feof(arch));

    *tamanio_indice = i;

    /* Reordenar indice */
    ordenar_indice(indice, *tamanio_indice);
}

void cargar_alumno(FILE* arch, t_indice indice, long* tamanio_indice)
{
    t_alumno alumno;
    long posicion;
```

```
int i;

printf("Legajo: ");
scanf("%ld", &(alumno.legajo));
while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Nombre y apellido: ");
fgets(alumno.nomyap, 31, stdin);
if (alumno.nomyap[strlen(alumno.nomyap)-1] == '\n')
    alumno.nomyap[strlen(alumno.nomyap)-1] = '\0';
else
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Número de documento: ");
fgets(alumno.documento, 11, stdin);
if (alumno.documento[strlen(alumno.documento)-1] == '\n')
    alumno.documento[strlen(alumno.documento)-1] = '\0';
else
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

printf("Dirección: ");
fgets(alumno.direccion, 41, stdin);
if (alumno.direccion[strlen(alumno.direccion)-1] == '\n')
    alumno.direccion[strlen(alumno.direccion)-1] = '\0';
else
    while (getchar() != '\n'); /* Limpieza buffer de teclado */
printf("Fecha de nacimiento:\n");
printf("\tDía: ");
scanf("%d", &(alumno.fecha_nac.dia));
while (getchar() != '\n'); /* Limpieza buffer de teclado */
printf("\tMes: ");
scanf("%d", &(alumno.fecha_nac.mes));
while (getchar() != '\n'); /* Limpieza buffer de teclado */
printf("\tAño: ");
scanf("%d", &(alumno.fecha_nac.anio));
while (getchar() != '\n'); /* Limpieza buffer de teclado */

fseek(arch, 0, SEEK_END); /* Posicionamiento al final, para agregado */
posicion = ftell(arch);
fwrite(&alumno, sizeof(t_alumno), 1, arch);

/* Nueva entrada en el índice */
i = *tamanio_indice;
indice[i].clave = alumno.legajo;
indice[i].posicion = posicion;
*tamanio_indice = *tamanio_indice + 1;

/* Reordenar índice */
ordenar_indice(indice, *tamanio_indice);
```

```

}

void eliminar_alumno(FILE* arch, t_indice indice, long* tamanio_indice)
{
    long legajo;
    int i;

    printf("Legajo a eliminar: ");
    scanf("%ld", &legajo);
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    i = buscar(legajo, indice, *tamanio_indice);
    if (i == -1)
    {
        printf("El legajo %ld no fue encontrado.\n", legajo);
    }
    else
    {
        /* Para borrar la entrada en el índice, es suficiente con asignar */
        /* el mayor valor posible para la clave y decrementar el tamaño del */
        /* índice. De esta manera, cuando se reordene el índice, ese */
        /* registro se moverá al final y no se tendrá en cuenta en futuros */
        /* accesos. */
        indice[i].clave = LONG_MAX;
        ordenar_indice(indice, *tamanio_indice);
        *tamanio_indice = *tamanio_indice - 1;
    }
}

void mostrar_indice(t_indice indice, long tamanio_indice)
{
    int i;

    for (i=0; i<tamanio_indice; i++)
    {
        printf("%ld -> %ld\n", indice[i].clave, indice[i].posicion);
    }
    printf("\n");
}

void ordenar_indice(t_indice indice, long tamanio_indice)
{
    int i, j;
    t_entrada_indice aux;

    for (i = tamanio_indice-1; i > 0; i--)
        for (j = 1; j <= i; j++)
            if (indice[j-1].clave > indice[j].clave)
            {

```

```

    /* Intercambio */
    aux = indice[j];
    indice[j] = indice[j-1];
    indice[j-1] = aux;
}
}

long buscar(long legajo, t_indice indice, long tamano_indice)
{
    long inferior, superior, centro;

    inferior = 0;
    superior = tamano_indice-1;

    while ( inferior <= superior )
    {
        centro = (inferior + superior) / 2; /* División entera */

        if (indice[centro].clave == legajo)
            return centro;
        else
            if (legajo < indice[centro].clave)
                superior = centro - 1; /* Buscar en la primera mitad */
            else
                inferior = centro + 1; /* Buscar en la segunda mitad */
    }

    return -1;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap08_ej02;

CONST MAX_TAMANIO_INDICE = 100;
      LONG_MAX = 2147483647;

TYPE t_fecha = RECORD
            dia, mes, anio: INTEGER;
          END;

t_alumno = RECORD
            legajo:     LONGINT;
            nomyap:    STRING[30];
            documento: STRING[10];
            direccion: STRING[40];
            fecha_nac: t_fecha;
          END;

```

```

{ Estructura de cada entrada en el índice }
t_entrada_indice = RECORD
    clave: LONGINT; { Clave de acceso (legajo) }
    posicion: LONGINT; { Posición del registro en el archivo }
END;

t_indice = ARRAY[1..MAX_TAMANIO_INDICE] OF t_entrada_indice;
t_arch_alumnos = FILE OF t_alumno;

PROCEDURE ordenar_indice(VAR indice: t_indice; tamano_indice: LONGINT);

VAR i,j: LONGINT;
    aux: t_entrada_indice;
BEGIN
    FOR i:=tamano_indice DOWNTO 2 DO
        FOR j:=2 TO i DO
            IF (indice[j-1].clave > indice[j].clave) THEN
                BEGIN
                    { Intercambio }
                    aux := indice[j];
                    indice[j] := indice[j-1];
                    indice[j-1] := aux;
                END
            END;
    END;

FUNCTION buscar(legajo: LONGINT; indice: t_indice; tamano_indice: LONGINT):
LONGINT;

VAR inferior, superior, centro: LONGINT;
    encontrado: BOOLEAN;

BEGIN
    encontrado := FALSE;
    inferior := 1;
    superior := tamano_indice;

    WHILE ( (inferior <= superior) AND (NOT encontrado) ) DO
        BEGIN
            centro := (inferior + superior) DIV 2; { División entera }

            IF (indice[centro].clave = legajo) THEN
                BEGIN
                    encontrado := TRUE;
                END
            ELSE
                BEGIN
                    IF (legajo < indice[centro].clave) THEN

```

```
        superior := centro - 1      { Buscar en la primera mitad }
    ELSE
        inferior := centro + 1;    { Buscar en la segunda mitad }
    END;
END;

IF (encontrado) THEN
    buscar := centro
ELSE
    { Si no lo encuentra, retorna -1 }
    buscar := -1;

END;

PROCEDURE indexar(VAR arch: t_arch_alumnos; VAR indice: t_indice; VAR
tamanio_indice: LONGINT);

VAR alumno:          t_alumno;
    posicion, i:  LONGINT;

BEGIN
    i := 1;
    RESET(arch);

    WHILE (NOT EOF(arch)) DO
        BEGIN
            posicion := FILEPOS(arch);
            READ(arch, alumno);

            indice[i].clave     := alumno.legajo;
            indice[i].posicion   := posicion;
            INC(i);
        END;

        tamanio_indice := i-1;

        { Reordenar indice }
        ordenar_indice(indice, tamanio_indice);
    END;

PROCEDURE cargar_alumno(VAR arch: t_arch_alumnos; VAR indice: t_indice; VAR
tamanio_indice: LONGINT);

VAR alumno:          t_alumno;
    posicion: LONGINT;

BEGIN
    WRITE('Legajo: ');
    READLN(alumno.legajo);
```

```

WRITE('Nombre y apellido: ');
READLN(alumno.nomyp);
WRITE('Documento de identidad: ');
READLN(alumno.documento);
WRITE('Dirección: ');
READLN(alumno.direccion);
WRITE('Fecha Nacimiento (DD MM AAAA): ');
READLN(alumno.fecha_nac.dia, alumno.fecha_nac.mes, alumno.fecha_nac.anio);

SEEK(arch, FILESIZE(arch)); { Posicionamiento al final, para agregado }
posicion := FILEPOS(arch);
WRITE(arch, alumno);

{ Nueva entrada en el indice }
INC(tamanio_indice);
indice[tamanio_indice].clave := alumno.legajo;
indice[tamanio_indice].posicion := posicion;

{ Reordenar indice }
ordenar_indice(indice, tamanio_indice);
END;

PROCEDURE eliminar_alumno(VAR arch: t_arch_alumnos; VAR indice: t_indice; VAR
tamanio_indice: LONGINT);

VAR legajo,i: LONGINT;

BEGIN
  WRITE('Legajo a eliminar: ');
  READLN(legajo);

  i := buscar(legajo, indice, tamanio_indice);
  IF (i = -1) THEN
    BEGIN
      WRITELN('El legajo ', legajo, ' no fue encontrado.');
    END
  ELSE
    BEGIN
      { Para borrar la entrada en el indice, es suficiente con asignar   }
      { el mayor valor posible para la clave y decrementar el tamaño del   }
      { índice. De esta manera, cuando se reordene el índice, ese   }
      { registro se moverá al final y no se tendrá en cuenta en futuros   }
      { accesos al índice. }
      indice[i].clave := LONG_MAX;
      ordenar_indice(indice, tamanio_indice);
      DEC(tamanio_indice);
    END;
  END;
END;

```

```

PROCEDURE mostrar_indice(indice: t_indice; tamanio_indice: LONGINT);

VAR i: LONGINT;

BEGIN
  FOR i:=1 TO tamanio_indice DO
    WRITELN(indice[i].clave, ' -> ', indice[i].posicion);
  WRITELN;
END;

VAR alumnos:          t_arch_alumnos;
indice:              t_indice;
tamanio_indice:      LONGINT;
opcion:              INTEGER;

{ Programa Principal }
BEGIN

ASSIGN(alumnos, 'alumnos_pas.dat');
RESET(alumnos);

{ Indexación del archivo }
indexar(alumnos, indice, tamanio_indice);

REPEAT
  WRITELN('1 -> Cargar nuevo alumno');
  WRITELN('2 -> Eliminar alumno existente');
  WRITELN('3 -> Salir');
  READLN(opcion);

  CASE (opcion) OF
    1: cargar_alumno(alumnos, indice, tamanio_indice);
    2: eliminar_alumno(alumnos, indice, tamanio_indice);
  END;
  UNTIL (opcion = 3);

mostrar_indice(indice, tamanio_indice);

CLOSE(alumnos);

END.

```

3- Con el mismo archivo ALUMNOS.DAT del primer problema del capítulo. cree un algoritmo que abra el archivo para lectura y se lo indexe por medio de #legajo y de #documento de identidad. Ambos índices deben mantenerse en la memoria principal.

Por último, liste el contenido del archivo completo, ordenado por #legajo (haciendo uso del índice por #legajo) y ordene por #documento de identidad (haciendo uso del índice por #documento de identidad). En este ejemplo se ilustra de qué manera se puede manipular un archivo como si estuviese ordenado por diferentes campos sin necesidad de reordenarlo.

Solución en lenguaje C

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

#define MAX_TAMANIO_INDICE 50

typedef struct {
    int dia, mes, anio;
} t_fecha;

typedef struct {
    long legajo;
    char nomayap[30+1];
    char documento[10+1];
    char direccion[40+1];
    t_fecha fecha_nac;
} t_alumno;

/* Índice por #legajo */
typedef struct {
    long clave;          /* Clave de acceso (legajo) */
    long posicion;       /* Posición del registro en el archivo */
} t_entrada_indice_legajo;
typedef t_entrada_indice_legajo t_indice_legajo[MAX_TAMANIO_INDICE];

/* Índice por #documento */
typedef struct {
    char clave[10+1]; /* Clave de acceso (documento de identidad) */
    long posicion;     /* Posición del registro en el archivo */
} t_entrada_indice_documento;
typedef t_entrada_indice_documento t_indice_documento[MAX_TAMANIO_INDICE];

typedef enum {FALSO, VERDADERO} t_bool;

void indexar(FILE* arch, t_indice_legajo ind_leg, t_indice_documento ind_doc,
long* tamano_indice);
void listar_por_legajo(FILE* arch, t_indice_legajo indice, long
tamano_indice);
void listar_por_documento(FILE* arch, t_indice_documento indice, long
tamano_indice);
void ordenar_indice_legajo(t_indice_legajo indice, long tamano_indice);
void ordenar_indice_documento(t_indice_documento indice, long tamano_indice);
void mostrar_datos(t_alumno alumno);

int main()
{
    FILE* alumnos;
    t_indice_legajo indice_legajo;
```

```
t_indice_documento indice_documento;
long tamanio_indice;

if ((alumnos=fopen("alumnos.dat","rb")) == NULL)
{
    printf("No se puede abrir el archivo.\n");
    return 1;
}

/* Indexación por #legajo y #documento */
indexar(alumnos, indice_legajo, indice_documento, &tamanio_indice);

/* Listado del archivo ordenado por #legajo */
listar_por_legajo(alumnos, indice_legajo, tamanio_indice);

/* Listado del archivo ordenado por #documento */
listar_por_documento(alumnos, indice_documento, tamanio_indice);

fclose(alumnos);

return 0;
}

void indexar(FILE* arch, t_indice_legajo ind_leg, t_indice_documento ind_doc,
long* tamanio_indice)
{
    t_alumno alumno;
    int posicion, i = 0;

    rewind(arch);

    do
    {
        posicion = ftell(arch); /* Posición actual del puntero a registro */
        fread(&alumno, sizeof(t_alumno), 1, arch);

        if (!feof(arch))
        {
            /* Nueva entrada en el índice por legajo */
            ind_leg[i].clave      = alumno.legajo;
            ind_leg[i].posicion   = posicion;

            /* Nueva entrada en el índice por documento de identidad */
            strcpy(ind_doc[i].clave, alumno.documento);
            ind_doc[i].posicion   = posicion;

            i++;
        }
    }
}
```

```
    } while (!feof(arch));

    *tamanio_indice = i;

    /* Reordenar índices */
    ordenar_indice_legajo(ind_leg, *tamanio_indice);
    ordenar_indice_documento(ind_doc, *tamanio_indice);
}

void listar_por_legajo(FILE* arch, t_indice_legajo indice, long
tamanio_indice)
{
    t_alumno alumno;
    int i;

    printf("LISTADO ORDENADO POR #LEGAJO\n\n");
    for (i=0; i<tamanio_indice; i++)
    {
        fseek(arch, indice[i].posicion, SEEK_SET);
        fread(&alumno, sizeof(t_alumno), 1, arch);
        mostrar_datos(alumno);
        printf("\n");
    }
}

void listar_por_documento(FILE* arch, t_indice_documento indice, long
tamanio_indice)
{
    t_alumno alumno;
    int i;

    printf("LISTADO ORDENADO POR #DOCUMENTO DE IDENTIDAD\n\n");
    for (i=0; i<tamanio_indice; i++)
    {
        fseek(arch, indice[i].posicion, SEEK_SET);
        fread(&alumno, sizeof(t_alumno), 1, arch);
        mostrar_datos(alumno);
        printf("\n");
    }
}

void ordenar_indice_legajo(t_indice_legajo indice, long tamanio_indice)
{
    int i, j;
    t_entrada_indice_legajo aux;

    for (i = tamanio_indice-1; i > 0; i--)
        for (j = 1; j <= i; j++)
            if (indice[j-1].clave > indice[j].clave)
```

```

    {
        /* Intercambio */
        aux = indice[j];
        indice[j] = indice[j-1];
        indice[j-1] = aux;
    }
}

void ordenar_indice_documento(t_indice_documento indice, long tamanio_indice)
{
    int i, j;
    t_entrada_indice_documento aux;

    for (i = tamanio_indice-1; i > 0; i--)
        for (j = 1; j <= i; j++)
            if (strcmp(indice[j-1].clave, indice[j].clave) > 0)
            {
                /* Intercambio */
                aux = indice[j];
                indice[j] = indice[j-1];
                indice[j-1] = aux;
            }
    }

void mostrar_datos(t_alumno alumno)
{
    printf("Legajo: %ld\n", alumno.legajo);
    printf("Nombre y apellido: %s\n", alumno.nomyap);
    printf("Documento de identidad: %s\n", alumno.documento);
    printf("Dirección: %s\n", alumno.direccion);
    printf("Fecha de nacimiento: %d/%d/%d\n", alumno.fecha_nac.dia, alumno.fecha_nac.mes, alumno.
    fecha_nac.anio);
}

```

Solución en lenguaje Pascal

```

PROGRAM cap08_ej03;

CONST MAX_TAMANIO_INDICE = 50;

TYPE t_fecha = RECORD
    dia, mes, anio: INTEGER;
END;

t_alumno = RECORD
    legajo:     LONGINT;
    nomyap:     STRING[30];
    documento:  STRING[10];
    direccion:  STRING[40];

```

```

    fecha_nac:  t_fecha;
END;

{ Índice por #legajo }
t_entrada_indice_legajo = RECORD
    clave:      LONGINT; { Clave de acceso (legajo) }
    posicion:  LONGINT; { Posición del registro en el archivo }
END;
t_indice_legajo = ARRAY[1..MAX_TAMANIO_INDICE] OF
t_entrada_indice_legajo;

{ Índice por #documento }
t_entrada_indice_documento = RECORD
    clave:      STRING[10]; { Clave de acceso (documento de identidad) }
    posicion:  LONGINT; { Posición del registro en el archivo }
END;
t_indice_documento = ARRAY[1..MAX_TAMANIO_INDICE] OF
t_entrada_indice_documento;

t_arch_alumnos = FILE OF t_alumno;
PROCEDURE ordenar_indice_legajo(VAR indice: t_indice_legajo; tamanio_indice: LONGINT);

VAR i,j:  LONGINT;
    aux:  t_entrada_indice_legajo;

BEGIN
    FOR i:=tamanio_indice DOWNTO 2 DO
        FOR j:=2 TO i DO
            IF (indice[j-1].clave > indice[j].clave) THEN
                BEGIN
                    { Intercambio }
                    aux := indice[j];
                    indice[j] := indice[j-1];
                    indice[j-1] := aux;
                END
    END;
PROCEDURE ordenar_indice_documento(VAR indice: t_indice_documento; tamanio_indice: LONGINT);

VAR i,j:  LONGINT;
    aux:  t_entrada_indice_documento;

BEGIN
    FOR i:=tamanio_indice DOWNTO 2 DO
        FOR j:=2 TO i DO
            IF (indice[j-1].clave > indice[j].clave) THEN
                BEGIN
                    { Intercambio }

```

```

        aux := indice[j];
        indice[j] := indice[j-1];
        indice[j-1] := aux;
    END
END;

PROCEDURE mostrar_datos(alumno: t_alumno);

BEGIN
    WRITELN('Legajo: ', alumno.legajo);
    WRITELN('Nombre y apellido: ', alumno.nomyap);
    WRITELN('Documento de identidad: ', alumno.documento);
    WRITELN('Dirección: ', alumno.direccion);
    WRITELN('Fecha de nacimiento: ', alumno.fecha_nac.dia, '/', alumno.fecha_nac.mes, '/',
            alumno.fecha_nac.anio);
END;
PROCEDURE indexar(VAR arch: t_arch_alumnos; VAR ind_leg: t_indice_legajo; VAR
ind_doc: t_indice_documento; VAR tamanio_indice: LONGINT);

VAR alumno: t_alumno;
    posicion, i: LONGINT;

BEGIN
    i := 1;
    RESET(arch);

    WHILE (NOT EOF(arch)) DO
        BEGIN
            posicion := FILEPOS(arch);
            READ(arch, alumno);

            ind_leg[i].clave := alumno.legajo;
            ind_leg[i].posicion := posicion;

            ind_doc[i].clave := alumno.documento;
            ind_doc[i].posicion := posicion;

            INC(i);
        END;

    tamanio_indice := i-1;

    { Reordenar indices }
    ordenar_indice_legajo(ind_leg, tamanio_indice);
    ordenar_indice_documento(ind_doc, tamanio_indice);
END;

PROCEDURE listar_por_legajo(VAR arch: t_arch_alumnos; VAR indice: t_indice_legajo; VAR tama-
nio_indice: LONGINT);

```

```

VAR alumno: t_alumno;
    i:      LONGINT;

BEGIN
    WRITELN('LISTADO ORDENADO POR #LEGAJO');
    WRITELN;

    FOR i:=1 TO tamanio_indice DO
        BEGIN
            SEEK(arch, indice[i].posicion);
            READ(arch, alumno);
            mostrar_datos(alumno);
            WRITELN;
        END;
    END;

PROCEDURE listar_por_documento(VAR arch: t_arch_alumnos; VAR indice: t_indice_documento; VAR tamanio_indice: LONGINT);

VAR alumno: t_alumno;
    i:      LONGINT;

BEGIN
    WRITELN('LISTADO ORDENADO POR #DOCUMENTO DE IDENTIDAD');
    WRITELN;

    FOR i:=1 TO tamanio_indice DO
        BEGIN
            SEEK(arch, indice[i].posicion);
            READ(arch, alumno);
            mostrar_datos(alumno);
            WRITELN;
        END;
    END;

VAR alumnos:          t_arch_alumnos;
    indice_legajo:   t_indice_legajo;
    indice_documento: t_indice_documento;
    tamanio_indice:  LONGINT;

{ Programa Principal }
BEGIN

    ASSIGN(alumnos, 'alumnos_pas.dat');
    RESET(alumnos);

    { Indexación por #legajo y #documento }
    indexar(alumnos, indice_legajo, indice_documento, tamanio_indice);

```

```
{ Listado del archivo ordenado por #legajo }
listar_por_legajo(alumnos, indice_legajo, tamanio_indice);

{ Listado del archivo ordenado por #documento }
listar_por_documento(alumnos, indice_documento, tamanio_indice);

CLOSE(alumnos);

END.
```

8.8 Contenido de la página Web de apoyo.



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- Organización de archivo por un índice.



Autoevaluación.



Video explicativo (02:38 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas. *



Presentaciones. *

9

Recurrencia

Contenido

9.1 Introducción	248
9.2 Algoritmos recursivos	248
9.3 Tipos de recursividad.....	252
9.4 Resumen.....	253
9.5 Problemas propuestos	254
9.6 Problemas resueltos.....	254
9.7 Contenido de la página Web de apoyo.....	262

Objetivos

- Explicar el concepto de recursividad. Se plantea su uso para soluciones a problemas que, por sus características, requieren el diseño de algoritmos que deben llamarse a sí mismos.
- Comprender la estrategia de implementación de problemas recursivos, para luego hacer un análisis comparativo entre las soluciones iterativas clásicas y las soluciones que utilizan la recursividad.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

9.1 Introducción.

El concepto de recurrencia (también suelen utilizarse los términos alternativos recursividad y recursión), puede aplicarse en los campos (fenómenos naturales y físicos, estructuras matemáticas, etc.) donde la definición o la construcción de una “cosa” puede llevarse a cabo a partir de la “cosa” en sí. Para ilustrar esta idea, podemos analizar un ejemplo de la naturaleza: Una hoja de helecho. En la imagen se muestra una hoja de *Pterophyta* (o “helecho”); si consideramos la porción marcada con un círculo, podemos comprobar que su morfología es igual a la de la hoja completa. Más aún, si continuamos tomando porciones más y más pequeñas veremos que esta estructura se conserva.

De manera similar, la naturaleza nos brinda otros tantos casos de estructuras recurrentes, como los copos de nieve, los cristales, los vasos sanguíneos, entre otros.

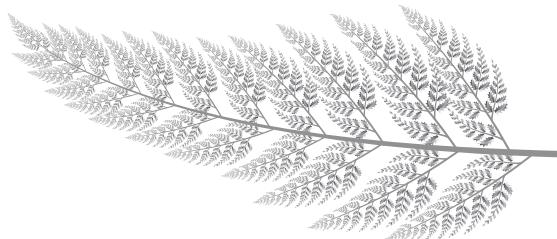


Fig. 9-1. Hoja de “helecho”.

La matemática también es un campo en el que prolifera el concepto de recurrencia. En particular, algunas funciones como la serie de Fibonacci o el cálculo del factorial presentan esta característica. Los fractales, por su parte, son estructuras geométricas que pueden definirse de manera recursiva; en la figura se muestra (de arriba hacia abajo) los pasos en la construcción de un fractal simple: El copo de nieve de Koch.

En el ámbito de la programación, en numerosas oportunidades nos topamos con algoritmos que se definen en forma recursiva, básicamente porque modelan un problema cuya naturaleza también lo es. El caso más ilustrativo es cuando se implementa (en un lenguaje de programación) la función factorial (o alguna función matemática recurrente). En esas situaciones la construcción de un algoritmo recursivo es la aproximación más simple. En este capítulo analizaremos el concepto de recurrencia en el campo de la algoritmia y los lenguajes de programación.

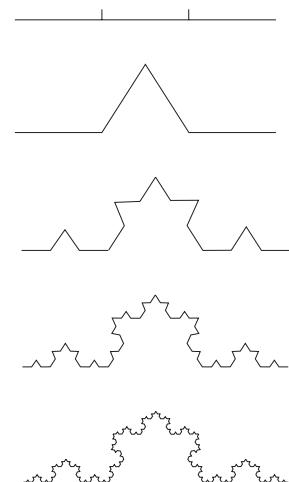


Fig. 9-2. Fractal simple (copo de nieve de Koch).

9.2 Algoritmos recursivos.

La noción de recurrencia también puede aplicarse a la construcción de algoritmos. En este caso, el algoritmo está definido a partir de sí mismo, lo que significa que si profundizamos en su implementación, encontraremos partes que son el propio algoritmo (autoinvocaciones), al igual que cuando observamos con más detalle una hoja de helecho. Por ejemplo, analicemos una función recursiva muy sencilla, capaz de imprimir los números naturales hasta cierto límite recibido como parámetro:

```
mostrar_naturales(5); /* Recibe el límite como parámetro */
```

Si suponemos que el límite es 5, la primera llamada puede dedicarse a mostrar el último número de la serie (en este caso, "5") y delegar el trabajo de imprimir la serie 1...4 a una llamada recursiva a sí misma, pero ahora con límite 4. Así, esta nueva llamada mostraría el último número de la serie (en este caso "4") y pasaría la responsabilidad de imprimir la serie 1...3 a una nueva llamada a sí misma, pero con límite 3. Este "anidamiento" de llamadas continúa hasta que se produce una condición de corte. En este ejemplo, la condición de corte se genera cuando una nueva llamada a `mostrar_naturales()` recibe el valor 1 como límite. Dado que no hay más números naturales inferiores a 1, entonces esta última llamada se encarga de mostrar el valor "1" y retomar el control a la llamada previa, que hará lo mismo hasta que se deshaga por completo el anidamiento de llamadas recursivas. En la figura siguiente se ilustra parte de este proceso:

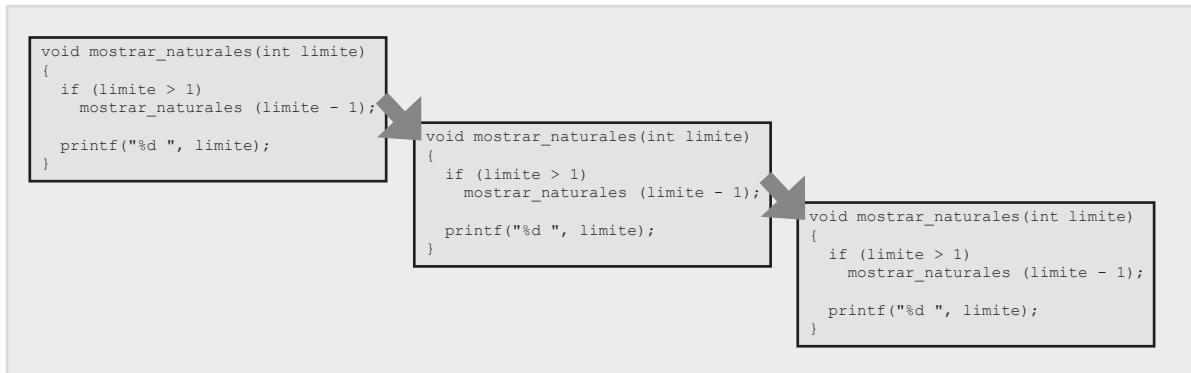


Fig. 9-3. "Anidamiento" de llamadas.

Si bien ya se describirá con más detalle a lo largo de este capítulo, es necesario que el lector comience a comprender que las sucesivas llamadas a la misma función son instancias diferentes de ella.

Los algoritmos recursivos presentan, básicamente, las características siguientes:

- Hay una autorreferencia (el algoritmo se llama a sí mismo).
- Existe una condición de corte; esto es, un caso en el que se decide "cortar" el anidamiento de llamadas.
- Hay un caso general en el que se evidencia la autorreferencia. Para este caso general se verifica que en un número finito de llamadas recursivas, se alcanza la condición de corte.

La importancia singular de esta técnica tiene que ver con que en numerosas ocasiones debemos programar la solución a un problema que es, en sí mismo, de naturaleza recursiva. Para este fin se desaconseja el uso de estructuras iterativas (`for`, `while`, `do-while`), ya que en los casos típicos este enfoque requiere mayor esfuerzo y líneas de código respecto de la versión recursiva del algoritmo. Sin embargo, un factor a tener en cuenta es el rendimiento del programa construido; el uso de autorreferencias para implementar la recurrencia tiene un costo muy elevado en *performance* que en muchas oportunidades no estamos dispuestos a "pagar" (en esos casos, la implementación de una solución iterativa puede ser más conveniente).

Un problema recursivo típico, tomado del campo de las matemáticas, es la función factorial. El factorial de un número natural n (que se expresa con la notación $n!$) está definido como:

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n \quad (\text{con caso particular } 0! = 1)$$

La "naturaleza recursiva" de esta función se observa al reescribir la relación anterior de la manera siguiente:

$$n! = \begin{cases} n \times (n-1)! & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

Así queda claro que, salvo para el caso $n = 0$ (condición de corte), el cálculo del factorial de un número requiere el cálculo del factorial de su antecesor. A continuación se muestra la implementación, en lenguaje C, de la función factorial y el modo de invocación para $n = 3$:

```
#include <stdio.h>

long factorial(int n)
{
    long aux;

    if (n > 0)
        aux = n * factorial(n - 1); /* Llamada recursiva */
    else
        aux = 1; /* Condición de corte */

    return aux;
}

int main()
{
    int n = 3;
    printf("El factorial de %d es %lu\n", n, factorial(n));
    return 0;
}
```

Antes de analizar el comportamiento de las llamadas recursivas, es necesario que el lector repase el concepto de mapa de memoria expuesto en el capítulo 3. En particular, debe quedar claro el uso de la pila (*stack*), que es la porción de memoria donde las rutinas (procedimientos y funciones) almacenan sus variables locales y parámetros durante su ejecución. Entender el funcionamiento del *stack* es crítico para comprender cómo una computadora implementa las llamadas recursivas.

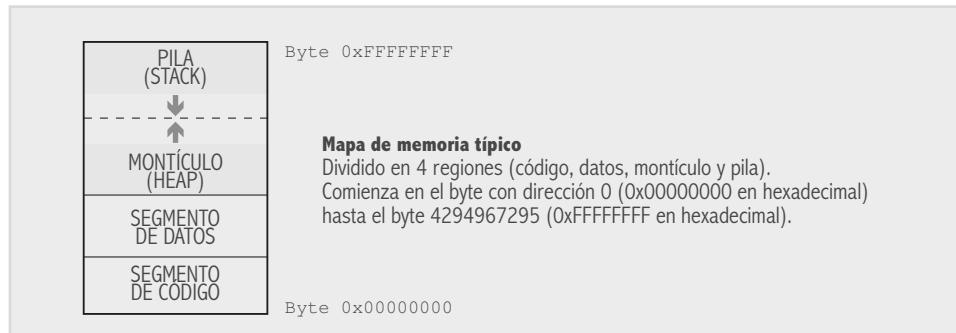


Fig. 9-4. Mapa de memoria (uso de la pila).

Cuando se hace la primera llamada a la función `factorial()` con parámetro $n = 3$, el *stack* (que asumimos que al inicio estaba vacío) presenta la forma siguiente:

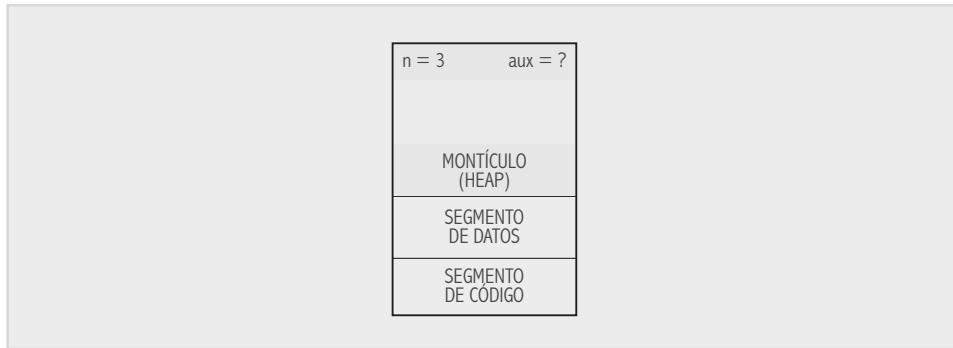


Fig. 9-5. Representación de la pila cuando se hace la primera llamada a función `factorial()`.

La primera llamada a `factorial()` tomó una porción de la pila, en la que almacena el parámetro recibido n (con valor 3) y la variable local aux (aún no inicializada). Debido a que n es mayor que 0, se hace una llamada recursiva a `factorial()` con parámetro n igual a 2. Ahora, el *stack* toma la forma siguiente:

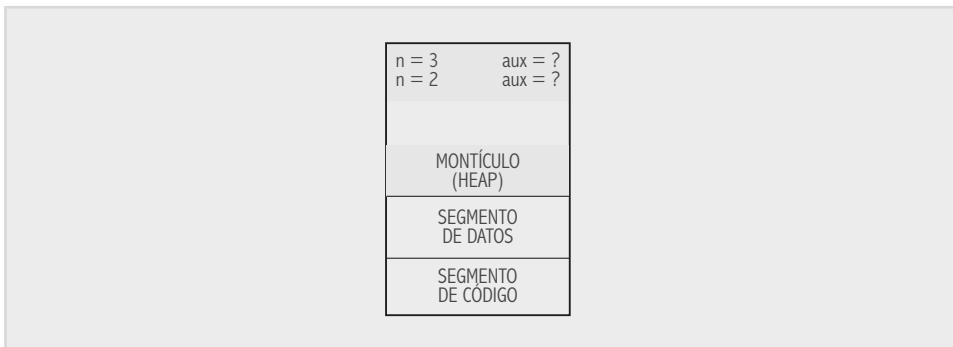


Fig. 9-6. Representación de la pila cuando se hace una llamada recursiva a `factorial()`.

En este punto es importante notar (y entender) que la segunda llamada a `factorial()` tomó una nueva porción del *stack*, y por eso afirmamos que es independiente de la llamada anterior (los parámetros n y las variables locales aux son independientes, ocupan posiciones diferentes de memoria). En esta segunda llamada, debido a que n es mayor que 0, la variable local aux tampoco puede inicializarse y se ejecuta una nueva llamada recursiva. A continuación se muestra el nuevo estado de la pila:

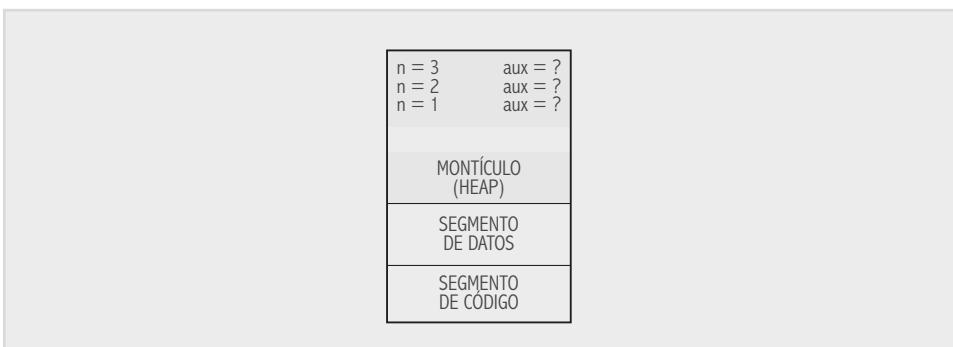


Fig. 9-7. Estado de la pila cuando se ejecuta una nueva llamada recursiva.

El parámetro n aún es mayor que 0; se llama recursivamente a `factorial()` y el *stack* presenta la forma siguiente:

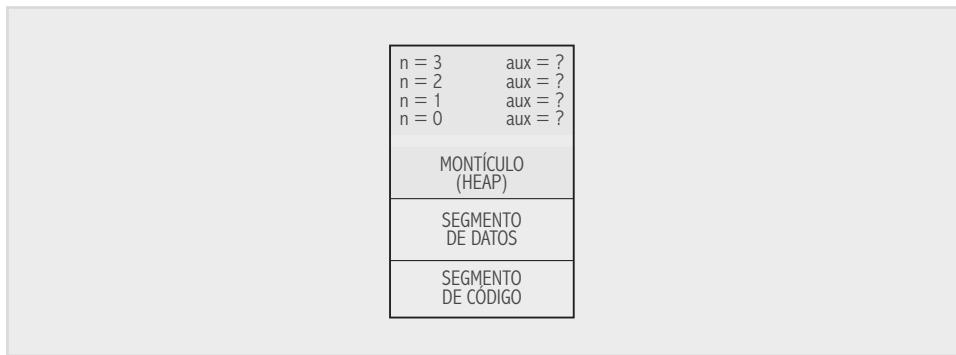


Fig. 9-8. Forma que presenta la pila con una tercer llamada recursiva.

Por último, se alcanza la condición de corte ($n = 0$). No se continúa con las llamadas recursivas y la función retorna el valor 1. Cada una de las llamadas previas hará lo mismo hasta que, al final, la primera de ellas vuelva al punto del programa principal donde fue invocada. De esta manera, la pila se “deshace” en el sentido contrario a como se construyó, lo que se ilustra en las figuras siguientes:

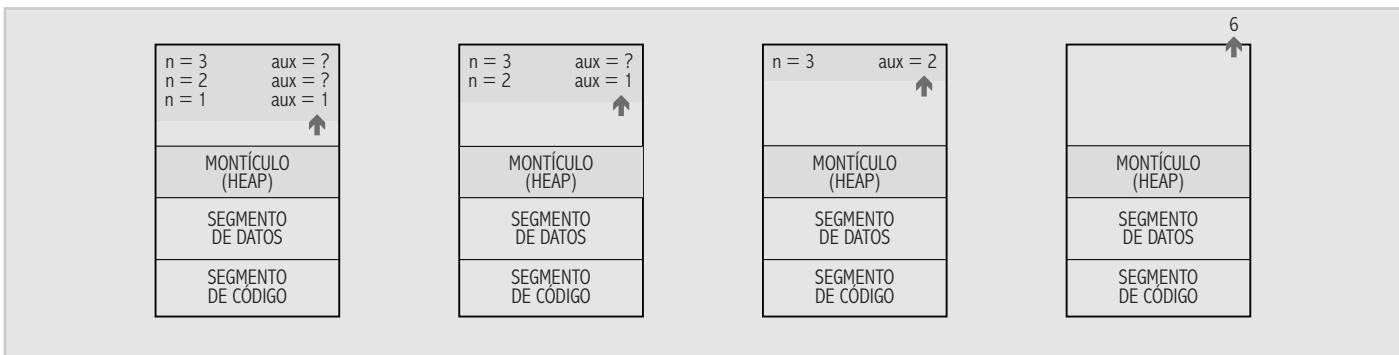


Fig. 9-9. Modo en que se “deshace” la pila en sentido inverso a como se construyó.

A medida que se produce el “anidamiento” de llamadas recursivas, el tamaño de la pila crece en igual proporción. Éste no es un detalle menor; significa que, según el número de llamadas recursivas, puede agotarse el espacio disponible en el *stack*, situación conocida como desbordamiento de pila (*stack overflow*).

Otra consideración que debe hacerse cuando se implementa un algoritmo recursivo tiene que ver con el rendimiento (*performance*) del programa en cuestión. Llamar a una subrutina tiene un precio elevado; todo “buen programador” debe utilizar herramientas para medición de rendimiento y decidir hasta qué punto conviene implementar una solución recursiva a un problema.



Encuentre un simulador sobre recursividad directa en la Web de apoyo.

9.3 Tipos de recursividad.

A lo largo de este capítulo se desarrolló el caso en que una función “A” se llama a sí misma. A este tipo de recursividad se la denomina directa.

Cuando una subrutina “A” llama a otra subrutina “B”, y ésta, por su parte, vuelve a invocar a “A”, la situación es diferente. A este tipo de recursividad se la denomina indirecta. En cualquier caso (autorreferencias directas o indirectas) debe haber una condición de corte.

Cuando la llamada recursiva es el último paso ejecutado por la función, hablamos de recursividad de cola (*tail recursion*). A continuación se presenta la función factorial, ahora escrita con recursividad de cola:

```
long factorial(int n)
{
    if (n == 0)
        return 1; /* Condición de corte */

    return (n * factorial(n-1)); /* Llamada recursiva */
}
```

El uso de este tipo de recursividad permite que el compilador aplique optimizaciones que pueden favorecer el rendimiento.

9.4 Resumen.

El punto más importante que debe rescatar el lector de este capítulo es entender que la recursividad no es una competencia exclusiva de la informática; por el contrario, *a priori* podemos encontrar en otros ámbitos ejemplos de situaciones con carácter recursivo, como una hoja de helecho, un copo de nieve o una estructura fractalica.

De manera de poder tratar problemas que son de naturaleza recursiva, muchos lenguajes de programación permiten la construcción de algoritmos recursivos. En este caso, el algoritmo está definido a partir de sí. En un algoritmo implementado en forma recursiva hay una **autorreferencia** (el algoritmo se llama a sí mismo), una **condición de corte** y un **caso general** en el que se evidencia la autorreferencia.

Considerando el mapa de memoria de una computadora (descripto en el capítulo 3), el *stack* cumple un papel fundamental en la implementación recursiva de un algoritmo. Las autoinvocaciones suelen generar confusión respecto de si las variables de la invocación actual sobreescriben (o no) las de la invocación previa. El *stack* es una pila, lo que significa que las distintas invocaciones “apilarán” sus variables, sin que se produzcan conflictos entre sucesivas llamadas del mismo algoritmo. Si el lector es capaz de comprender esta idea, habrá entendido la clave del funcionamiento de la recursividad en una computadora.

Existen tres tipos de recursividad:

- **Directa:** Cuando una subrutina se llama a sí misma.
- **Indirecta:** Cuando una subrutina “A” llama a otra subrutina “B”, y ésta, por su parte, vuelve a invocar a “A”.
- **De cola:** Cuando la llamada recursiva es el último paso ejecutado por la función.

Otra consideración que debe hacerse cuando se implementa un algoritmo recursivo tiene que ver con el desempeño (*performance*) del programa en cuestión. Llamar a una subrutina tiene un precio elevado; todo “buen programador” debe utilizar herramientas para medición de desempeño y decidir hasta qué punto conviene implementar una solución recursiva a un problema.



Encuentre un simulador sobre recursividad indirecta en la Web de apoyo.

9.5 Problemas propuestos.

- 1) Escriba una función recursiva para calcular el término n -ésimo de la secuencia de Lucas: 1, 3, 4, 7, 11, 18, 29, 47, ...
- 2) Dado un arreglo de enteros, diseñar algoritmos recursivos que calculen:
 - a) El mayor elemento del arreglo.
 - b) La suma de los elementos del arreglo.
 - c) La media de todos los elementos del arreglo.
- 3) Diseñar un algoritmo recursivo que imprima los dígitos de un número decimal en orden inverso.
- 4) Escriba una versión recursiva del cálculo del máximo común divisor de dos números enteros por el método de Euclides.
- 5) Escribir el procedimiento de búsqueda binaria de forma recursiva.
- 6) Sabiendo que 0 es par, esto es,

`EsPar (0) = true`

`Eslmpar (0) = false`

y que la paridad de cualquier otro entero positivo es la opuesta que la del entero anterior, desarrolle las funciones lógicas, mutuamente recursivas, `EsPar` y `Eslmpar`, que se complementen a la hora de averiguar la paridad de un entero positivo.

- 7) Un palíndromo es una palabra capicúa, o sea que se lee igual de derecha a izquierda que de izquierda a derecha. Sea $P(n)$ el número de palíndromos de longitud n , formados usando sólo letras minúsculas, diseñar una función recursiva que calcule $P(n)$.
- 8) Escribir un programa recursivo que ordene un arreglo por el método mezcla: Se va dividiendo el arreglo sucesivamente en dos mitades, y cuando la longitud de cada mitad sea 2 se comparan los elementos y se ordenan. Después de ordenadas las dos mitades, ambas se mezclan en forma ordenada en un solo arreglo, aprovechando que las mitades ya están ordenadas.

9.6 Problemas resueltos.

- 1- Implementación recursiva de la función factorial. Esta función se define como:

$$n! = \begin{cases} n \times (n-1)! & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

El usuario ingresa por teclado el valor de n (y se valida que sea natural).

Solución en lenguaje C

```
#include <stdio.h>

unsigned long factorial(int n)
{
    if (n == 0)
        return 1; /* Condición de corte */

    return (n * factorial(n-1)); /* Llamada recursiva */
}

int main()
{
    int n;

    printf("Ingrese el valor de N: ");
}
```



```

scanf("%d", &n);
if (n >= 0)
    printf("\t%d! = %lu\n", n, factorial(n));
else
    printf("No es posible calcular el factorial de números negativos!\n");

return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap09_ej01;

FUNCTION factorial(n: INTEGER): LONGWORD;

BEGIN
    IF (n = 0) THEN
        factorial := 1      { Condición de corte }
    ELSE
        factorial := (n * factorial(n-1)); { Llamada recursiva }
    END;
    VAR n: INTEGER;

{ Programa Principal }
BEGIN

    WRITE('Ingrese el valor de N: ');
    READLN(n);
    IF (n >= 0) THEN
        WRITELN(' ', n, '! = ', factorial(n))
    ELSE
        WRITELN('No es posible calcular el factorial de números negativos!');

END.

```

2- Implementación recursiva de la serie de Fibonacci. Para obtener el n -ésimo elemento de esta sucesión, se emplea la expresión siguiente:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n > 1 \end{cases}$$

El usuario ingresa por teclado el valor de n (y se valida que sea natural).

Solución en lenguaje C

```

#include <stdio.h>

unsigned long fibonacci(int n)

```

```

{
    if (n == 0)
        return 0; /* Condición de corte */

    if (n == 1)
        return 1; /* Condición de corte */

    return (fibonacci(n-1) + fibonacci(n-2)); /* Llamadas recursivas */
}

int main()
{
    int n;

    printf("Ingrese el valor de N: ");
    scanf("%d", &n);
    if (n >= 0)
        printf("\tEl %do elemento de la serie de Fibonacci es = %lu\n", n+1,
fibonacci(n));
    else
        printf("La serie de Fibonacci está conformada con números positivos!\n");

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap09_ej02;

FUNCTION fibonacci(n: INTEGER): LONGWORD;

BEGIN
    IF (n = 0) THEN
        fibonacci := 0 { Condición de corte }
    ELSE
        IF (n = 1) THEN
            fibonacci := 1 { Condición de corte }
        ELSE
            fibonacci := fibonacci(n-1) + fibonacci(n-2); { Llamadas recursivas }
END;

VAR n: INTEGER;

{ Programa Principal }
BEGIN
    WRITE('Ingrese el valor de N: ');
    READLN(n);

```

```

IF (n >= 0) THEN
    WRITELN(' El ', n+1, 'o elemento de la serie de Fibonacci es = ',
fibonacci(n))
ELSE
    WRITELN('La serie de Fibonacci está conformada con números positivos!');

END.

```

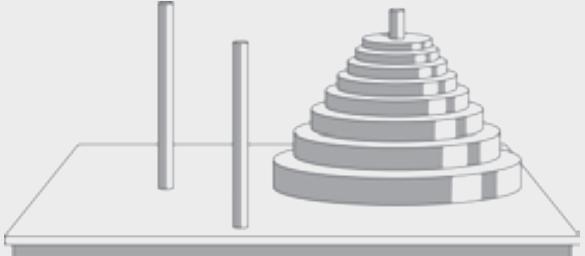
3- El juego de las Torres de Hanoi fue inventado a fines del siglo XIX y es un problema interesante para resolver de forma recursiva. Consiste en tres pilas (o varillas) verticales denominadas origen, destino y auxiliar. En la pila de origen hay un número n de discos, dispuestos de mayor a menor tamaño, como se muestra en la figura siguiente:

El objetivo del “rompecabezas” es mover todos los discos desde la pila de origen hacia la de destino, haciendo uso de la varilla auxiliar. Aunque su solución parezca trivial, deben respetarse las reglas que siguen:

- Sólo podrá desplazarse el disco que se encuentre en el tope de la pila.
- En ningún momento podrá haber un disco de mayor tamaño sobre otro más pequeño.

Los discos están numerados desde 1 hasta n (el disco 1 es el más pequeño, y el n , el más grande).

Se recomienda que, luego de analizar este caso, el lector implemente la solución a este problema de manera iterativa (no recursiva) mediante estructuras de control repetitivas.



Solución en lenguaje C

```

#include <stdio.h>

#define ORIGEN      1
#define DESTINO     2
#define AUXILIAR   3

void hanoi(int cant_discos, int origen, int destino, int auxiliar)
{
    /* No hay más discos para mover */
    if (cant_discos == 0)
        return;

    /* Se mueven todos los discos, salvo el de más abajo, a la pila auxiliar */
    hanoi(cant_discos - 1, origen, auxiliar, destino);

    /* Una vez que ha quedado libre, se mueve el último disco */
    /* a la pila de destino */
    printf("Moviendo disco %d desde la pila %d hacia la pila %d\n", cant_discos,
origen, destino);

    /* Por último se mueven todos los discos que estaban (temporalmente) en */
    /* la pila auxiliar hacia la pila de destino. */
    hanoi(cant_discos - 1, auxiliar, destino, origen);
}

int main()

```

```

{
    int cant_discos;

    printf("Juego de las Torres de Hanoi\n");
    printf("*****\n");

    printf("Ingrese la cantidad de discos: ");
    scanf("%d", &cant_discos);
    if (cant_discos > 0)
    {
        hanoi(cant_discos, ORIGEN, DESTINO, AUXILIAR);
        printf("FIN DEL ROMPECABEZAS!\n\n");
    }
    else
    {
        printf("La cantidad de discos a utilizar debe ser mayor que cero!\n");
    }

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap09_ej03;

CONST ORIGEN      = 1;
CONST DESTINO     = 2;
CONST AUXILIAR   = 3;

PROCEDURE hanoi(cant_discos, origen, destino, auxiliar: INTEGER);

BEGIN
    IF (cant_discos > 0) THEN
    BEGIN
        { Se mueven todos los discos, salvo el de más abajo, }
        { a la pila auxiliar }
        hanoi(cant_discos - 1, origen, auxiliar, destino);

        { Una vez que quedó libre, se mueve el último }
        { disco a la pila de destino }
        WRITELN('Moviendo disco ', cant_discos, ' desde la pila ', origen, ' '
hacia la pila ', destino);

        { Por último se mueven todos los discos que estaban (temporalmente) en }
        { la pila auxiliar hacia la pila de destino. }
        hanoi(cant_discos - 1, auxiliar, destino, origen);
    END;
END;

```

```

VAR cant_discos: INTEGER;

{ Programa Principal }
BEGIN
  WRITELN('Juego de las Torres de Hanoi');
  WRITELN('*****');

  WRITE('Ingrese la cantidad de discos: ');
  READLN(cant_discos);
  IF (cant_discos > 0) THEN
    BEGIN
      hanoi(cant_discos, ORIGEN, DESTINO, AUXILIAR);
      WRITELN('FIN DEL ROMPECABEZAS!');
      WRITELN;
    END
  ELSE
    BEGIN
      WRITELN('La cantidad de discos a utilizar debe ser mayor que cero!');
    END;
  END.

```

4- Localice un entero en un arreglo usando una versión recursiva del algoritmo de búsqueda binaria. El número que se debe buscar (clave) se debe ingresar como argumento por línea de comandos. El arreglo de datos debe estar previamente ordenado (requisito de la búsqueda binaria).

Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

Solución en lenguaje C

```

#include <stdio.h>
#include <stdlib.h>

#define N 10

typedef int t_arreglo_enteros[N];

/* Prototipo */
int buscar(int clave, t_arreglo_enteros datos, int inferior, int superior);

/* Programa Principal */
int main(int argc, char *argv[])
{
  t_arreglo_enteros datos = { 3 , 4 , 6 , 12 , 13 , 17 , 19 , 24 , 29 , 30 };
  int pos;
  if (argc != 2)
  {
    printf("Modo de uso: %s <clave a buscar>\n", argv[0]);
    return 1;
  }

```

```

/* Búsqueda */
pos = buscar(atoi(argv[1]), datos, 0, N-1);

if (pos != -1)
    printf("El entero %d se encontró en la posición %d\n", atoi(argv[1]),
pos);
else
    printf("El entero %d no pudo encontrarse\n", atoi(argv[1]));

return 0;
}

int buscar(int clave, t_arreglo_enteros datos, int inferior, int superior)
{
    int centro;

    /* Si el dato no se encuentra, retorna -1 */
    if (inferior > superior)
        return -1;

    centro = (inferior + superior) / 2; /* División entera */

    if (datos[centro] == clave)
        return centro;
    else
        if (datos[centro] > clave)
            return buscar(clave, datos, inferior, centro-1);
        else
            return buscar(clave, datos, centro+1, superior);
}

```

Solución en lenguaje Pascal

```

PROGRAM cap09_ej04;

CONST N = 10;

TYPE t_arreglo_enteros = ARRAY[1..N] OF INTEGER;

FUNCTION buscar(clave: INTEGER; datos: t_arreglo_enteros; inferior, superior:
INTEGER): INTEGER;

VAR centro: INTEGER;
BEGIN
    IF (inferior > superior) THEN
    BEGIN
        { Si el dato no se encuentra, retorna -1 }
        buscar := -1;
    END
END

```

```
ELSE
BEGIN
    centro := (inferior + superior) DIV 2; { División entera }

    IF (datos[centro] = clave) THEN
        buscar := centro
    ELSE
        IF (datos[centro] > clave) THEN
            buscar := buscar(clave, datos, inferior, centro-1)
        ELSE
            buscar := buscar(clave, datos, centro+1, superior);
    END;
END;

VAR datos: t_arreglo_enteros;
    clave: INTEGER;
    error: INTEGER;
    pos: INTEGER;

{ Programa Principal }
BEGIN

    IF (ParamCount <> 1) THEN
        BEGIN
            WRITELN('Modo de uso: ', ParamStr(0), ' <clave a buscar>');
            EXIT;
        END;

    VAL(ParamStr(1), clave, error);

    datos[1] := 3;
    datos[2] := 4;
    datos[3] := 6;
    datos[4] := 12;
    datos[5] := 13;
    datos[6] := 17;
    datos[7] := 19;
    datos[8] := 24;
    datos[9] := 29;
    datos[10] := 30;

    { Búsqueda }
    pos := buscar(clave, datos, 1, N);

    IF (pos <> -1) THEN
        WRITELN('El entero ', clave, ' se encontró en la posición ', pos)
    ELSE
        WRITELN('El entero ', clave, ' no pudo encontrarse');

END.
```

9.7 Contenido de la página Web de apoyo.



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- Recursividad directa.
- Recursividad indirecta.



Autoevaluación.



Video explicativo (02:24 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas. *



Presentaciones. *

10

Memoria dinámica y manejo de punteros

Contenido

10.1 Introducción	264
10.2 Administración de memoria dinámica	265
10.3 Punteros	265
10.4 Punteros sin tipo	271
10.5 Aritmética de punteros.....	273
10.6 Punteros y arreglos	274
10.7 Punteros a funciones.....	275
10.8 Resumen	275
10.9 Problemas propuestos	277
10.10 Problemas resueltos	277
10.11 Contenido de la página Web de apoyo.....	290

Objetivos

- Introducir el uso de punteros como mecanismo para el manejo de la memoria.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

10.1 Introducción

Todas las variables y las estructuras de datos declaradas en el ámbito de un programa residen en memoria. Así, por ejemplo, una variable de tipo entero ocupará 4 bytes (dependiendo de la arquitectura en que se trabaje), una cadena de 10 caracteres, otros 11 bytes (10 para los caracteres en sí mismos, y un byte adicional para el carácter de fin de cadena), y un dato booleano, 1 byte. Este espacio en memoria es solicitado en el instante en que arranca la ejecución del programa, y liberado cuando termina. En este caso, las variables en cuestión se denominan estáticas.

En la mayoría de los casos tomar memoria en forma estática suele ser un enfoque demasiado conservador. Supongamos que un programa se ejecuta durante un lapso determinado y que una variable X sólo se utiliza de manera esporádica durante períodos muy breves, como se ilustra en la figura siguiente:

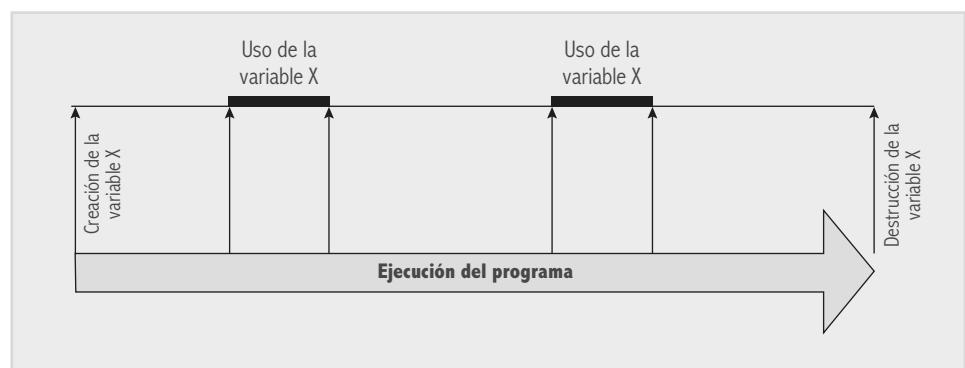


Fig. 10-1. Asignación de memoria a variables y estructuras en forma estática.

En esa situación, mantener en memoria variables o estructuras que no se utilizan no es una buena política si se pretende hacer un uso racional de la memoria disponible. Otra alternativa para asignar memoria a variables y estructuras de datos es hacerlo de manera dinámica: La memoria se solicita y se libera en el punto donde se hará uso de ella (durante la ejecución del programa):

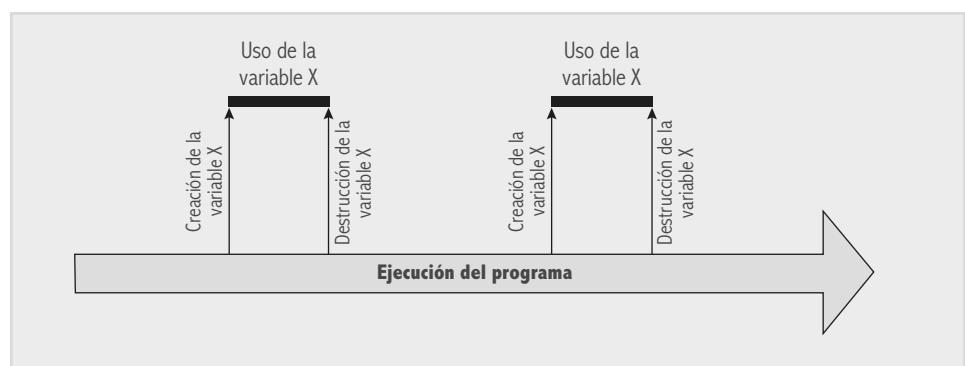


Fig. 10-2. Asignación de memoria a variables y estructuras en forma dinámica.



En la Web de apoyo encontrará un capítulo del libro *Arquitectura de Computadoras*, de Quiroga, sobre memorias.

Así, es posible hacer un uso más eficiente de la memoria disponible, al evitar que variables y estructuras “ocupen espacio” cuando en realidad no se utilizan. Sin embargo, administrar la memoria en forma dinámica puede tener un costo significativo debido al tiempo necesario para crear y destruir la memoria durante el tiempo de ejecución del programa. Este costo se amortiza si las variables se declaran estáticamente, ya que sólo se “paga” una vez, al principio y al final

de la ejecución. Como siempre sucede en estas situaciones de compromiso, es el programador quien debe decidir respecto de la conveniencia en el uso dinámico o estático de memoria.

El acceso a una variable dinámica no puede llevarse a cabo por su nombre, como sucede con las variables estáticas, sino por su dirección en memoria. Para esto se utiliza un mecanismo denominado puntero y que analizaremos a lo largo de este capítulo.

10.2 Administración de memoria dinámica.

Lenguajes de programación como C y Pascal brindan la posibilidad de administrar (solicitar y liberar) memoria en forma dinámica, durante la ejecución del programa, según las necesidades. Esta memoria puede tomarse desde el segmento denominado montículo (*heap*):

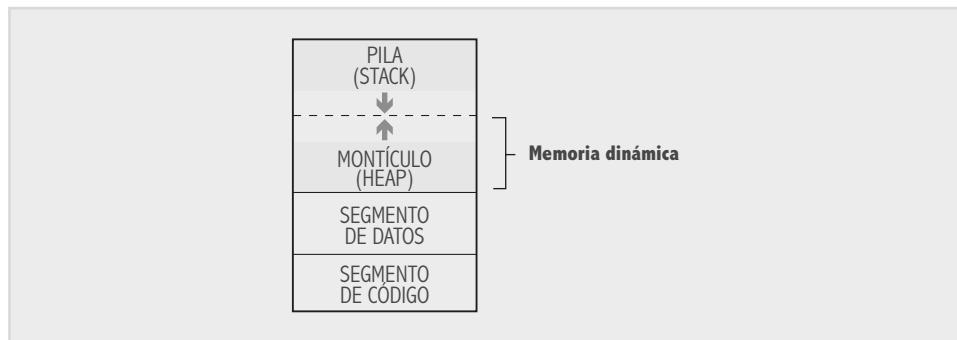


Fig. 10-3. Ubicación de la memoria dinámica.

La gestión de la memoria dinámica corresponde al sistema operativo. Así, un proceso (en tiempo de ejecución) solicita al sistema operativo una cantidad determinada de memoria dinámica. Éste tomará memoria del *heap* y devolverá al proceso la dirección del primer byte de este bloque asignado. Si no fuese posible tomar la cantidad solicitada, porque no hay memoria dinámica suficiente en ese momento, el sistema operativo se lo notificará al programa solicitante, por ejemplo, al retornar un puntero “nulo” como respuesta.

En algún punto durante la ejecución el programa libera la memoria que se solicitó con anterioridad, en los casos típicos porque ya no la necesita. Esta liberación es explícita, mediante una llamada a una función o procedimiento para ese fin, que notifica de esta situación al sistema operativo. Por otra parte, en lenguajes de programación como Perl, Smalltalk o Java, la liberación de memoria dinámica no es una tarea competente al programa, en cambio, hay un mecanismo (transparente al usuario) denominado “recolector de basura” (*garbage collector*) encargado de detectar porciones de memoria que no están en uso y liberarlas.

10.3 Punteros.

El concepto de puntero es muy simple; se trata de una variable capaz de alojar una dirección de memoria. Dado que, en la práctica, las direcciones de memoria se representan como enteros largos (sin signo), entonces un puntero no es más que una variable entera. Aunque la definición de puntero no reviste dificultad, su manipulación suele acarrear muchos dolores de cabeza.

En primer lugar, los punteros pueden clasificarse en tipados y no tipados. Un puntero con tipo es aquel que “apunta” a un tipo de dato determinada *a priori* (por ejemplo, puntero a entero, puntero a carácter, puntero a cadena). Por otra parte, un puntero sin tipo “apunta” a una porción genérica en memoria (no se especifica de antemano qué clase de datos se almacenarán allí).



Si la liberación de memoria debe hacerse de manera explícita, como sucede en C y Pascal, entonces el programador deberá tener la precaución suficiente para liberar (en algún momento) cada bloque de memoria asignado dinámicamente. No hacerlo implica que habrá memoria tomada y no usada, lo cual es un desperdicio. Este problema se ve agravado en ciertos escenarios muy comunes; por ejemplo, supongamos que en cada una de 10 000 iteraciones de un bucle *for* se toma 1 KB de memoria para usarlo en cálculos temporales durante la iteración, pero que nunca se libera. En ese caso al final de la última iteración tendremos ¡10 MB de memoria tomada inútilmente! A esa situación, muy común sobre todo entre los programadores novatos, se la denomina “fuga de memoria” (*memory leak*) y debe evitarse por todos los medios.

En lenguaje C, la declaración de una variable puntero se lleva a cabo de la siguiente manera:

```
<tipo base> *<nombre variable>;
```

Si `<tipo base>` es un tipo de dato conocido, entonces se trata de un puntero con tipo. Si, en cambio, `<tipo base>` es `void`, el puntero es "genérico" (sin tipo). En el ejemplo siguiente se declara un puntero a entero y se lo hace "apuntar" a la variable `numero`; de esta forma, acceder a `numero` a través de su nombre o mediante el puntero es equivalente:

```
#include <stdio.h>
int main()
{
    int numero;
    int *p_numero; /* Declaración de un puntero a entero */
    numero = 5;
    p_numero = &numero;
    *p_numero = 10; /* Indirección del puntero */
    printf("numero = %d\n", numero);
    return 0;
}
```

En este programa hay dos puntos clave en los que es necesario detenerse. En primer lugar, en la línea

```
p_numero = &numero;
```

se toma la dirección de memoria de la variable `numero` y se la asigna a `p_numero`, mediante el operador `&`. En segundo lugar, en la línea

```
*p_numero = 10;
```

no se accede al contenido de la variable puntero, sino adonde él apunta. Debido a que `p_numero` apunta a `numero`, entonces esta asignación modifica el contenido original de `numero`. El resultado de la ejecución del programa muestra el valor "10" en pantalla (aunque la variable `numero` se haya inicializado con el valor 5).

En la figura siguiente se ilustra el comportamiento del programa anterior. En la parte (a) se observa la declaración de las variables y la inicialización de `numero` con el valor 5. Nótese que la variable `numero` está alojada en la posición `0xaf1100862` de memoria. En la parte (b), el puntero se inicializa con la dirección de la variable `numero`. Por último, en la parte (c), se modifica el contenido de `numero` y se accede en forma indirecta por medio del puntero.

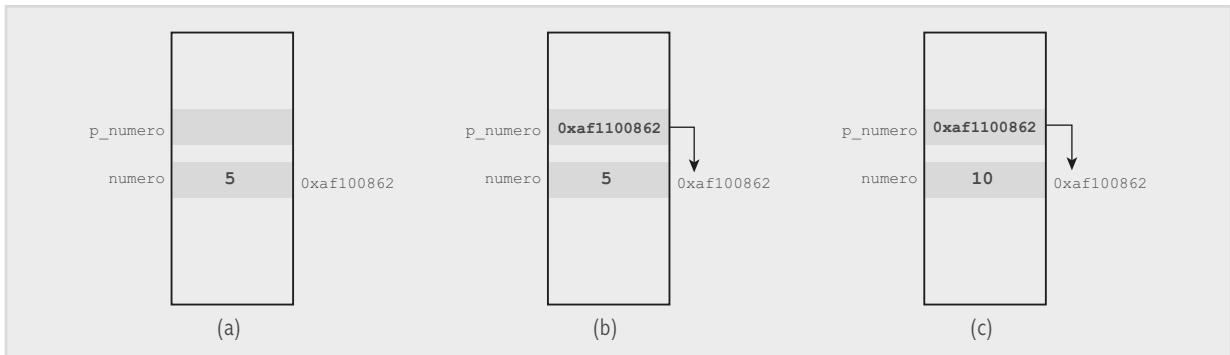


Fig. 10-4. Puntero a entero que "apunta" a la variable `numero`.

Siguiendo con el análisis del ejemplo dado, es importante notar la diferencia entre las siguientes sentencias:

```
1) *p_numero = 10;      2) p_numero = 10;
```

En la sentencia 1) se asigna el valor 10 en la posición de memoria, que es apuntada por `p_numero`. En la sentencia 2), en cambio, se sobreescribe el contenido del puntero (`p_numero` quedaría apuntando a la dirección 10 de memoria). Asignar un valor absoluto a una variable puntero, como ocurre en la sentencia 2), en la mayoría de los casos es un error de programación.

Por su parte, la declaración de un puntero en Pascal se realiza respetando la sintaxis siguiente:

Con tipo:

```
var <nombre variable>: ^<tipo base>;
```

Sin tipo (genérico):

```
var <nombre variable>: pointer;
```

A continuación se muestra el mismo ejemplo dado antes, pero ahora escrito en lenguaje Pascal:

```
program punteros;

var numero: integer;
    p_numero: ^integer; { Declaración de un puntero a entero }

begin
    numero := 5;
    p_numero := @numero;
    p_numero^ := 10; { Indirección del puntero }
    writeln('numero = ', numero);
end.
```

El operador unario `@` de Pascal equivale al `&` de C, mientras que el operador de indirección `^` de Pascal equivale al `*` de C. Más allá de las diferencias sintácticas, el concepto y el uso de punteros es igual en ambos lenguajes.

10.3.1 Punteros a memoria dinámica.

En la sección anterior los punteros declarados apuntan a variables estáticas (por ejemplo, el puntero `p_numero` apunta a la variable estática `numero`). Sin embargo, y como se comentó en la introducción de este capítulo, con frecuencia los punteros se utilizan para la administración de memoria dinámica.

Como ya se indicó, la memoria dinámica es aquella que el programa solicita y libera durante su ejecución, en función de la demanda. En lenguajes como C y Pascal, tanto la petición como la liberación de esta memoria se lleva a cabo mediante rutinas (funciones y procedimientos) para acceder al sistema operativo. En el caso particular de C, las dos funciones más importantes para este fin son:

```
void *malloc(size_t size);
void free(void *ptr);
```



Es importante que quede claro que las posiciones de memoria dinámica no tienen un nombre "amigable", como sucede con las variables estáticas, por eso, debemos acceder a ellas mediante sus direcciones en memoria. Ésta es una de las razones de la existencia de los punteros.

`malloc()` recibe el tamaño, en bytes, del espacio de memoria que se requiere y devuelve un puntero a void al primer byte de ese bloque de memoria, o el valor NULL si no encuentra la cantidad de memoria contigua solicitada. NULL es una constante cuyo valor significa sin dirección de memoria definida.

`free()` recibe un puntero, y libera el bloque de memoria dinámica al que apunta. Ambas funciones están definidas en la biblioteca `stdlib.h`.

A continuación se ofrece un ejemplo de uso de las funciones `malloc()` y `free()`. En el ejemplo se toma memoria dinámica para almacenar un valor entero (se puede acceder a esta porción de memoria dinámica por medio del puntero `p_numero`).

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

int main()
{
    int *p_numero; /* Declaración de un puntero a entero */

    /* Asignación de memoria dinámica */
    p_numero = malloc(sizeof(int));
    if (p_numero == NULL)
    {
        printf("No se pudo tomar memoria para el puntero!\n");
        return 1;
    }

    /* Indirección del puntero y asignación de un valor */
    *p_numero = 50;

    /* Lectura del valor apuntado */
    printf("El puntero apunta al valor %d\n", *p_numero);

    /* Liberación de la memoria dinámica */
    /* No queremos "memory leaks" en nuestro programa, verdad? */
    free(p_numero);

    return 0;
}
```

Analicemos la sentencia siguiente:

```
p_numero = malloc(sizeof(int));
```

En esta llamada a `malloc()` estamos solicitando al sistema operativo una cantidad `sizeof(int)` de bytes. La función `sizeof()` retorna el tamaño del tipo de dato especificado como parámetro y es fundamental para la portabilidad de un programa. Si bien en los casos típicos un entero ocupa 4 bytes, podría existir una arquitectura donde un entero ocupe, por ejemplo, 8 bytes. Al utilizar la función `sizeof()` nos desentendemos del problema relacionado con los tamaños de los tipos.

Mediante el uso de `malloc()` es posible solicitar una cantidad arbitraria de bytes. Así, si el programa toma (de manera dinámica) 4 bytes, podría almacenar allí un valor entero o 4 caracteres, o 4 valores booleanos, entre otros ejemplos. De la misma manera, si se solicitan 100 bytes, en esa porción de memoria podrían alojarse 25 enteros, 100 caracteres, 25 valores de punto flotante, etc. O sea que la memoria dinámica asignada es una porción "genérica" capaz

de alojar tantos datos como quepan en ella. Lo más importante de esta reflexión es poder construir arreglos dinámicos; en el ejemplo siguiente se construye un vector dinámico de enteros, se lo inicializa y se lo muestra por pantalla:

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

#define TAMANIO_ARREGLO 10

int main()
{
    int *numeros; /* Declaración de un puntero a entero */
    int i;

    /* Asignación de memoria dinámica */
    numeros = malloc(TAMANIO_ARREGLO * sizeof(int));
    if (numeros == NULL)
    {
        printf("No se pudo tomar memoria para el puntero!\n");
        return 1;
    }

    /* Inicialización del arreglo dinámico */
    for (i=0; i<TAMANIO_ARREGLO; i++)
        numeros[i] = i;

    /* Visualización del arreglo dinámico */
    for (i=0; i<TAMANIO_ARREGLO; i++)
        printf("%d ", numeros[i]);
    printf("\n");

    /* Liberación de la memoria dinámica */
    free(numeros);

    return 0;
}
```

Uno de los puntos más importantes en este ejemplo es que, para construir un arreglo de enteros, se utiliza un puntero a un entero. La explicación a esto es simple: No importa el tamaño que tenga el arreglo, la forma de acceder a él siempre es a partir del primer elemento. Esta idea se ilustra en la figura siguiente:

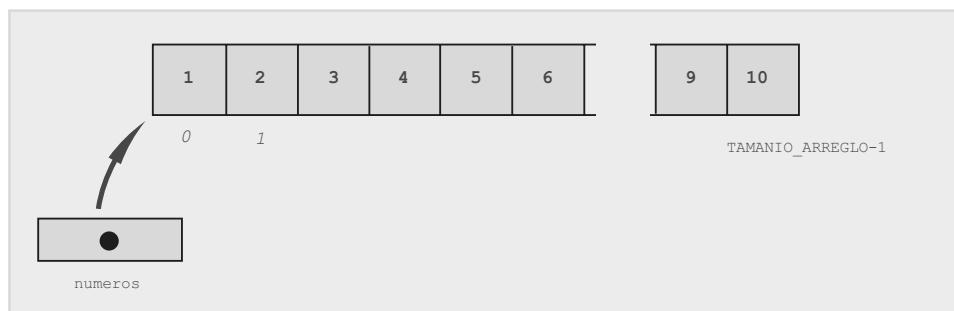


Fig. 10-5. Modo de acceder a un arreglo mediante un puntero al primer entero.

Así, a partir de un puntero al primer entero es posible acceder a los elementos que siguen. El programa (o el programador, para ser más precisos) debe conocer la longitud del arreglo en cuestión, y no leer o escribir más allá del último elemento.

El segundo punto importante para remarcar es el uso de la función malloc ():

```
numeros = malloc(TAMANIO_ARREGLO * sizeof(int));
```

Nótese que la cantidad de memoria dinámica solicitada es TAMANIO_ARREGLO enteros. Si TAMANIO_ARREGLO es 10 y se asumen enteros de 4 bytes, el sistema operativo estaría asignando 40 bytes de memoria dinámica.

Ahora que contamos con una idea básica de cómo manejar memoria dinámica en C, veamos cómo hacerlo en Pascal. Equivalentes a las funciones malloc () y free () de C, en Pascal se cuenta con los procedimientos siguientes:

```
new(var ptr: ^tipo_base)
dispose(var ptr: ^tipo_base)
```

El procedimiento new () toma memoria dinámica y fija el puntero ptr (recibido como parámetro) apuntando al inicio del bloque de memoria en cuestión. A diferencia de malloc (), en este caso no se especifica la cantidad de memoria solicitada; el sistema operativo "deduce" esto a partir del tipo de dato tipo_base al que apunta el puntero (por ejemplo, si el puntero pasado por parámetro apunta a un carácter, se asignará la memoria necesaria para almacenar ese tipo de dato). El procedimiento new () sólo puede utilizarse con punteros con tipo.

El procedimiento dispose () libera la porción de memoria dinámica a la que apunta ptr. Equivale a la función free () de C.

En el ejemplo siguiente (que ya se mostró antes, para el caso del lenguaje C) se toma memoria dinámica para almacenar un valor entero:

```
program punteros;
var p_numero: ^integer; { Declaración de un puntero a entero }
begin
{ Asignación de memoria dinámica }
new(p_numero);
if (p_numero = NIL) then
begin
writeln('No se pudo tomar memoria para el puntero!');
exit;
end;
{ Indirección del puntero y asignación de un valor }
p_numero^ := 50;
{ Lectura del valor apuntado }
writeln('El puntero apunta al valor ', p_numero^);
{ Liberación de la memoria dinámica }
{ No queremos "memory leaks" en nuestro programa, verdad? }
dispose(p_numero);
end.
```

La constante NIL de Pascal equivale a la constante NULL de C. En el caso del procedimiento new(), éste establecerá NIL en el puntero recibido como parámetro en caso de no poder tomar la memoria requerida.

10. 4 Punteros sin tipo.

Hasta este punto, en los programas dados como ejemplo, los punteros son con tipo; esto es, en la declaración del puntero se especifica a qué tipo de dato apuntará. Un enfoque más flexible es el uso de punteros “genéricos” (sin tipo). Básicamente, un puntero genérico es capaz de apuntar a cualquier tipo de dato.

En C, un puntero genérico se declara de la manera siguiente:

```
void *<nombre variable>;
```

Para ilustrar este concepto se presenta el ejemplo que sigue, donde un único puntero genérico (p_genérico) se utiliza para almacenar una cadena de caracteres (en la primera parte del programa) y un valor entero (en la segunda parte):

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

#define TAMANIO_CADENA 20

int main()
{
    void *p_genérico; /* Declaración de un puntero genérico */

    /* Asignación de memoria dinámica para una cadena */
    p_genérico = malloc(TAMANIO_CADENA * sizeof(char));
    if (p_genérico == NULL)
    {
        printf("No se pudo tomar memoria para el puntero!\n");
        return 1;
    }

    /* Se inicializa la cadena y se la muestra en pantalla */
    strcpy(p_genérico, "Hola Mundo!");
    printf("Valor apuntado por el puntero: %s\n",
           (char*)p_genérico);

    /* Liberación de la memoria dinámica */
    free(p_genérico);

    /* Asignación de memoria dinámica para un entero */
    p_genérico = malloc(sizeof(int));
    if (p_genérico == NULL)
    {
        printf("No se pudo tomar memoria para el puntero!\n");
        return 1;
    }

    /* Se inicializa el entero y se lo muestra en pantalla */
    *(int*)p_genérico = 123;
```

```

printf("Valor apuntado por el puntero: %d\n",
      *(int*)p_generico);

/* Liberación de la memoria dinámica */
free(p_generico);

return 0;
}

```

El uso de un puntero genérico es análogo al de un puntero con tipo. Sin embargo, cuando se accede a él, es necesario brindar información adicional respecto del tipo de dato en cuestión. Por ejemplo, en la sentencia



```
* (int*)p_generico = 123;
```

se ha casteado el puntero, para que se lo vea como un puntero a entero (`int*`). No es posible indireccionar un puntero genérico sin especificar el tipo de dato; al menos en esta instancia, el compilador necesita saber de qué tipo es el dato leído o escrito.

En el caso de Pascal, la declaración de punteros genéricos debe ajustarse a la sintaxis siguiente:

```
var <nombre variable>: pointer;
```

Los procedimientos `new()` y `dispose()` no pueden utilizarse con punteros genéricos en Pascal; en cambio, se proveen los siguientes:

```

getmem(var ptr: pointer; size: longint)
freemem(var ptr: pointer; size: longint)

```

Debido a que el puntero genérico carece de tipo, entonces es necesario indicar el tamaño de memoria solicitado o liberado. El mismo ejemplo dado antes para C se presenta a continuación para el caso de Pascal:

```

program punteros;

const TAMANIO_CADENA = 20;

var p_generico: pointer; { Declaración de un puntero genérico }

begin

{ Asignación de memoria dinámica para una cadena }
getmem(p_generico, TAMANIO_CADENA);
if (p_generico = NIL) then
begin
  writeln('No se pudo tomar memoria para el puntero!');
  exit;
end;

{ Se inicializa la cadena y se la muestra en pantalla }
string(p_generico^) := 'Hola Mundo!';
writeln('Valor apuntado por el puntero: ',
       string(p_generico^));

{ Liberación de la memoria dinámica }
freemem(p_generico, TAMANIO_CADENA);

```

Castear un puntero (o una variable general) consiste en convertir de manera explícita su tipo de dato. De esta forma se obliga al compilador a tratar al puntero o variable como si fuera de otro tipo.



```

{ Asignación de memoria dinámica para un entero }
getmem(p_generico, sizeof(integer));
if (p_generico = NIL) then
begin
  writeln('No se pudo tomar memoria para el puntero!');
  exit;
end;

{ Se inicializa el entero y se lo muestra en pantalla }
integer(p_generico^) := 123;
writeln('Valor apuntado por el puntero: ',
       integer(p_generico^));

{ Liberación de la memoria dinámica }
freemem(p_generico, sizeof(integer));

end.

```

Nótese que, como sucede en el caso de C, los punteros genéricos de Pascal también requieren que se los castee:

```

string(p_generico^) := 'Hola Mundo!';
writeln('Valor apuntado por el puntero:', string(p_generico^));

```

10.5 Aritmética de punteros.

Como se desarrolló al principio del capítulo, los punteros son valores enteros. Así, es posible aplicar sobre ellos algunas operaciones aritméticas como la suma y la resta. Dado un puntero P apuntando a un tipo de dato T , sumar una unidad a P significa que el puntero se incrementará en el tamaño del tipo T . Por ejemplo, si P es un puntero a entero (4 bytes), incrementarlo en una unidad hace que la dirección alojada en P se incremente en 4. Esta forma de operación facilita el uso de punteros como índices de arreglos. Por ejemplo, si el puntero P apunta al elemento i de un arreglo de enteros, la operación $P+1$ incrementa la dirección de memoria en 4 bytes y, de esta manera, P pasa a apuntar al elemento $i+1$, como se ilustra en la figura siguiente:

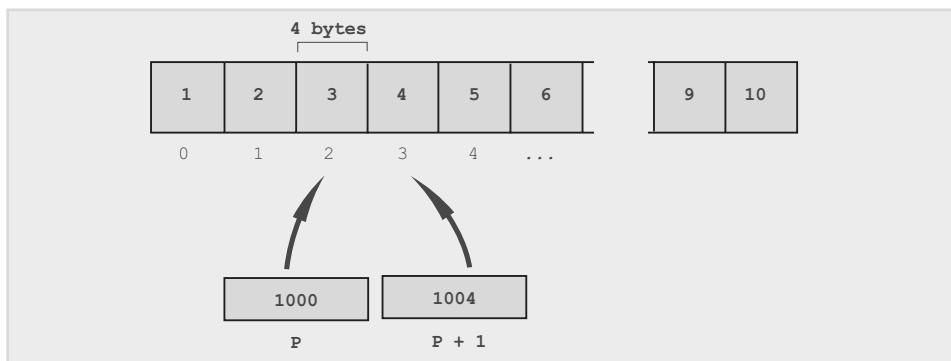


Fig. 10-6. Aplicación de operaciones aritméticas en punteros.

En el fragmento de código que sigue se ilustra el uso de aritmética de punteros para recorrer de manera secuencial una cadena de caracteres:

```
#include <stdio.h>

int main()
{
    char cadena[] = "Hola Mundo!";
    char *p_char;

    for (p_char = &cadena[0]; *p_char != '\0'; p_char++)
        printf("%c", *p_char);
    printf("\n");

    return 0;
}
```

En el caso de Pascal, algunos compiladores soportan aritmética de punteros, como es el caso de FreePascal.

10.6 Punteros y arreglos.

Los arreglos y los punteros están estrechamente relacionados. El nombre de un arreglo es un valor constante igual a la dirección de memoria de su primera posición. Por ejemplo, si `ptr` es un puntero a un tipo *T*, y `mi_arreglo` es un arreglo de elementos de tipo *T*, entonces es lícita la expresión siguiente:

```
ptr = mi_arreglo;
```

En este caso, la dirección de memoria del primer elemento de `mi_arreglo` es asignada al puntero `ptr`. Esto tiene una implicación muy fuerte, y es que a partir de esta asignación, tanto `ptr` como `mi_arreglo` son equivalentes, al igual que las expresiones que siguen:

```
mi_arreglo[i] = ... o ptr[i] = ...
```

Cuando el compilador encuentra la expresión `ptr[i]`, la traduce como `ptr+i`, esto es, una simple operación de aritmética de punteros. Si se toma de nuevo el ejemplo dado antes, donde se recorre una cadena mediante aritmética de punteros, se muestra a continuación:

```
#include <stdio.h>

int main()
{
    char cadena[] = "Hola Mundo!";
    char *p_char;
    int i;
    p_char = cadena; /* o también: p_char = &cadena[0] */

    for (i = 0; p_char[i] != '\0'; i++)
        printf("%c", p_char[i]);
    printf("\n");

    return 0;
}
```

Nótese que si bien `p_char` es una variable atómica (un puntero a `char`), en el bucle `for` se la trata como a un arreglo. Más aún, como el nombre de un arreglo es también la dirección de memoria del primer elemento, entonces el nombre del arreglo puede manipularse como un puntero:

```
#include <stdio.h>

int main()
{
    char cadena[] = "Hola Mundo!";
    int i;

    for (i = 0; *(cadena+i) != '\0'; i++)
        printf("%c", *(cadena+i));
    printf("\n");

    return 0;
}
```

Esta relación estrecha entre punteros y arreglos es de una importancia significativa y se exhorta al lector a que no avance hasta no entender con claridad este concepto junto con el de aritmética de punteros.

10.7 Punteros a funciones.

Se sabe que para que una función se pueda ejecutar, su código debe estar en memoria. Cuando el compilador traduce el código fuente a código objeto, también establece un punto de entrada a cada función, que no es más que la dirección de memoria donde esa función se encuentra alojada. Así, durante el tiempo de ejecución, si se llama a una rutina X , el control continúa a partir de la dirección de memoria de X (en este caso se indica que se transfiere el control a la función).

Dado que cada función tiene una dirección de memoria asociada, entonces es posible declarar un puntero que “apunte” a ella. Básicamente, y de manera análoga a lo que sucede con los arreglos, el nombre de una función en lenguaje C guarda la dirección de memoria donde ella se encuentra alojada. Por lo tanto, si se tienen las siguientes declaraciones:

```
int mi_funcion();           /* 1 */
int (*p_funcion)();         /* 2 */
```

en (1) se declara una función que devuelve un entero y en (2), un puntero a una función que devuelve un entero. Por lo tanto, la siguiente asignación es posible:

```
p_funcion = mi_funcion;
```

De esta forma puede pasarse una función, como argumento, a otra función. Esto puede ser útil para reutilizar código o hacer más flexible una rutina.

En este capítulo se brindó una visión amplia y general sobre el concepto y el uso de memoria dinámica y punteros. Son conceptos críticos y sensibles en la formación de programadores. Por esta razón se solicita al lector que, de haber algún concepto de este capítulo que no le haya quedado claro, no avance, sino, por el contrario, trate primero de aclarar las dudas.

10.8 Resumen.

La memoria principal de una computadora es el recurso del que dispone un programador para almacenar los datos que son manipulados por el programa. El uso de esta memoria puede hacerse según dos esquemas: De manera estática o dinámica. En el primer caso, el espacio en memoria se solicita en el instante en que el programa arranca su ejecución, y se libera cuando

termina. En el segundo caso, la memoria se solicita y se libera en el punto (durante la ejecución del programa) donde se hará uso de ella.

Manejar la memoria estáticamente tiene como ventaja que el costo de la asignación y la liberación sólo se “paga” una vez, al principio y al final de la ejecución. Además, durante el arranque, el programa se asegura contar con toda la memoria que necesitará durante su ejecución. Sin embargo, este esquema es demasiado conservador; es posible hacer un uso más racional de la memoria si se la asigna de manera dinámica sólo cuando se la necesita.

La gestión de la memoria dinámica corresponde al sistema operativo. Así, un proceso (en tiempo de ejecución) solicita al sistema operativo una cantidad determinada de memoria dinámica. Éste tomará memoria del *heap* y devolverá al proceso la **dirección de memoria** del primer byte de este bloque asignado. Esta dirección se almacena en una variable capaz de alojar direcciones de memoria: Un puntero. Es importante que el lector entienda que un puntero puede “apuntar” a una porción de memoria asignada dinámicamente, pero que también puede utilizarse para apuntar a una variable o a una estructura estática. También debe quedar en claro que una cosa es el **contenido del puntero** (una dirección de memoria “X”) y otra diferente es el **dato que está almacenado en esa dirección de memoria “X”**.

En cuanto a los punteros, podemos distinguir dos clases: Tipados y genéricos. En el primer caso, el puntero sólo puede apuntar a un tipo de dato en particular (por ejemplo, un puntero a entero). En el segundo caso, el puntero no está ligado a ningún tipo de dato.

Dado que las direcciones de memoria son números enteros, es posible aplicar sobre los punteros algunas operaciones aritméticas, como la suma y la resta. Dado un puntero P apuntando a un tipo de dato T , sumar una unidad a P significa que el puntero se incrementará en el tamaño del tipo T . Esta técnica, denominada aritmética de punteros, facilita el uso de punteros como índices de arreglos, y es soportada por todos los compiladores de C y por algunos compiladores de Pascal.

Los punteros y los arreglos mantienen una estrecha relación. El nombre de un arreglo es un valor constante igual a la dirección de memoria de la primera posición de éste. Así, si ptr es un puntero a un tipo T , y mi_arreglo es un arreglo de elementos de tipo T , entonces la expresión

```
ptr = mi_arreglo;
```

es válida y, a partir de entonces, ptr y mi_arreglo son equivalentes, y también son equivalentes las siguientes expresiones:

```
mi_arreglo[i] = ... o      ptr[i] = ...
```

Cuando el compilador encuentra la expresión $\text{ptr}[i]$, la traduce como $\text{ptr}+i$; esto es, una simple operación de aritmética de punteros.

Por último, en este capítulo se hace una introducción al concepto de puntero a función. Durante la ejecución de un programa el código de cada subrutina reside en memoria principal, y cada una de ellas se alojará a partir de una dirección de memoria determinada. Por lo tanto, es posible declarar un puntero que “apunte” a la función. Básicamente, y de manera análoga a lo que sucede con los arreglos, el nombre de una función en lenguaje C guarda la dirección de memoria donde ella se encuentra alojada.

10.9 Problemas propuestos.

- 1) Escribir un programa que efectúe las siguientes operaciones:
 - a) Declarar las variables enteras largas `value1` y `value2`, e inicializar `value1` a 200 000.
 - b) Declarar la variable `lPtr` como apuntador a un tipo `long`.
 - c) Asignar la dirección de la variable `value1` a la variable de apuntador `lPtr`.
 - d) Imprima el valor al que apunta `lPtr`.
 - e) Asígnele a la variable `value2` el valor al que apunta `lPtr`.
 - f) Imprima el valor de `value2`.
 - g) Imprima la dirección de `value1`.
 - h) Imprima la dirección almacenada en `lPtr`. ¿Es igual el valor impreso que la dirección de `value1`?

- 2) Crear un programa que calcule el valor de la intensidad que pasa a través de una resistencia dada, cuando se le aplica un voltaje determinado.

El programa deberá estar dividido en las siguientes funciones:

- **explicar_programa ()**

Esta función mostrará una introducción del programa por la pantalla.

- **obtener_valores ()**

Esta función pedirá los valores para la resistencia y el voltaje, los cuales se pasarán por referencia al programa principal.

- **calcular ()**

Esta función efectuará el cálculo de la intensidad a partir de la resistencia y el voltaje aplicado.

- **imprimir_respuesta ()**

Esta función se encargará de mostrar un mensaje con los resultados.

- 3) Crear una función que intercambie el contenido de dos variables. Para ello se pasarán como parámetros las direcciones de las variables.

Para probar la función escribir un programa que pida los datos por pantalla y muestre los contenidos después de llamar a la función.

- 4) Crear un programa que lea un número determinado (< 100) de números reales introducidos por teclado y los almacene en un vector para mostrarlos luego en orden inverso.

Nota: Para recorrer el arreglo se deberá usar aritmética de punteros en lugar de emplear los índices del arreglo.

- 5) Escribir una función que, tras pedir un día de la semana (de 1 a 7), devuelva un puntero a cadena con el nombre del día. La función contendrá un arreglo de apuntadores a cadena.

Para probar la función se realizará un programa que pida un día de la semana en número y escriba el día de la semana en letra.

- 6) Escribir un programa que inicialice una cadena con una palabra cualquiera. El programa deberá obtener la dirección de la primera letra de la cadena. Una vez sabida esta dirección la mostrará por pantalla y realizará un bucle dando tres oportunidades para que el usuario introduzca la dirección de la tercera letra de la cadena. En caso de no introducirla bien después de los tres intentos, deberá sacar un mensaje que indique cuál es la dirección correcta.

10.10 Problemas resueltos.

- 1- Use punteros para implementar pasajes de parámetros por referencia. Este programa debe permitir cargar los datos de un alumno en un registro. El pasaje de este registro como parámetro de la función '`ingresar_datos()`' debe llevarse a cabo por referencia, haciendo uso de la dirección de memoria de él.

Solución en lenguaje C

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

/* Tipo registro para almacenar un alumno */
typedef struct {
    long padron;
```



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

```
char      nombre[50];
float     notas[3];
} t_alumno;

void ingresar_datos(t_alumno *alumno)
{
    printf("Ingrese los datos del alumno:\n\n");
    printf("Padrón: ");
    scanf("%ld", &(alumno->padron));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("Nombre: ");
    fgets(alumno->nombre, 50, stdin);
    if (alumno->nombre[strlen(alumno->nombre)-1] == '\n')
        alumno->nombre[strlen(alumno->nombre)-1] = '\0';
    else
        while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("Nota 1: ");
    scanf("%f", &(alumno->notas[0]));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("Nota 2: ");
    scanf("%f", &(alumno->notas[1]));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */

    printf("Nota 3: ");
    scanf("%f", &(alumno->notas[2]));
    while (getchar() != '\n'); /* Limpieza buffer de teclado */
}

int main()
{
    t_alumno alumno;
    float promedio;

    /* Observar el pasaje por referencia usando */
    /* la dirección del registro. */
    ingresar_datos( &alumno );

    promedio = (alumno.notas[0]+alumno.notas[1]+alumno.notas[2])/3;

    printf("\n");
    printf("Padrón: %ld\n", alumno.padron);
    printf("Nombre y apellido: %s\n", alumno.nombre);
    printf("Promedio de notas: %.2f\n", promedio);
    printf("\n");

    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM cap10_ej01;

{ Tipo registro para almacenar un alumno }
TYPE t_alumno = RECORD
    padron:      LONGINT;
    nombre:      STRING[50];
    notas:       ARRAY[1..3] OF REAL;
END;
t_ptr_alumno = ^t_alumno;

PROCEDURE ingresar_datos(alumno: t_ptr_alumno);

BEGIN
    WRITELN('Ingrese los datos del alumno:');
    WRITELN;
    WRITE('Padrón: ');
    READLN(alumno^.padron);
    WRITE('Nombre: ');
    READLN(alumno^.nombre);
    WRITE('Nota 1: ');
    READLN(alumno^.notas[1]);
    WRITE('Nota 2: ');
    READLN(alumno^.notas[2]);
    WRITE('Nota 3: ');
    READLN(alumno^.notas[3]);
END;

VAR alumno:   t_alumno;
    promedio: REAL;

{ Programa Principal }
BEGIN
    { Observar el pasaje por referencia usando }
    { la dirección del registro. }
    ingresar_datos( @alumno );
    promedio := (alumno.notas[1]+alumno.notas[2]+alumno.notas[3])/3;

    WRITELN;
    WRITELN('Padrón: ', alumno.padron);
    WRITELN('Nombre y apellido: ', alumno.nombre);
    WRITELN('Promedio de notas: ', promedio:4:2);
    WRITELN;

END.
```

2- Inicialice un tablero para el juego "la batalla naval". El tablero es una matriz bidimensional, donde cada casillero es un puntero a registro. Para cada celda de la matriz, la memoria se toma dinámicamente.

Este problema es el correspondiente al número 5 del capítulo 6 (es aconsejable tratar de identificar las diferencias).

Solución en lenguaje C

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

#define COLS      20
#define FILAS     10

typedef enum {ACORAZADO, CRUCERO, PORTAAVIONES} t_tipo;
typedef enum {FALSE, TRUE} t_bool;

typedef struct {
    t_bool      ocupado;
    t_tipo      tipo;
    t_bool      hundido;
} t_casillero;

/* 't_tablero' es una matriz de punteros a 't_casillero' */
typedef t_casillero* t_tablero[FILAS][COLS];

/* Prototipos */
void inicializar_tablero(t_tablero);
void mostrar_tablero(t_tablero);
void liberar_tablero(t_tablero);

int main()
{
    t_tablero tablero;

    inicializar_tablero(tablero);
    mostrar_tablero(tablero);
    liberar_tablero(tablero);

    return 0;
}

void inicializar_tablero(t_tablero tablero)
{
    int i,j;

    for (i=0; i<FILAS; i++)
        for (j=0; j<COLS; j++)
    {
        /* Se toma memoria dinámicamente para cada celda */
        tablero[i][j] = malloc( sizeof(t_casillero) );
        if (tablero[i][j] == NULL)
        {
            /* No se pudo tomar memoria dinámica. Se liberan las */
            /* posiciones que pudiesen haber sido asignadas hasta */
    }
}
```

```
/* este punto y se sale de la aplicación. */  
liberar_tablero(tablero);  
fprintf(stderr, "Error intentando tomar memoria dinámica.");  
exit(1);  
}  
  
/* Equivalente a: (*tablero[i][j]).ocupado = FALSE; */  
tablero[i][j]->ocupado = FALSE;  
}  
  
/* 1 acorazado */  
tablero[4][18]->ocupado = TRUE;  
tablero[4][18]->tipo = ACORAZADO;  
tablero[4][18]->hundido = FALSE;  
tablero[5][18]->ocupado = TRUE;  
tablero[5][18]->tipo = ACORAZADO;  
tablero[5][18]->hundido = FALSE;  
tablero[6][18]->ocupado = TRUE;  
tablero[6][18]->tipo = ACORAZADO;  
tablero[6][18]->hundido = FALSE;  
  
/* 2 cruceros */  
tablero[9][2]->ocupado = TRUE;  
tablero[9][2]->tipo = CRUCERO;  
tablero[9][2]->hundido = FALSE;  
tablero[9][3]->ocupado = TRUE;  
tablero[9][3]->tipo = CRUCERO;  
tablero[9][3]->hundido = FALSE;  
  
tablero[3][9]->ocupado = TRUE;  
tablero[3][9]->tipo = CRUCERO;  
tablero[3][9]->hundido = FALSE;  
tablero[4][9]->ocupado = TRUE;  
tablero[4][9]->tipo = CRUCERO;  
tablero[4][9]->hundido = FALSE;  
  
/* 3 portaaviones */  
tablero[1][1]->ocupado = TRUE;  
tablero[1][1]->tipo = PORTAAVIONES;  
tablero[1][1]->hundido = FALSE;  
  
tablero[6][12]->ocupado = TRUE;  
tablero[6][12]->tipo = PORTAAVIONES;  
tablero[6][12]->hundido = FALSE;  
  
tablero[8][19]->ocupado = TRUE;  
tablero[8][19]->tipo = PORTAAVIONES;  
tablero[8][19]->hundido = FALSE;  
}  
  
void mostrar_tablero(t_tablero tablero)  
{  
    int i,j;
```

```

printf("\t");
for (i=0; i<COLS+2; i++)
    printf("-");
printf("\n");

for (i=0; i<FILAS; i++)
{
    printf("\t|");
    for (j=0; j<COLS; j++)
    {
        if (!tablero[i][j]->ocupado) /* o: (*tablero[i][j]).ocupado */
            printf(" ");
        else
            switch ( (*tablero[i][j]).tipo ) /* o: tablero[i][j]->tipo */
            {
                case ACORAZADO:      printf("#");
                break;
                case CRUCERO:        printf("X");
                break;
                case PORTAAVIONES:   printf("@");
                break;
            }
    }
    printf("|\n");
}

printf("\t");
for (i=0; i<COLS+2; i++)
    printf("-");
printf("\n");
}

void liberar_tablero(t_tablero tablero)
{
    int i,j;

    for (i=0; i<FILAS; i++)
        for (j=0; j<COLS; j++)
            if (tablero[i][j] != NULL) /* Tiene memoria asignada. */
                free( tablero[i][j] );
}

```

Solución en lenguaje Pascal

```

PROGRAM cap10_ej02;

CONST COLS  = 20;
CONST FILAS = 10;

TYPE t_tipo = (ACORAZADO, CRUCERO, PORTAAVIONES);

TYPE t_casillero = RECORD
    ocupado: BOOLEAN;
    tipo:     t_tipo;

```

```
        hundido: BOOLEAN;
END;

t_ptr_casillero = ^t_casillero;

{ 't_tablero' es una matriz de punteros a 't_casillero' }
t_tablero = ARRAY[1..FILAS, 1..COLS] OF t_ptr_casillero;

PROCEDURE liberar_tablero(tablero: t_tablero);

VAR i,j: INTEGER;

BEGIN
  FOR i:=1 TO FILAS DO
    FOR j:=1 TO COLS DO
      IF (tablero[i][j] <> NIL) THEN { Tiene memoria asignada. }
        DISPOSE( tablero[i,j] );
  END;

PROCEDURE inicializar_tablero(VAR tablero: t_tablero);

VAR i,j: INTEGER;

BEGIN
  FOR i:=1 TO FILAS DO
    FOR j:=1 TO COLS DO
      BEGIN
        { Se toma memoria dinámicamente para cada celda }
        NEW( tablero[i,j] );
        IF (tablero[i,j] = NIL) THEN
          BEGIN
            { No se pudo tomar memoria dinámica. Se liberan las }
            { posiciones que pudiesen haber sido asignadas hasta }
            { este punto y se sale de la aplicación. }
            liberar_tablero(tablero);
            WRITELN('Error intentando tomar memoria dinámica.');
            HALT;
          END;
        tablero[i,j]^^.ocupado := FALSE;
      END;
      { 1 acorazado }
      tablero[4,18]^^.ocupado := TRUE;
      tablero[4,18]^^.tipo := ACORAZADO;
      tablero[4,18]^^.hundido := FALSE;
      tablero[5,18]^^.ocupado := TRUE;
      tablero[5,18]^^.tipo := ACORAZADO;
      tablero[5,18]^^.hundido := FALSE;
      tablero[6,18]^^.ocupado := TRUE;
      tablero[6,18]^^.tipo := ACORAZADO;
      tablero[6,18]^^.hundido := FALSE;

      { 2 cruceros }
      tablero[9,2]^^.ocupado := TRUE;
```

```

tablero[9,2]^ . tipo      := CRUCERO;
tablero[9,2]^ . hundido   := FALSE;
tablero[9,3]^ . ocupado   := TRUE;
tablero[9,3]^ . tipo      := CRUCERO;
tablero[9,3]^ . hundido   := FALSE;

tablero[3,9]^ . ocupado   := TRUE;
tablero[3,9]^ . tipo      := CRUCERO;
tablero[3,9]^ . hundido   := FALSE;
tablero[4,9]^ . ocupado   := TRUE;
tablero[4,9]^ . tipo      := CRUCERO;
tablero[4,9]^ . hundido   := FALSE;

{ 3 portaaviones }
tablero[1,1]^ . ocupado   := TRUE;
tablero[1,1]^ . tipo      := PORTAAVIONES;
tablero[1,1]^ . hundido   := FALSE;

tablero[6,12]^ . ocupado   := TRUE;
tablero[6,12]^ . tipo      := PORTAAVIONES;
tablero[6,12]^ . hundido   := FALSE;

tablero[8,19]^ . ocupado   := TRUE;
tablero[8,19]^ . tipo      := PORTAAVIONES;
tablero[8,19]^ . hundido   := FALSE;

END;

PROCEDURE mostrar_tablero(tablero: t_tablero);
VAR i,j: INTEGER;
BEGIN

  WRITE(' ');
  FOR i:=1 TO COLS+2 DO
    WRITE('-');
  WRITELN;

  FOR i:=1 TO FILAS DO
    BEGIN

      WRITE(' | ');
      FOR j:=1 TO COLS DO
        BEGIN
          IF (NOT tablero[i,j]^ . ocupado) THEN
            WRITE(' ')
          ELSE
            CASE (tablero[i,j]^ . tipo) OF
              ACORAZADO:      WRITE('#');
              CRUCERO:        WRITE('X');
              PORTAAVIONES:   WRITE('@');
            END;
        END;
      WRITELN(' | ');
    END;
  END;

```

```

WRITE(' ');
FOR i:=1 TO COLS+2 DO
  WRITE('-');
  WRITELN;
END;

VAR tablero: t_tablero;
{ Programa Principal }
BEGIN

  inicializar_tablero(tablero);
  mostrar_tablero(tablero);
  liberar_tablero(tablero);

END.

```

3- En lenguajes de programación como C y Pascal, las matrices se almacenan linealmente en memoria. Existen dos esquemas: Si se almacenan las filas una tras otra, hablamos de orden por filas (por ejemplo, en C y Pascal), y si se almacenan las columnas una tras otra, el orden es por columnas (Fortran). Esto significa que, en la práctica, una matriz puede recorrerse con un solo índice.

Tome una matriz de números enteros y busque el mayor. Para ello se pide hacer uso de un solo índice y aritmética de punteros.

Solución en lenguaje C

```

#include <stdio.h>

#define N 5
#define TAMANIO (N*N)

int main()
{
    int *ptr, mayor=0;
    int matriz[N][N] = { { 19, 83, 7, 39, 21},
                         { 66, 1, 39, 70, 12},
                         { 21, 54, 29, 91, 30},
                         { 14, 27, 4, 87, 39},
                         { 5, 48, 24, 13, 8} };

    /* Búsqueda del entero mayor. */
    /* La matriz se recorre linealmente con un puntero y */
    /* haciendo uso de aritmética de punteros. */

    /* 'ptr' apunta al primer elemento de la matriz */
    for (ptr = &matriz[0][0]; ptr < (&matriz[0][0] + TAMANIO); ptr++)
        if (*ptr > mayor)
            mayor = *ptr;

    printf("%d es el mayor número entero en la matriz.\n", mayor);

    return 0;
}

```

Solución en lenguaje Pascal

```

PROGRAM cap10_ej03;
CONST N = 5;
TYPE t_matriz = ARRAY[1..N, 1..N] OF INTEGER;

VAR ptr: ^INTEGER;
    mayor: INTEGER;
    matriz: t_matriz;
    i, j: INTEGER;

{ Programa Principal }
BEGIN
    mayor := 0;
    { La matriz se inicializa con enteros aleatorios positivos }
    RANDOMIZE;
    FOR i := 1 TO N DO
        FOR j := 1 TO N DO
            matriz[i,j] := RANDOM(100); { >= a 0 y < a 100}

    { Búsqueda del entero mayor. }
    { La matriz se recorre linealmente con un puntero y }
    { haciendo uso de aritmética de punteros. }

    { 'ptr' apunta al primer elemento de la matriz }
    ptr := @matriz[1][1];
    WHILE (ptr <= @matriz[N][N]) DO
        BEGIN
            IF (ptr^ > mayor) THEN
                mayor := ptr^;
            INC(ptr);
        END;
    WRITELN(mayor, ' es el mayor número entero en la matriz.');
END.

```

4- Ordene por burbujeo un arreglo de caracteres mediante aritmética de punteros.

Solución en lenguaje C

El arreglo de entrada se recibe como parámetro por línea de comandos.

```

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

/* Prototipo */
void ordenar(char datos[], int n);

/* Programa Principal */
int main(int argc, char *argv[])

```

```

{
    char *datos, *ptr;
    long n;

    if (argc != 2)
    {
        fprintf(stderr, "Modo de uso: %s <cadena>\n", argv[0]);
        exit(1);
    }

    datos = argv[1];
    n = strlen(datos);

    /* Ordenamiento */
    ordenar(datos, n);

    printf("Arreglo ordenado: ");
    for (ptr=datos; ptr<(datos+n); ptr++)
        printf("%c ", *ptr);
    printf("\n");

    return 0;
}

void ordenar(char datos[], int n)
{
    char *ptr1, *ptr2, aux;

    /* 'ptr1' comienza apuntando al último elemento del arreglo */
    /* y desciende hasta terminar apuntando al primer elemento. */
    /* 'ptr2' se mueve desde el primer elemento hasta el ante- */
    /* rior a 'ptr1'. */

    for (ptr1 = datos+n-1; ptr1 >= datos; ptr1--)
        for (ptr2 = datos; ptr2 < ptr1; ptr2++)
            if ( *ptr2 > *(ptr2+1) )
            {
                /* Intercambio */
                aux = *ptr2;
                *ptr2 = *(ptr2+1);
                *(ptr2+1) = aux;
            }
}
}

```

Solución en lenguaje Pascal

```

PROGRAM cap10_ej04;

{ El arreglo de datos debe pasarse por referencia: VAR }

PROCEDURE ordenar(VAR datos: STRING; n: INTEGER);

VAR ptr1, ptr2: ^CHAR;
    aux: CHAR;

BEGIN

```

```

{ 'ptr1' comienza apuntando al último elemento del arreglo }
{ y desciende hasta terminar apuntando al primer elemento. }
{ 'ptr2' se mueve desde el primer elemento hasta el ante- }
{ rior a 'ptr1'. }

ptr1 := @datos[n];

WHILE (ptr1 >= @datos[1]) DO
BEGIN
    ptr2 := @datos[1];
    WHILE (ptr2 < ptr1) DO
    BEGIN
        IF ( ptr2^ > (ptr2+1)^ ) THEN
        BEGIN
            { Intercambio }
            aux := ptr2^;
            ptr2^ := (ptr2+1)^;
            (ptr2+1)^ := aux;
        END;
        INC(ptr2);
    END;
    DEC(ptr1);
END;
END;

VAR datos: STRING;
n: INTEGER;
ptr: ^CHAR;

{ Programa Principal }
BEGIN

IF (PARAMCOUNT <> 1) THEN
BEGIN
    WRITELN('Modo de uso: ', PARAMSTR(0), ' <cadena>');
    EXIT;
END;

datos := PARAMSTR(1);
n := LENGTH(datos);

{ Ordenamiento }
ordenar(datos, n);

WRITE('Arreglo ordenado: ');
ptr := @datos[1];
WHILE (ptr <= @datos[N]) DO
BEGIN
    WRITE(ptr^, ' ');
    INC(ptr);
END;
WRITELN;

END.

```

5- En este problema resuelto se busca ejemplificar el uso de punteros a funciones.

Cree una rutina para el cálculo de promedios, que sea lo suficientemente genérica y capaz de calcular el promedio sobre cualquier otra función que reciba un entero como parámetro y retorne un real de doble precisión.

Para ejemplificar el uso de 'promedio()' construya dos funciones (lineal y cuadrática).

Para el problema en Pascal, si la compilación se lleva a cabo con FreePascal deberá usarse la opción -Mtp que habilita la compatibilidad con TurboPascal (lo cual permite el pasaje de funciones y/o procedimientos como parámetros de otras funciones y/o procedimientos).

Solución en lenguaje C

```
#include <stdio.h>

/* La función 'promedio()' es genérica, en el sentido de que */
/* es capaz de calcular el promedio de cualquier función      */
/* recibida como parámetro.                                     */

double promedio(double (*func)(int), int inicio, int fin)
{
    double acum = 0.0;
    int x;

    for (x = inicio; x < fin; x++)
        acum += (*func)(x);
    return acum / (fin - inicio);
}

/* Función lineal: f(x) = x */
double lineal(int x)
{
    return (double)x;
}

/* Función cuadrática: f(x) = x^2 */
double cuadratica(int x)
{
    return (double)x*x;
}

int main()
{
    printf("Promedio de f(x) = x (0 <= x < 10) : %.2f\n",
           promedio(lineal, 0, 10));
    printf("Promedio de f(x) = x^2 (4 <= x < 6) : %.2f\n",
           promedio(cuadratica, 4, 6));
    return 0;
}
```

Solución en lenguaje Pascal

```
PROGRAM cap10_ej05;

TYPE t_func = FUNCTION(x: INTEGER): DOUBLE;

{ La función 'promedio()' es genérica, en el sentido de que }
{ es capaz de calcular el promedio de cualquier función      }
{ recibida como parámetro.                                     }

FUNCTION promedio(func: t_func; inicio, fin: INTEGER): DOUBLE;
```

```

VAR acum: DOUBLE;
x: INTEGER;

BEGIN
  acum := 0;
  FOR x := inicio TO fin-1 DO
    acum := acum + func(x);

  promedio := acum / (fin - inicio);
END;

{ Función lineal: f(x) = x }
FUNCTION lineal(x: INTEGER): DOUBLE;

BEGIN
  lineal := x;
END;

{ Función cuadrática: f(x) = x^2 }
FUNCTION cuadratica(x: INTEGER): DOUBLE;

BEGIN
  cuadratica := x*x;
END;

{ Programa Principal }
BEGIN
  WRITELN('Promedio de f(x) = x (0 <= x < 10) : ', promedio(lineal, 0, 10):6:2);
  WRITELN('Promedio de f(x) = x^2 (4 <= x < 6) : ', promedio(cuadratica, 4, 6):6:2);

```

10.11 Contenido de la página Web de apoyo.



El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Lecturas adicionales:

Memorias, de Patricia Quiroga, es parte del libro *Arquitectura de Computadoras* de Alfaomega Grupo Editor (60 páginas). Agradecemos a su autora por permitir que su escrito sea parte de las lecturas complementarias de esta obra.



Autoevaluación.



Video explicativo (03:50 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas. *



Presentaciones. *

11

El proceso de compilación

Contenido

11.1 Introducción	292
11.2 El proceso de compilación.....	292
11.3 Preprocesamiento	292
11.4 Compilación	296
11.5 Enlace	297
11.6 Automatización del proceso de compilación .	298
11.7 Resumen.....	301
11.8 Problemas resueltos.....	302
11.9 Contenido de la página Web de apoyo.....	306

Objetivos

- Entender como es el proceso de compilación, el proceso de traducción de un código fuente a lenguaje maquina, para que pueda ser ejecutado por la computadora u ordenador.
- Conocer las etapas más importantes de la compilación (pre-procesamiento, generación de código y enlace).
- Aprender a automatizar la compilación, técnica que se torna relevante cuando los proyectos son de gran envergadura.



En la página Web de apoyo encontrará un breve comentario del autor sobre este capítulo.

11.1 Introducción.

En este punto del libro se espera que el lector haya incorporado los conceptos fundamentales que le permitan desarrollar algoritmos correctos para la resolución de problemas. Por eso es que nos permitiremos ver con mayor detalle temas relacionados con el ambiente de desarrollo para la implementación y la compilación de algoritmos. En particular, será de interés analizar por separado las etapas más importantes que comprenden el proceso de compilación (preprocesamiento, generación de código y enlace), y también se presentará una forma de automatizar ese proceso (mediante la herramienta `make`), en especial cuando los proyectos adquieran tamaños considerables e involucran varios archivos fuente, de cabecera o bibliotecas.

11.2 El proceso de compilación.

En el ámbito de las computadoras, los algoritmos se expresan mediante lenguajes de programación, como C, Pascal, Fortran o Java (entre muchos otros). Sin embargo, esta representación no es suficiente, ya que el microprocesador necesita una expresión mucho más detallada del algoritmo, que especifique en forma explícita todas las señales eléctricas que involucra cada operación. La tarea de traducción de un programa desde un lenguaje de programación de alto nivel hasta el lenguaje de máquina se denomina compilación, y la herramienta encargada de ello es el compilador. En la figura siguiente se pueden distinguir las etapas más importantes:

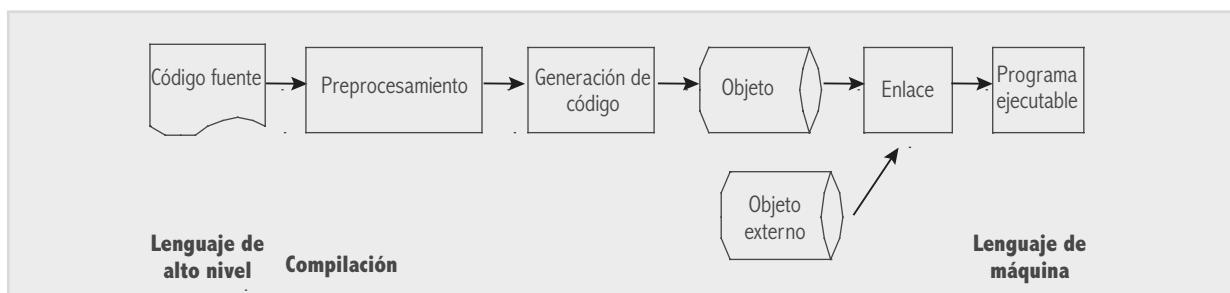


Fig. 11-1. Etapas más importantes de compilación.

A continuación se tratarán estas etapas de manera más extensa, con mayor atención en la primera de ellas: El preprocesamiento.

11.3 Preprocesamiento.

La primera etapa del proceso de compilación se conoce como preprocesamiento. En ocasiones a esta etapa ni siquiera se la considera parte de la compilación, ya que es una traducción previa y básica que tiene como finalidad “acomodar” el código fuente antes de que éste sea procesado por el compilador en sí. En este libro no haremos esa distinción.

El preprocesador modifica el código fuente según las directivas que haya escrito el programador. En el caso del lenguaje C, estas directivas se reconocen en el código fuente porque comienzan con el carácter `#`, y se utilizaron en gran medida a lo largo de este libro (por ejemplo, `#define...` o `#include...`). A continuación, veremos algunas de ellas:

```
#define, #undef, #error, #include, #if, #ifdef, #ifndef, #else,
#endif, #pragma.
```

11.3.1 Directivas #define #undef.

```
#define <nombre macro>    <expansión de la macro>
#define <nombre identificador>
```

La directiva `#define` crea un identificador con nombre `<nombre macro>` y una cadena de sustitución de ese identificador. Cada vez que el preprocesador encuentre el identificador de la macro, realizará la sustitución de éste por la cadena de sustitución. Esta última termina con un salto de línea. Cuando la cadena ocupa más de una línea, se utiliza la barra invertida para indicarle al compilador que omita el salto de línea. Algunos ejemplos:

```
#define MAX_PILA      100
#define VAL_ABS(a) ((a) < 0 ? -(a) : (a))
```

Por su parte, `#undef` elimina valores definidos por `#define`.

11.3.2 Directiva #error.

```
#error <mensaje de error>
```

Esta directiva fuerza al compilador a detener la compilación del programa. El mensaje de error no va entre comillas.

11.3.3 Directiva #include.

```
#include "archivo de cabecera"
#include <archivo de cabecera>
```

Esta directiva indica al preprocesador que incluya el archivo de cabecera indicado. Los archivos incluidos, a su vez, pueden tener directivas `#include`. Si el nombre del archivo está entre llaves `< y >` significa que se encuentra en alguno de los directorios que almacenan archivos de cabecera (los estándares del compilador y los especificados con la opción `-I` cuando se ejecuta la compilación). En el otro caso, significa que el archivo que se ha de incluir se encuentra en el directorio actual de trabajo (el directorio a partir del cual se ejecuta el compilador).

11.3.4 Directivas #if #ifdef #ifndef #else #endif.

```
#if <expresión constante>
  <secuencia de sentencias>
#endif

#ifndef <identificador>
  <secuencia de sentencias>
#endif

#define <identificador>
  <secuencia de sentencias>
#endif
```

Estas directivas permiten incluir o excluir condicionalmente partes del código durante la compilación. Si `<expresión constante>` es verdadera, se compila el código encerrado entre `#if` y `#endif`; en caso contrario, se ignora ese código.

En el segundo caso, si `<identificador>` está definido (por ejemplo, usando la directiva `#define`), entonces el código encerrado entre `#ifdef` y `#endif` es compilado.

En el tercer caso, si `<identificador> no` está definido, entonces el código encerrado entre `#ifndef` y `#endif` es compilado.

La directiva `#else` establece una alternativa:

```
#if <expresión constante>
    <secuencia de sentencias 1>
#else
    <secuencia de sentencias 2>
#endif
```

11.3.5 Directiva `#pragma`.

```
#pragma <nombre directiva>
```

Un pragma es una directiva que permite proveer información adicional al compilador. Depende del compilador utilizado y de la arquitectura donde se ejecutará la aplicación compilada.

En el caso de Pascal, el preprocessamiento también está gobernado a partir de un conjunto de directivas, de las cuales se detallarán sólo las que se utilizan con mayor frecuencia. A diferencia de C, una directiva en Pascal se define con la sintaxis `{$... }`. Algunas de ellas:

```
{$define}, {$undef}, {$ifdef}, {$else}, {$endif}, {$I}
```

11.3.6 Directivas `{$define}` `/{$undef}`.

```
{$define <nombre identificador>}
{$undef <nombre identificador>}
```

Al igual que en el caso de C, las directivas `{$define}` y `/{$undef}` en Pascal permiten definir y eliminar la definición de un identificador, respectivamente. A diferencia de C, no todos los compiladores permiten definir macros (un identificador con una cadena de sustitución asociada, que será reemplazada cada vez que se encuentre una ocurrencia del identificador).

11.3.7 Directivas `{$ifdef}` `{$else}` `{$endif}`.

```
{$ifdef <identificador>}
    <secuencia de sentencias 1>
{$else}
    <secuencia de sentencias 2>
{$endif}
```

La compilación condicional de código en Pascal está soportada mediante las directivas `{$ifdef}`, `{$else}` y `{$endif}`. En el ejemplo siguiente se cargan dos números de punto flotante y, en función de si el identificador `DOBLE_PRECISION` está definido o no, se computa su suma en precisión doble o simple:

```
program ejemplo;

{$define DOBLE_PRECISION}

{$ifdef DOBLE_PRECISION}
var num1, num2, resultado: double;
{$else}
var num1, num2, resultado: real;
{$endif}

begin

{$ifdef DOBLE_PRECISION}
  writeln('Ingrese dos números de precisión doble:');
{$else}
  writeln('Ingrese dos números de precisión simple:');
{$endif}

  readln(num1);
  readln(num2);

  resultado := num1 + num2;

{$ifdef DOBLE_PRECISION}
  writeln(num1:6:4, ' + ', num2:6:4, ' = ', resultado:6:4);
{$else}
  writeln(num1:6:2, ' + ', num2:6:2, ' = ', resultado:6:2);
{$endif}

end.
```

11.3.8 Directiva `{$I}`.

```
{$I+} / {$I-}
{$I <nombre de archivo>}
```

La directiva `{$I}` tiene dos usos. En el primer caso, cuando se la acompaña de los modificadores `+ y -`, permite habilitar e inhabilitar la comprobación de errores de entrada/salida, respectivamente. Su uso ya se explicó en el capítulo 7, ya que está estrechamente ligado al manejo de archivos. En el ejemplo siguiente se escribe sobre un archivo de texto. Para evitar que el programa se interrumpa con un error en la llamada a `append()` en caso de que el archivo no exista, se utilizan las directivas `{$I-}` y `{$I+}`, y se verifica el contenido de la variable predefinida `ioreresult` (si su valor es distinto de 0, significa que se produjo un error en la última operación de entrada/salida).

```

program ejemplo;

var archivo: text;

begin
  assign(archivo, 'mi_archivo.txt');

{${I-}
append(archivo);
{${I+
if (ioresult <> 0) then
  rewrite(archivo);

...
close(archivo);

end.

```

Con esta introducción se cubren los aspectos relacionados al preprocesamiento. La etapa que sigue, la compilación en sí, se explicará a grandes rasgos a continuación.

11.4 Compilación.

La compilación es el proceso que, en sí, traduce el lenguaje de alto nivel en lenguaje de máquina. Dentro de esta etapa pueden reconocerse, al menos, cuatro fases:

1. Análisis léxico.
2. Análisis sintáctico.
3. Análisis semántico.
4. Generación de código.

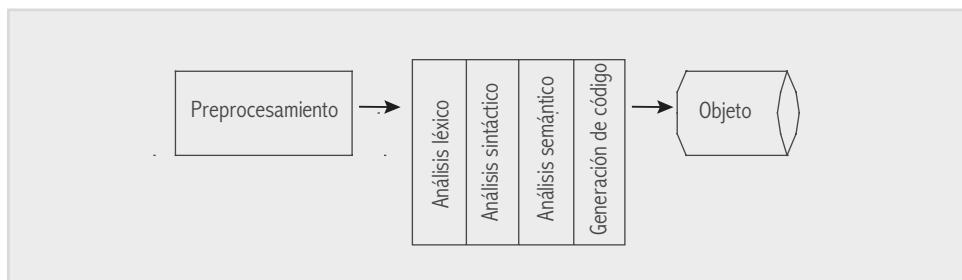


Fig. 11-2. Fases de la compilación.

El **análisis léxico** extrae del archivo fuente todas las cadenas de caracteres que reconoce como parte del vocabulario y genera un conjunto de *tokens* como salida. En caso de que parte del archivo de entrada no pueda reconocerse como lenguaje válido, se generarán los mensajes de error correspondientes.

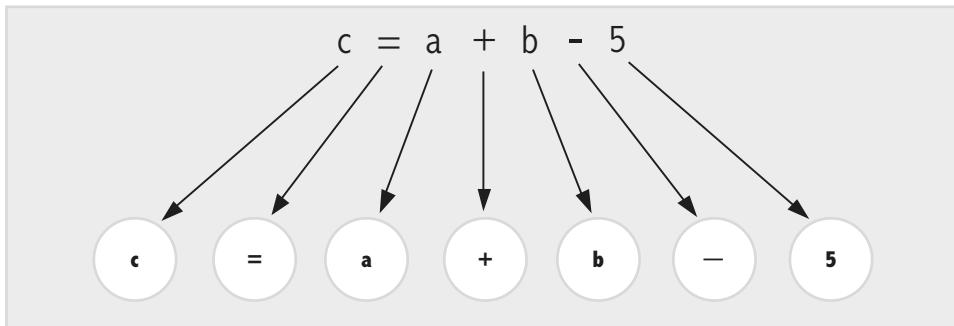


Fig. 11-3. Análisis léxico.

A continuación, durante el **análisis sintáctico**, se procesa la secuencia de *tokens* generada con anterioridad, y se construye una representación intermedia, que aún no es lenguaje de máquina, pero que le permitirá al compilador realizar su labor con más facilidad en las fases sucesivas. Esta representación suele ser un árbol.

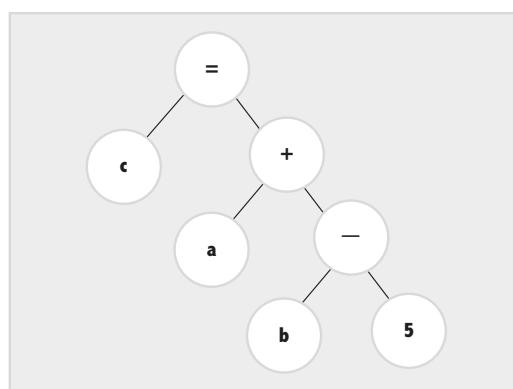


Fig. 11-4. Análisis sintáctico.

Durante el **análisis semántico** se utiliza el árbol generado en la fase previa para detectar posibles violaciones a la semántica del lenguaje de programación, como podría ser la declaración y el uso consistente de identificadores (por ejemplo, que el tipo de dato en el lado derecho de una asignación sea acorde con el tipo de dato de la variable destino, en el lado izquierdo de la asignación).

Por último, en la **generación de código** se transforma la representación intermedia en lenguaje de máquina (código objeto). En los casos típicos esta fase involucra mucho trabajo relacionado con la optimización del código, antes de generarse el lenguaje de máquina.

11. 5 Enlace

No siempre las aplicaciones se construyen de manera monolítica, a partir de un solo archivo fuente. En la práctica sólo se escribe una parte, y lo demás se toma de bibliotecas externas que, en la última etapa de la compilación, se enlazarán unas con otras para generar la aplicación ejecutable final. Ésta es, básicamente, la tarea del enlazador.

Los archivos objeto que se enlazan con nuestro programa se denominan bibliotecas externas que, por su parte, pueden haber sido construidas por nosotros mismos o pueden provenir de terceras partes (por ejemplo, las bibliotecas estándares del compilador). Una biblioteca, en este contexto, es una colección de funciones. Este tipo de archivos almacena el nombre de



En el caso particular del lenguaje C, cada biblioteca externa está acompañada de un archivo de cabecera ("header file") con extensión ".h". Se trata de un archivo de texto que contiene los encabezados (prototipos) de cada función de la biblioteca, además de otras definiciones de tipos o macros. Resumiendo, el archivo de cabecera contiene la parte "pública" de la biblioteca, a lo que el usuario tiene acceso. Es importante aclarar que, si bien es posible hacerlo, en los archivos de cabecera no debe incluirse código (solo definiciones de prototipos, tipos de datos o macros).

cada función, los códigos objeto de las funciones y la información de reubicación necesaria para el proceso de enlace. Entonces, en el proceso de enlace sólo se añade al código objeto el código de la función a la que se hizo referencia.

11.6 Automatización del proceso de compilación.

A lo largo de este libro se consideró únicamente la compilación de programas aislados, con un solo archivo fuente. Sin embargo, en la práctica, esta situación no suele ser la más habitual, en cambio, nuestros proyectos por lo general están conformados por varios archivos que, luego del proceso de compilación, darán lugar a una aplicación ejecutable. A medida que estos proyectos crecen en cuanto a cantidad de archivos, la compilación puede tomarse engorrosa. Felizmente, existe la posibilidad de automatizar el proceso de compilación mediante el uso de la herramienta make.

A medida que se escriben programas más largos, se observa que la compilación toma cada vez más tiempo. En general, cuando el programador corrige, modifica o mantiene un programa, trabaja en una sección acotada de él, y la mayor parte del código existente no cambia. La herramienta make es capaz de distinguir los archivos fuente que fueron alterados, lo que evita la compilación innecesaria de los que no se modificaron. Make construye proyectos sobre la base de comandos contenidos en un archivo de descripción o archivo de dependencias, comúnmente denominado makefile.

Para ilustrar el uso de la herramienta make, vamos a analizar un ejemplo simple. En este caso se cuenta con tres archivos propios y, además, se accede a la biblioteca estándar stdio.h. El archivo mi_aplicacion.c contiene el programa principal. Su trabajo es solicitar al usuario dos valores numéricos y sumarlos, para lo que invoca la función auxiliar mi_suma(), que reside en los archivos mis_funciones.h y mis_funciones.c (puede afirmarse que mis_funciones es una biblioteca externa). Además, el programa principal también hace uso de las funciones scanf() y printf(), pertenecientes a stdio. El código fuente de los archivos involucrados se muestra a continuación:

```
/* mi_aplicacion.c */
#include <stdio.h>
#include "mis_funciones.h"

int main()
{
    int a,b,c;

    scanf("%d", &a);
    scanf("%d", &b);

    c = mi_suma(a,b);

    printf("%d + %d = %d\n", a, b, c);

    return 0;
}
```

```
/* mis_funciones.h */
int mi_suma(int a, int b);
```

```
/* mis_funciones.c */
#include "mis_funciones.h"

int mi_suma(int a, int b)
{
    return a + b;
}
```

El archivo makefile define las reglas de dependencia entre estos archivos, además de especificar cómo deben compilarse (compilador que se debe usar, opciones de compilación, etc.):

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
```

```

mi_aplicacion: mi_aplicacion.o mis_funciones.o
$(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o

mi_aplicacion.o: mi_aplicacion.c mis_funciones.h
$(CC) $(CFLAGS) -c -o mi_aplicacion.o mi_aplicacion.c

mis_funciones.o: mis_funciones.c mis_funciones.h
$(CC) $(CFLAGS) -c -o mis_funciones.o mis_funciones.c

clean:
rm *.o mi_aplicacion

```

La primera línea define el compilador a utilizar (`CC=gcc`) y la segunda, las opciones de compilación (`CFLAGS=-Wall -ansi -pedantic`). Luego siguen tres reglas; la primera de ellas especifica cómo construir la aplicación `mi_aplicacion`, que depende de `mi_aplicacion.o` y `mis_funciones.o`. (Nótese que debajo de cada dependencia se encuentra el comando de compilación para ejecutar: `$(CC) -o mi_aplicacion...`). Las reglas escritas en el makefile conforman un grafo de dependencias:

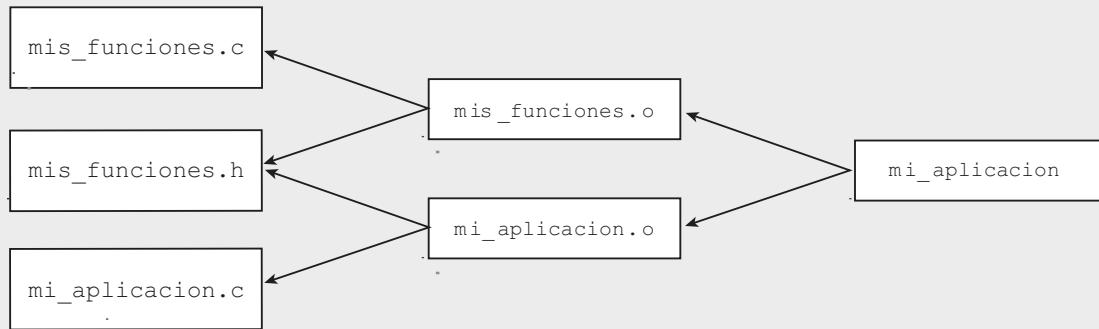


Fig. 11-5. Grafo de dependencias.

Al observar este grafo de dependencias puede deducirse que en caso de modificarse el archivo `mis_funciones.c`, sólo sería necesario regenerar el objeto `mis_funciones.o` y el ejecutable `mi_aplicacion`.

En el makefile cada dependencia se expresa utilizando el siguiente formato:

objetivo: dependencias (archivos fuentes, objetos)

comando de compilación (debe estar precedido por un tab)

donde objetivo es el archivo que será creado o actualizado cuando alguna dependencia (archivos fuente, objetos, etc.) sea modificada. Para realizar esta tarea se ejecutará el comando de compilación especificado (nótese que el comando de compilación debe estar indentado con un carácter de tabulación).

11.6.1 Herramienta make.

Una vez creado el makefile es posible ejecutar el comando `make` para llevar a cabo la compilación del proyecto. Esta herramienta buscará en el directorio de trabajo un archivo con nombre `Makefile`, lo procesará para obtener las reglas de dependencias y demás información y, por último, ejecutará la compilación de todos los archivos y el enlace final para generar la aplicación ejecutable.

11.6.2 Estructura del archivo makefile.

Con un poco más de detalle, se puede ver que un archivo `makefile` contiene:

- Bloques de descripción.
- Comandos.
- Macros.
- Reglas de inferencia.
- Directivas '.' (punto).
- Directivas de preprocesamiento.

Y otros componentes como caracteres comodines, nombres largos de archivos, comentarios y caracteres especiales.

11.6.3 Bloques de descripción.

Un bloque de descripción es una línea donde se expresan las dependencias correspondientes a cada objetivo:

objetivo: dependencias (archivos fuentes, objetos)

11.6.4 Comandos.

Un bloque de descripción o una regla de inferencia especifica un bloque de comandos para ejecutarse si la dependencia está desactualizada. Un bloque de comandos contiene uno o más comandos, cada uno en su propia línea. Una línea de comandos comienza con un carácter de tabulación. A continuación se muestra un ejemplo donde la primera línea es el bloque de descripción y la segunda, el bloque de comandos:

```
mi_aplicacion: mi_aplicacion.o mis_funciones.o
$(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o
```

11.6.5 Macros.

Las macros son definiciones que tienen un nombre y un valor asociado, y que pueden utilizarse a lo largo del `makefile`. El formato es:

`<nombre macro>=<valor>`

Algunos ejemplos:

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
```

La herramienta `make` reemplazará cada ocurrencia de `$ (CC)` y `$ (CFLAGS)` por el valor asociado. También es posible especificar el valor de la macro cuando se ejecuta `make`:

```
make 'CC=gcc'
```

Existen macros utilizadas internamente por la herramienta `make`, algunas de las cuales se muestran a continuación:

`CC`: Contiene el compilador C utilizado en el desarrollo. El default es `CC`.

`$@`: El nombre completo del archivo destino.

`$<`: El archivo fuente de la dependencia actual.

11.6.6 Reglas de inferencia.

Las reglas de inferencia proveen comandos para generar los archivos objetivo infiriendo sus dependencias (evita la necesidad de tener que especificar una regla por cada archivo que se ha de compilar). En el ejemplo siguiente se establece una regla de inferencia que determina que para generar un archivo objeto `.obj` (en ocasiones puede usarse la extensión `.o` en lugar de `.obj`, como se pudo apreciar en ejemplos anteriores) es necesario que exista un archivo fuente `.c` con igual nombre:

```
% .obj : %.c  
$(CC) $(CFLAGS) -c $(.SOURCE)
```

Esta regla de inferencia le indica a `make` cómo construir un archivo `.obj` a partir de un archivo `.c`. La macro predefinida `.SOURCE` es el nombre del archivo de dependencia inferido. A continuación, se presenta el archivo `makefile` expuesto antes, pero ahora utilizando reglas de inferencia:

```
CC=gcc  
CFLAGS=-Wall -ansi -pedantic  
  
mi_aplicacion: mi_aplicacion.o mis_funciones.o  
$(CC) -o mi_aplicacion mi_aplicacion.o mis_funciones.o  
  
%.obj : %.c  
$(CC) $(CFLAGS) -c $(.SOURCE)  
  
clean:  
rm *.o mi_aplicacion
```

En la sección de problemas correspondientes a este capítulo se podrán encontrar otros casos, un poco más elaborados, sobre el uso de `makefiles`. Aquí sólo se pretende dar una introducción básica y que el lector se familiarice con el uso de estas herramientas.

11.7 Resumen.

Luego de estudiar los conceptos fundamentales referidos a la construcción de algoritmos, en el último capítulo se trataron las cuestiones relacionadas con el proceso de compilación, que permite generar un programa ejecutable. Las etapas más importantes que pueden distinguirse en este proceso son el preprocessamiento, la compilación y el enlace.

El preprocessamiento, como su nombre lo indica, es una etapa previa que tiene como finalidad “acomodar” el código fuente antes de que éste sea procesado por el compilador. Para ese fin, el preprocessador modifica el código fuente según un conjunto de directivas que el programador puede incluir en distintos puntos del programa. La compilación condicional es un buen ejemplo: Si se cumple cierta condición, el preprocessador podría eliminar porciones del código del programa.

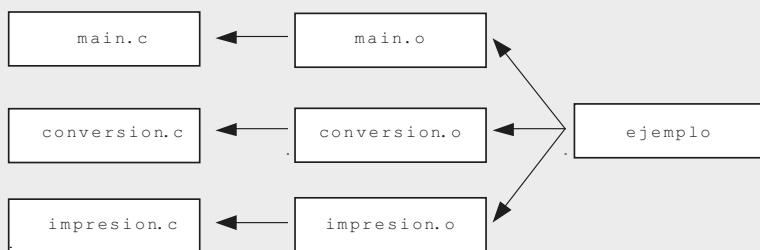
Durante la compilación se traduce el lenguaje de alto nivel en lenguaje de máquina. Dentro de esta etapa pueden distinguirse las siguientes fases: Análisis léxico, análisis sintáctico, análisis semántico y generación de código.

La última de las etapas, el enlace, se encarga de tomar el código de bibliotecas externas a las que nuestro programa podría estar haciendo referencia, y generar (a partir de nuestro código y el de las bibliotecas externas) el programa ejecutable.

En la práctica, nuestros proyectos suelen estar conformados por varios archivos que, luego del proceso de compilación, darán lugar a una aplicación ejecutable. A medida que estos proyectos crecen, en cuanto a cantidad de archivos, la compilación puede tornarse engorrosa. Felizmente, existe la posibilidad de automatizar el proceso de compilación mediante el uso de la herramienta `make`. Ésta es capaz de distinguir dentro de un proyecto cuáles son los archivos fuente que han sido alterados, y evitar la compilación innecesaria de los que no se modificaron. `Make` construye proyectos sobre la base de comandos contenidos en un archivo de descripción o archivo de dependencias, comúnmente denominado `makefile`.

11.8 Problemas resueltos.

1- Esta aplicación permite convertir kilómetros en millas. El programa principal se encuentra en el archivo `main.c`. La conversión se lleva a cabo llamando a la función `convertir()` en `conversion.c` (y cuyo prototipo está definido en el archivo de cabecera `conversion.h`). Por último, para imprimir el resultado se invoca a la función `imprimir_resultado()` en `impresion.c` (y cuyo prototipo está definido en el archivo de cabecera `impresion.h`). El grafo de dependencias es el siguiente:



En la página Web de apoyo encontrará el código fuente de los problemas resueltos en este capítulo.

El `makefile` se construyó usando reglas de inferencia, aprovechando que los archivos de dependencias (`main.c`, `conversion.c` e `impresion.c`) y los objetivos (`main.o`, `conversion.o` e `impresion.o`) tienen el mismo nombre.

Obsérvese que en el `makefile` se incluye una regla adicional cuyo objetivo es `clean`. Esta regla permite borrar todos los archivos generados en la compilación del proyecto, mediante el comando: `make clean`. Como puede verse, esta regla no tiene dependencia alguna y su comando asociado es `rm * .o ejemplo` (para eliminar todos los archivos con extensión `.o` y el ejecutable `ejemplo`).

El comando `rm` (remove) permite eliminar archivos en ambientes Unix/Linux; en el caso de Windows el comando equivalente es `del /`.

main.c

```
#include <stdio.h>
#include "conversion.h"
#include "impresion.h"

int main (void)
{
    int kms;
    float millas;

    printf("Conversión de kilómetros a millas\n");
    printf("-----\n\n");

    printf("Introduzca la cant. de kms a convertir: ");
    scanf("%i", &kms);

    millas = convertir(kms);
    imprimir_resultado(kms, millas);

    return 0;
}
```

conversion.h

```
#define FACTOR 0.6214

/* Prototipos */
float convertir(int);
```

conversion.c

```
#include "conversion.h"

float convertir(int kms)
{
    return FACTOR * kms;
}
```

impresion.h

```
/* Prototipos */
void imprimir_resultado(int, float);
```

impresion.c

```
#include <stdio.h>

void imprimir_resultado(int km, float millas)
{
    printf("%d km equivalen a %.2f millas", km, millas);
}
```

Makefile

```

CC=gcc
CFLAGS=-Wall -ansi -pedantic

ejemplo: main.o conversion.o impresion.o
        $(CC) -o ejemplo main.o conversion.o impresion.o

# REGLA DE INFERENCIA
# Esta forma es equivalente a: %.o : %.c
# La macro '$<' es el archivo fuente del cual el objetivo
# depende (por ejemplo, conversion.c).
.c.o:
        $(CC) $(CFLAGS) -c $<

clean:
        rm *.o ejemplo

```

2- Hasta el momento en todos los casos se asumió que los archivos fuente y de cabecera residen en el mismo directorio y que también los archivos generados (objetos y binario ejecutable) se ubicarán allí. Sin embargo, a medida que un proyecto crece en cuanto a cantidad de archivos, se hace necesario organizarlos en subdirectorios. En un proyecto típico podemos encontrar:

src/	Contiene los archivos fuente (.c).
include/	Contiene los archivos de cabecera (.h).
bin/	Contiene el binario ejecutable generado.

Tomando como caso el mismo proyecto que en el problema anterior, se muestra el archivo makefile utilizando la estructura de subdirectorios descripta.

Makefile

```

CC=gcc
CFLAGS=-Wall -ansi -pedantic

SRC_DIR=src
INCLUDE_DIR=include
BIN_DIR=bin

VPATH=$(SRC_DIR)

ejemplo: main.o conversion.o impresion.o
        $(CC) -o ejemplo main.o conversion.o impresion.o
        rm *.o
        mv ejemplo $(BIN_DIR)

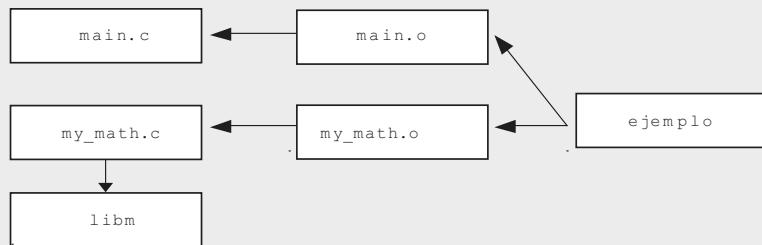
.c.o:
        $(CC) $(CFLAGS) -I$(INCLUDE_DIR) $< -c

clean:
        rm $(BIN_DIR)/ejemplo

```

La forma que tenemos de indicarle a `make` dónde buscar los archivos de dependencias (`.c`) es utilizar la macro `VPATH`. En nuestro caso, `VPATH` toma el valor `src` (o sea que los archivos fuente `.c` deberán buscarse en el subdirectorio `src/`). Además, también es necesario indicar al compilador dónde buscar los archivos de cabecera (ya que ahora no se encuentran en el mismo directorio que los archivos fuente, sino en el subdirectorio `include/`). Para ese fin se define la macro `INCLUDE_DIR` y se la utiliza con la opción `-I` del compilador. Por último, el binario ejecutable generado (`ejemplo`) se mueve al subdirectorío `bin/` usando el comando `mv` y todos los archivos objeto generados se eliminan (esto suele ser una práctica habitual cuando se hace un lanzamiento del proyecto, ya que el usuario final no necesita la existencia de los archivos objeto).

En este caso, una aplicación solicita al usuario dos valores enteros: Base y exponente, y muestra en pantalla el resultado $base^{exponente}$. La operación se realiza llamando a la función `potencia()` en el archivo `my_math.c`. Por su parte, esta función llama a `pow()`, que se encuentra implementada en la biblioteca matemática estándar de C (`libm`). El grafo de dependencias es el siguiente:

**main.c**

```

#include <stdio.h>
#include "my_math.h"

int main()
{
    int base, exponente;

    printf("Ingrese la base: ");
    scanf("%d", &base);
    printf("Ingrese el exponente: ");
    scanf("%d", &exponente);

    printf("%d elevado a la %d = %d\n", base, exponente, potencia(base,exponente));

    return 0;
}
  
```

my_math.h

```

/* Prototipo */
int potencia(int base, int exponente);
  
```

```

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

```
int potencia(int base, int exponente)
{
    return (int)pow(base,exponente);
}
```

Makefile

```
CC=gcc
CFLAGS=-Wall -ansi -pedantic
LIBS=-lm

ejemplo: main.o my_math.o
    $(CC) $(LIBS) -o ejemplo main.o my_math.o

math.o: math.c
    $(CC) $(CFLAGS) -c math.c

clean:
    rm *.o ejemplo
```

En este programa se hace uso de una biblioteca externa (`libm`), en la que se encuentra la implementación de la función `pow()`. Para indicar al compilador que el binario ejecutable debe enlazarse contra una biblioteca externa, se utiliza la opción `-l`. Además, se debe añadir el nombre de la biblioteca en cuestión, para lo cual se toma el nombre de ella y se quita el prefijo `lib`. Para el caso de `libm`, debe usarse: `-lm`. Para que esto quede más claro, en el archivo `makefile` se definió la macro:

```
LIBS=-lm
```

11.9 Contenido de la página Web de apoyo.

El material marcado con asterisco (*) sólo está disponible para docentes.



Mapa conceptual.



Simulación:

- El proceso de compilación.



Autoevaluación.



Video explicativo (02:21 minutos aprox.).



Código fuente de los ejercicios resueltos.



Evaluaciones propuestas.*



Presentaciones.*

Bibliografía

Aho, Alfred V., Ullman, Jeffrey D. y Hopcroft, John E. - *Data Structures and Algorithms* - Addison Wesley, EE.UU, 1983.

Aho, Alfred V., Hopcroft, John E. y Ullman, Jeffrey D. - *The Design and Analysis of Computer Algorithms* - Addison-Wesley, EE.UU 1974.

Echevarría, Adriana y López, Carlos Gustavo - *Elementos de Diseño y Programación con Ejemplos en C* - Nueva Librería, Argentina, 2005.

Garey, Michael R. y Johnson, David S. - *Computers and Intractability: a Guide to the Theory of NP-Completeness* - W. H. Freeman, EE.UU, 1979.

Kernighan, Brian W. y Ritchie, Dennis M. - *El Lenguaje de programación C*, 2^a ed. - Prentice Hall, Mexico, 1991.

Knuth, Donald . - *The Art of Computer Programming*, 2^a ed. - Addison-Wesley Professional, EE.UU, 1998.

Lewis, Harry R. y Papadimitriou , Christos.H (1978) - *The Efficiency of Algorithms- Scientific American*, páginas 96 a 108, enero de 1978.

López, Carlos Gustavo - *Algoritmia, Arquitectura de Datos y Programación Estructurada* - Nueva Librería, Argentina, 2003.

Papadimitriou , Christos.H. & Steiglitz, Kenneth. - *Combinatorial Optimization : Algorithms and Complexity* - Prentice-Hall, EE.UU, 1982.

Rheingold, Edward M., Nievergelt, Jurg, Deo, Narasingh - *Combinatorial Algorithms: Theory and Practice* - Prentice-Hall, EE.UU, 1982.

Índice analítico

- Ábaco 2
 Acceso directo 90, 104, 182, 189, 190, 191, 219, 220
 Acoplamiento 59, 74
 Acumulador 31
 Algoritmo 4, 43, 49, 292
 Algoritmos recursivos 248, 249, 253
Almacenamiento de datos 3
 Análisis 3, 7, 8, 24, 267, 296, 297, 302
 Análisis del problema 8, 7, 24
 ANSI 28, 37, 73, 181, 182, 183, 184
 Apareo de archivos 191
 Apertura de un archivo 182, 187
 Archivo de texto 10, 181, 183, 184, 185, 186, 190, 295, 298
 Archivos de acceso directo 189, 190, 191
 Arreglos 30, 41, 91, 92, 94, 96, 98, 99, 100, 101, 103, 104, 126, 129, 132, 134, 136, 150, 152, 156, 159, 180, 222, 269, 273, 274, 275, 276
 lineales 90, 91, 96, 98, 103, 129, 152, 156, 159, 194
 multidimensionales 90, 94, 100, 103
 ASCII 13, 27, 75, 181, 189, 194,
 auto 6
 Böhm, Corrado 5
 break 6, 15, 17, 18, 102, 175
 buffer 40, 153, 181, 182, 183, 184, 188, 194, 278
 Búsqueda 92, 93, 74, 127, 134, 222
 binaria 126, 129, 130, 132, 136, 217, 220
 secuencial 129, 130, 131, 132, 136, 147, 220
 C 4, 5, 6, 10, 12, 15, 19, 20, 21, 22, 23, 24
 Cadenas de caracteres 12, 26, 41, 68, 89, 100, 101, 103, 104, 296
 case 6, 15, 18, 21, 23, 102, 103
 Castear 272
 char 6, 13, 19, 26, 27, 28, 35, 36, 91, 99, 100, 101
 Cierre de un archivo 183, 188
 Clave
 candidata 218
 primaria 218, 219, 220, 221
 secundaria 218, 221
 Claves 216, 217, 218, 220, 222
 Cobol 4, 24
 Codificación 7, 8, 9, 13, 24, 27, 28, 121
 Código fuente 7, 10, 11, 275, 292, 298
 Cohesión 59, 74, 159
 Compilación 7, 9, 10, 11, 24, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302
 Compilador 9, 10, 11, 12, 18, 19, 292, 293, 294, 297, 298, 299, 301, 302
 Complejidad computacional 126
 const 6, 12, 20, 22, 29, 93, 99, 158, 183, 184, 185
 Contador 15, 31, 32, 34, 145
 continue 6, 17
 Controladores de versiones 11
 Cota
 ajustada asintótica 129
 inferior asintótica 128
 superior asintótica 127
 Creación 3, 187, 195
 Cuenta ganado 2
 Datos simples 5, 26, 27, 30, 90, 96
 Declaración
 de constantes 12
 de tipos 12, 91
 de variables 12, 20, 187
 Declaraciones 12, 62, 63, 64, 70, 75
 default 6, 15, 18, 102, 301
 Depuración 7, 9, 24
 Dijkstra, Edsger 5, 17, 24
 Diseño 2, 3, 5, 6
 del algoritmo 7, 8, 24
 Documentación 7, 10, 11, 24
 double 6, 13, 19, 26, 27, 28, 295
 else 6, 14, 20, 21, 23
 Enlace 9, 292, 297, 298, 300, 301, 302
 Enlazador 30, 297
 enum 6, 102, 173, 209, 280
 Errores en tiempo de ejecución 188
 Estructura
 estática 90, 91, 104
 homogénea 90, 104, 157
 lineal de acceso directo 90
 Estructuras de control 5, 20, 24, 91
 Expresión 14, 15, 17, 21, 31, 33, 34
 extern 6, 29, 30
 Fibonacci 248
 Fortran 4
 float 6, 10, 11, 12, 13, 26, 27, 28, 96, 97, 150, 151, 303
 for 6, 15, 17, 21, 92, 94
 Fuga de memoria 265
 Funciones 2, 10, 11, 12, 14, 17, 18, 19, 20, 22, 23, 183, 184, 188
 Función scanf() 39, 40, 41
 Garbage collector 265
 goto 6, 17
 Identificador 10, 31, 62, 218, 293, 294, 295
 if 6, 14, 20, 34, 35, 130
 Índices 111, 216, 218, 219, 220, 221, 222, 273, 276
 primarios 221
 secundarios 221
 y archivos 219
 inlining 73
 Instrucciones 2, 3, 4, 5, 7, 12, 20, 24, 72, 74, 180
 int 6, 13, 26, 27, 28, 30
 Intersección de archivos 193
 Iteración 14, 15, 16, 17, 21, 131, 188, 265
 Iterativa 7, 46, 47, 50, 249
 Jacopini, Giuseppe 5
 Laboratorios Bell 6
 Lectura 13, 37, 41, 70, 74, 90, 91, 103, 150, 180, 181, 182, 183, 184, 185, 187, 188, 189, 190, 194, 268, 270
 Lenguajes
 de alto nivel 3, 6, 13
 de bajo nivel 3, 5
 de programación 1, 3, 4, 5, 7, 24, 26, 32, 41, 65, 91, 248, 253, 265, 292
 Línea de comandos 67, 68, 69, 137, 207, 300
 Llamada a sistema 181
 long 6, 13, 26, 28, 152, 157, 189, 250, 253
 Macros 73, 182, 189, 294, 298, 300, 301
 main() 11, 12, 14, 19, 97, 154
 make 292, 298, 300, 301, 302
 makefile 298, 299, 300, 301, 302
 Manipulación de archivos 181, 182, 183, 188, 216, 222
 Máquina diferencial de Babbage 2
 Matrices 90, 103, 126, 136, 152, 159, 285
 Memoria 3, 10, 13, 26, 28, 31, 32, 39, 40, 41, 65, 67, 71, 75, 98, 104, 152, 154, 156, 159, 182, 184, 186, 188, 250, 251, 253, 264, 266, 274
 caché 29, 74
 dinámica 37, 91, 265, 267, 268, 269, 270, 271, 272, 273, 275, 276
 principal 30, 66, 69, 70, 74, 95, 96, 97, 101, 155, 180, 193, 194, 220, 221, 222, 275, 276
 RAM 70, 180
 Secundaria 70, 180, 220, 221, 222
 Métodos
 de búsqueda 126, 129, 222
 de ordenamiento 126, 132, 136, 222
 Mezcla 134, 135, 191, 192, 193, 194
 de archivos 222
 de arreglos 34, 136
 Modificación 191, 192
 Modularización 5, 20, 58, 59, 60, 73, 74
 Objeto 7, 9, 30, 129, 188, 216, 275, 297, 298, 299, 301
 externo 292
 Ocurrencia 101, 141, 294, 301
 Operadores
 aritméticos 32, 33, 42
 lógicos 34
 relacionales 34, 35, 42
 Ordenación 137
 por burbujeo 126, 132, 133, 136, 139, 144
 por inserción 133, 134
 por selección 133, 141
 Parámetros de funciones 104, 152, 186

- Pasaje
 por nombre 65
 por referencia 65, 66, 67, 104, 152, 154, 159
 por valor 65, 66, 67, 79
- Pascal 4, 5, 6, 10, 12, 15, 19, 20, 21, 22, 23, 24
- Persistencia 70, 180, 193
- Preprocesador 11, 12, 18, 37, 73, 302
- Preprocesamiento 292, 294, 296, 300, 301, 302
- Programación
 estructurada 1, 4, 5, 6, 8, 14, 17, 20, 24, 99, 130, 159
 modular 6, 8
- Programa ejecutable 9, 14, 30, 301, 302
- Pseudocódigo 31
- Punteros 31, 32, 37, 39, 66, 67, 98, 154, 182, 265, 267, 268, 270, 272, 273, 274, 276
 a funciones 275
 sin tipo 271
- read() 41, 188
- readln() 23, 41
- Recolector de basura 265
- Recursividad 5, 248, 252, 253
- register 6, 30
- Registro 30, 35, 150, 151, 153, 154, 155, 156, 157, 158, 159, 181, 187, 188, 189, 190, 191, 192, 194, 216, 217, 219, 220, 221, 222
- Registros jerárquicos 154, 155
- Reglas de inferencia 300, 301, 302
- restrict 6
- return 6, 17, 18, 38
- Ritchie, Dennis 6
- scanf() 19, 37, 39, 40, 41, 121, 165, 185, 298
- Secuencial 17, 129, 130, 131, 132, 134, 136, 181, 188, 216, 218, 219, 220, 222, 273
- Secuenciales 5, 14, 20, 24, 220
- Selección 14, 20, 21, 34, 126, 133, 136, 141, 218
- Selectivas 5, 14, 20, 24
- Sentencias 3, 4, 5, 12, 14, 17, 31, 127, 267, 293, 294
- short 6, 13, 26, 28
- signed 6, 13, 26, 28
- sizeof 6, 37, 268, 269, 270, 271, 273
- Software
 de aplicación 3
 del sistema 3
- stack overflow 72, 252
- static 6, 30
- struct 6, 150, 151, 152, 154, 155, 156, 157, 159, 168, 173
- Subprogramas 6, 8, 24, 58, 74, 96
- switch 6, 14, 15, 17, 18, 19, 21, 102, 175
- system call 181
- Tablas 41, 156, 157, 159, 180, 217
- Tipados 12, 26, 265, 276
- Tipos 12, 13, 19, 26, 27, 28, 30, 32, 37, 41, 91, 92
- typedef 6, 91, 102, 150, 151, 152, 154, 155, 156, 157
- Unión 136, 155, 156, 191, 193
- Uniones 155, 156
- Unix 302
- unsigned 6, 13, 26, 27, 28, 35, 36, 50, 52
- Variable dinámica 265
- Variables 10, 12, 13, 19, 20, 23, 30, 32, 42, 62, 63, 64, 65, 66, 67, 69, 70, 71, 74, 75, 90, 91
 estáticas 265, 267, 268
- Vectores 89, 90, 99, 100, 103, 136
- Verificación 7, 9, 24, 58
- void 6, 11, 17, 18, 20, 61
- volatile 6, 29
- while 15, 16, 17, 18, 19, 21, 22, 127, 131
- Wirth, Niklaus 6

