

```

4 #define SIZE2 300
5 #define DONE 1
6 #define MIN_GRADE 6
7
8 typedef struct e{
9     char title[SIZE1];
10    char requirements[SIZE2];
11    int number;
12    struct e *nextPtr;
13    struct e *previousPtr;
14 } EXERCISE_NODE;
15
16 typedef EXERCISE_NODE * EXERCISE_NODE_PTR;
17
18 typedef struct l{
19     int counter;
20     EXERCISE_NODE_PTR listPtr;
21 } LIST;
22
23 typedef int status;
24
25 status doExercise(LIST *);
26
27 int main(int argc, char *argv[]){
28     LIST exercises = {0, NULL};
29     int grade;
30     /*
31     Aquí se insertan nodos a la lista de tareas por realizar
32     por parte de Ricardo Ruiz Rodríguez
33     */
34     if(doExercise(&exercises) == DONE){
35         grade = 10;
36         printf("Well done!\n");
37     }else{
38         grade = rand() % MIN_GRADE;

```

Una Introducción a la Programación Estructurada en C

Ricardo Ruiz Rodríguez

Una Introducción a la Programación Estructurada en C

Ricardo Ruiz Rodríguez

Ruiz Rodríguez, Ricardo

Una Introducción a la Programación Estructurada en C -1º ed.- El Cid Editor, 2013.

pdf

ISBN digital – pdf 978-144-9291-33-4

Fecha de catalogación: 17/09/2013

© Ricardo Ruiz Rodríguez
© El Cid Editor

ISBN versión digital pdf: 978-144-9291-33-4

A mis hijos Ricardo y Bruno

Prólogo

Estimado lector:

En 1962 ya se conocían ampliamente los conceptos de análisis y diseño de algoritmos y programación de computadoras, tanto así que Donald E. Knuth comenzó su titánica labor de documentar, en varios volúmenes de texto conocidos en conjunto como *The Art of Computer Programming*, su experiencia como programador. Hoy, en pleno siglo XXI, los conceptos expuestos por Knuth siguen siendo válidos, aunque, definitivamente, la labor del programador de computadoras se ha vuelto demasiado demandante y compleja.

En nuestros tiempos, los profesionales del desarrollo de sistemas de software no sólo deben conocer un lenguaje de programación, sino que deben ser capaces de analizar requerimientos, diseñar la arquitectura de dicho sistema e invertir una gran cantidad de tiempo en validar el producto. Desafortunadamente, las tecnologías, las metodologías y los lenguajes de programación han evolucionado tan rápidamente que han dejado rezagados a los estudiantes de la disciplina del software. Muchos egresados de las universidades no tienen muy claro el reto al que se enfrentan ni tienen habilidades sólidas de análisis y uso adecuado de paradigmas de programación.

Actualmente existen muchos libros que tratan sobre cómo escribir programas utilizando el lenguaje C. También, existen varios libros dedicados a exponer la teoría de los paradigmas de programación (estructurada, funcional, orientada a objetos, etcétera). Además, existen algunos libros que enseñan al lector a utilizar adecuadamente alguno de estos paradigmas para diseñar algoritmos que se implementan utilizando un lenguaje específico. Sin embargo, existen pocos libros que enseñan un paradigma específico utilizando un lenguaje específico para estudiantes que hablan una lengua específica.

El libro que me honro en presentar al lector, escrito por el maestro Ricar-

do Ruiz Rodríguez, no sólo es producto de la experiencia docente del autor, que en sí misma es invaluable, sino que se deriva de un anhelo sincero de presentar el enfoque de la programación estructurada a los estudiantes en las universidades de México de una manera clara y didáctica. El libro ilustra conceptos de la programación estructurada como estructuras de control, descomposición funcional, recursividad y estructuras de datos básicas. Además, el presente texto detalla cómo utilizar los conceptos anteriores utilizando el lenguaje C.

Esperamos que el presente libro le sirva como guía en su interminable aprendizaje y que le sirva como apoyo en su ruta hacia la excelencia como ingeniero de software. En éste camino debe enfrentarse a innumerables retos y adquirir conocimientos cada vez más avanzados, y dichas tareas se hacen más sencillas cuando se dispone de fundamentos sólidos, los cuales deseamos que el libro ayude a construir. Sin más por el momento, le envío un cordial saludo.

Dr. Tomás Balderas Contreras
Software Engineer
Guadalajara Design Center, Intel.

Índice general

Prólogo	v
Índice de figuras	xiii
Índice de tablas	xvii
Índice de ejemplos	xix
Introducción	xxiii
1. El proceso de programación en C	1
1.1. Diseño del algoritmo	2
1.2. Pruebas de escritorio para el algoritmo	3
1.3. Edición del programa en código fuente	3
1.4. Compilación	4
1.4.1. Error fatal	5
1.4.2. Error preventivo	5
1.5. Vinculación	6
1.6. Carga y ejecución	6
1.7. Depuración	7
1.8. Estructura de un programa en C	7
1.9. Ejercicios	9
2. Bienvenido a C!	11
2.1. El primer programa en C	11
2.2. Lectura de datos	15
2.3. Consideraciones adicionales	21
2.4. Ejercicios	22

3. Estructuras de control	25
3.1. Estructuras de control para la selección	25
3.1.1. La estructura de control if	26
3.1.2. La estructura de control if-else	30
3.1.3. El operador ternario ? :	33
3.1.4. La estructura de control switch	35
3.2. Estructuras de control para la repetición	37
3.2.1. La estructura de control while	37
3.2.2. La estructura de control do-while	41
3.2.3. La estructura de control for	44
3.3. Estructuras de control combinadas	50
3.4. Tipos de repetición	52
3.4.1. Repetición controlada por contador	53
3.4.2. Repetición controlada por centinela	55
3.4.3. Repeticiones híbridas	57
3.5. Ejercicios	60
4. Módulos de programa: Funciones	67
4.1. Funciones de biblioteca	67
4.2. Conceptos y estructura	69
4.2.1. Estructura para las funciones en C	71
4.2.2. Ejemplos de definición de funciones	72
4.3. Ámbitos de variables	78
4.4. Bibliotecas personalizadas de funciones	81
4.5. Ejercicios	85
5. Recursividad	87
5.1. Definición y conceptos	88
5.1.1. Aspectos inherentes: <i>overhead</i>	89
5.2. Soluciones iterativas vs. recursivas	91
5.2.1. Factorial	91
5.2.2. Fibonacci	95
5.2.3. Experimento empírico	99
5.3. Torres de Hanoi	100
5.4. Consideraciones finales	103
5.5. Ejercicios	104

6. Arreglos	107
6.1. Arreglos de una dimensión (vectores)	108
6.1.1. Conceptos, representación y estructura	108
6.1.2. Declaración, inicialización y recorrido	109
6.1.3. Arreglos y funciones	112
6.1.4. Ordenamiento por burbuja	114
6.1.5. Búsqueda lineal y búsqueda binaria	118
6.1.6. Arreglos de caracteres (cadenas)	121
6.2. Arreglos de dos dimensiones (matrices)	124
6.2.1. Conceptos, representación y estructura	124
6.2.2. Declaración, inicialización, recorrido y uso con funciones	125
6.2.3. Suma de matrices	130
6.3. Arreglos de n dimensiones	131
6.4. Consideraciones finales con arreglos	134
6.5. Ejercicios	136
7. Apuntadores	143
7.1. Definición, estructura y representación	143
7.1.1. Piedra angular de la piedra angular	144
7.2. Parámetros por valor y por referencia	146
7.3. Aritmética de apuntadores	148
7.4. Apuntadores y arreglos	154
7.4.1. Asignación dinámica de memoria	156
7.4.2. Argumentos en la invocación de programas	163
7.5. Apuntadores a funciones	166
7.5.1. Arreglos de apuntadores a funciones	171
7.6. Ejercicios	175
8. Abstracción de datos	185
8.1. Conceptos, representación y estructura	186
8.2. Abstracción en acción	188
8.2.1. Números racionales	188
8.3. Estructuras compuestas	194
8.4. Arreglos de estructuras	198
8.5. Abstracción de matrices	199
8.6. Ejercicios	205

9. Archivos	213
9.1. Definición y conceptos	213
9.2. Archivos de texto	214
9.2.1. Archivos de acceso secuencial	214
9.2.2. Aplicaciones adicionales	220
9.2.3. Consideraciones adicionales	224
9.3. Archivos binarios	225
9.3.1. Archivos de acceso aleatorio	225
9.3.2. Archivos con contenido especial	232
9.4. Ejercicios	235
A. Fundamentos de programación	241
A.1. Panorama general	241
A.1.1. Teorema de Böhm y Jacopini	243
A.1.2. Estructuras de Control	243
A.2. Algoritmos	245
A.2.1. Definición y conceptos	245
A.2.2. Uso de la computadora en la resolución de problemas	246
A.2.3. Cinco importantes condiciones de un algoritmo	248
A.2.4. Estructura de un algoritmo	248
A.2.5. Pruebas de algoritmos	250
A.3. Diagramas de flujo	251
A.3.1. Estructuras de control	251
A.3.2. Diagrama de flujo del algoritmo de Euclides	253
A.4. Pseudocódigo	255
A.4.1. Estructuras de control	256
A.4.2. Pseudocódigo del algoritmo de Euclides	258
A.5. Pruebas de escritorio	258
A.6. Diseño básico de programas estructurados	260
A.6.1. Reglas para la formación de algoritmos estructurados	260
A.7. Consideraciones finales	262
B. Compilación de programas	265
B.1. El compilador GCC	265
B.2. Compilación de programas con GCC	266
B.2.1. Comprobando la instalación de GCC	266
B.2.2. Compilación básica	267
B.2.3. Ligado de bibliotecas	268

B.2.4. Generación de ejecutables	269
B.3. Redireccionamiento	269
B.3.1. Redireccionamiento de la entrada estándar	270
B.3.2. Redireccionamiento de la salida estándar	270
B.4. Consideraciones finales	271
Bibliografía	273
Índice Analítico	275
Agradecimientos	279
Acerca del Autor	281

Índice de figuras

1.1. Etapas del proceso de compilación de programas en C	2
2.1. Salida del Ejemplo 2.1	14
2.2. Salida del Ejemplo 2.3	19
2.3. Salida del Ejemplo 2.4	20
2.4. Salida del Ejemplo 2.5	24
3.1. Salida del Ejemplo 3.1	28
3.2. Salida del Ejemplo 3.2	30
3.3. Salida del Ejemplo 3.3	31
3.4. Salida del Ejemplo 3.6	34
3.5. Salida del Ejemplo 3.7	37
3.6. Salida de los ejemplos de ciclos while , do-while y for	39
3.7. Salida del Ejemplo 3.12	44
3.8. Salida del Ejemplo 3.15	47
3.9. Salida del Ejemplo 3.16	49
3.10. Salida del Ejemplo 3.17	51
3.11. Una posible salida del Ejemplo 3.18	54
3.12. Salida del Ejemplo 3.19	57
3.13. Salida del Ejemplo 3.20	58
3.14. Salida del Ejemplo 3.21	59
4.1. Salida del Ejemplo 4.1	69
4.2. Programación modular	70
4.3. Salida del Ejemplo 4.2	75
4.4. Salida del Ejemplo 4.3	77
4.5. Salida del Ejemplo 4.4	80
5.1. Recursividad infinita (aquí sólo hasta el nivel 3)	89

5.2.	Mecanismo de llamado y retorno de función	90
5.3.	Llamados y retornos recursivos para el factorial de cuatro	94
5.4.	Salida de los Ejemplos 5.1 y 5.2 para $n = 4$	95
5.5.	Árbol de recursividad para $n = 4$ de la serie de Fibonacci . . .	97
5.6.	Salida de los Ejemplos 5.3 y 5.4 para $n = 4$	98
5.7.	Estado inicial para cuatro discos en el problema de las torres de Hanoi	102
5.8.	Después de algunos movimientos, los $n - 1$ discos están en la posición auxiliar B	102
5.9.	Movimiento del disco de mayor diámetro a su posición final . .	102
5.10.	Situación final para el problema de las torres de Hanoi con cuatro discos	102
6.1.	Representación de un arreglo	108
6.2.	Salida del Ejemplo 6.1	111
6.3.	Salida del Ejemplo 6.2	112
6.4.	Salida del Ejemplo 6.2	114
6.5.	Salida del Ejemplo 6.6	124
6.6.	Representación de una matriz	125
6.7.	Inicialización de matrices (Ejemplo 6.7)	128
6.8.	Salida del Ejemplo 6.7	129
6.9.	Representación de arreglos de tres dimensiones	132
6.10.	Salida del Ejemplo 6.9	134
7.1.	Representación de un apuntador	144
7.2.	Salida del Ejemplo 7.1	146
7.3.	Representación de parámetros por valor y por referencia . . .	148
7.4.	Salida del Ejemplo 7.2	149
7.5.	Salida del Ejemplo 7.3	152
7.6.	Salida del Ejemplo 7.4	153
7.7.	Representación del arreglo de apuntadores <i>codigoMorse</i> del Ejemplo 7.5	154
7.8.	Salida del Ejemplo 7.5	156
7.9.	Representación del arreglo generado en el Ejemplo 7.6	158
7.10.	Salida del Ejemplo 7.6	159
7.11.	Representación del arreglo generado en el Ejemplo 7.7	161
7.12.	Salida del Ejemplo 7.8	164
7.13.	Salida del Ejemplo 7.9	165

7.14. Representación de <i>argv</i> para la ejecución mostrada en 7.13 . . .	166
7.15. Salida del Ejemplo 7.12	170
7.16. Salida del Ejemplo 7.13	172
8.1. Representación de la estructura de las variables de tipo <i>struct</i> <i>r</i> o <i>RACIONAL</i> del Ejemplo 8.1	190
8.2. Salida del Ejemplo 8.1	191
8.3. Salida del Ejemplo 8.3	198
8.4. Salida del Ejemplo 8.4	200
8.5. Salida del Ejemplo 8.6	204
9.1. Salida del Ejemplo 9.1	217
9.2. Salida del Ejemplo 9.2	220
A.1. Símbolos y su significado en los diagramas de flujo	252
A.2. Estructura secuencial en diagrama de flujo	253
A.3. Estructuras de selección en diagrama de flujo	254
A.4. Estructuras de repetición en diagrama de flujo	254
A.5. Diagrama de flujo para el algoritmo de Euclides	255
A.6. Diagrama de flujo no estructurado	261
A.7. Aplicación de las reglas 1 y 2	262
A.8. Aplicación de la regla 3	263

Índice de tablas

2.1. Operadores aritméticos en C	18
3.1. Operadores relacionales en C	26
3.2. Operadores lógicos en C	52
5.1. Primeros 12 términos de la serie de Fibonacci	96
5.2. Algoritmo de las Torres de Hanoi	103
7.1. Laberinto de 5×5	181
9.1. Modos básicos de apertura de archivos para la función fopen .	216
A.1. Proceso general de resolución de problemas	247
A.2. Cinco condiciones que debe cumplir un algoritmo	249
A.3. Estructura general de un algoritmo	250
A.4. Estructura secuencial en pseudocódigo	256
A.5. Estructura de selección simple en pseudocódigo	257
A.6. Estructura de selección doble en pseudocódigo	257
A.7. Estructura de selección múltiple en pseudocódigo	257
A.8. Estructura de repetición hacer mientras (while)	258
A.9. Estructura de repetición repetir mientras (do-while)	258
A.10. Algoritmo de Euclides en pseudocódigo	259
A.11. Reglas para la formación de algoritmos estructurados	261

Índice de ejemplos

1.1. Estructura general de un programa en C	8
2.1. Primer programa en C	11
2.2. Versión masoquista del primer programa en C	14
2.3. Programa para sumar dos números enteros	16
2.4. Programa para sumar dos números enteros (segunda versión) .	19
2.5. Uso del operador unario sizeof	23
3.1. Uso de la estructura de selección if y los operadores relacionales	27
3.2. División de dos números enteros (primera versión)	29
3.3. División de dos números enteros (segunda versión)	31
3.4. División de dos números enteros (tercera versión)	32
3.5. Ejemplo 3.1 reescrito con estructuras if-else anidadas	33
3.6. Uso del operador ternario ?:	34
3.7. Uso de la estructura de selección switch	36
3.8. Uso de la estructura de control while	38
3.9. Uso de la estructura de control while y la directiva define . .	40
3.10. Programa para mostrar la diferencia entre el pre incremento y el pos incremento	41
3.11. Uso de la estructura de control do-while	42
3.12. Uso de la estructura de control do-while para la validación de datos	43
3.13. Uso de la estructura de repetición for	45
3.14. Sumatoria de n números enteros	46
3.15. Estructura de repetición for con dos variables de control . . .	47
3.16. Uso de la funciones srand y rand para generar números alea- torios	48
3.17. Programa para mostrar la combinación y anidamiento de es- tructuras de control	50

3.18. Promedio de calificaciones utilizando repetición controlada por contador	53
3.19. Promedio de calificaciones utilizando repetición controlada por centinela	56
3.20. Uso de la sentencia continue	57
3.21. Uso de la sentencia break	59
3.22. Misterioso 1	61
3.23. Misterioso 2	62
3.24. Misterioso 3	62
4.1. Cálculo de raíces para una ecuación de segundo grado utilizando la fórmula general	68
4.2. Uso de funciones para determinar el máximo y el mínimo de tres números enteros distintos	73
4.3. Implementación de una calculadora básica utilizando funciones	75
4.4. Ámbitos de variables, variables automáticas y estáticas	78
4.5. Funciones contenidas en el archivo “miBiblioteca.h”	82
4.6. Inclusión y uso de la biblioteca personalizada de funciones	83
5.1. Función para el factorial de un número (versión iterativa)	92
5.2. Función para el factorial de un número (versión recursiva)	93
5.3. Función de Fibonacci (versión iterativa)	96
5.4. Función de Fibonacci (versión recursiva)	98
5.5. Esqueleto para medir el tiempo de ejecución en segundos de un proceso largo	99
6.1. Uso de un arreglo de enteros	110
6.2. Uso de un arreglo de tipo float	111
6.3. Arreglos y funciones	113
6.4. Funciones para el ordenamiento por burbuja y burbuja mejorado	116
6.5. Funciones para la búsqueda lineal y la búsqueda binaria	119
6.6. Uso de cadenas	122
6.7. Uso de arreglos de dos dimensiones (matrices)	126
6.8. Biblioteca de funciones para operaciones con matrices	130
6.9. Uso de arreglos de tres dimensiones (cubos)	133
6.10. Matriz como valor de retorno	135
7.1. Uso de apuntadores	145
7.2. Parámetros por valor y por referencia	147
7.3. Recorrido de cadenas con apuntadores	151
7.4. Conversión de una cadena a mayúsculas	153
7.5. Arreglo de apuntadores a char	155

7.6. Asignación dinámica de memoria	157
7.7. Biblioteca de funciones para crear matrices con asignación dinámica de memoria	159
7.8. Asignación dinámica de memoria para matrices	162
7.9. Argumentos de la función main	165
7.10. Ordenamiento ascendente y descendente por burbuja	167
7.11. Burbuja con apuntador a funciones	168
7.12. Uso del ordenamiento por burbuja con apuntador a funciones .	169
7.13. Arreglo de apuntadores a funciones	173
8.1. Abstracción de números racionales (primera version)	189
8.2. Abstracción de números racionales (segunda versión)	192
8.3. Estructuras compuestas	196
8.4. Arreglo de estructuras	199
8.5. Biblioteca de funciones para matrices de números racionales .	200
8.6. Uso de matrices de números racionales	202
9.1. Uso de un archivo secuencial para el almacenamiento de datos .	214
9.2. Uso de un archivo secuencial para la lectura de datos	218
9.3. Implementación básica del comando <i>cat</i>	221
9.4. Cambia un carácter por otro dentro de un archivo	222
9.5. Creación de la estructura de un archivo binario de acceso alea- torio	227
9.6. Acceso para la modificación de datos de un archivo binario de acceso aleatorio	228
9.7. Lectura de un archivo binario	233

Introducción

La práctica de la programación es una de las disciplinas más interesantes, intelectuales y retadoras que se conozcan, ya que no requiere únicamente del conocimiento de la sintaxis del lenguaje seleccionado para escribir programas, sino de razonamiento, lógica, y capacidad para *especificar* de manera detallada, un conjunto de instrucciones que den solución a un problema determinado.

Tanto el paradigma de programación estructurada, como el lenguaje de programación C son relativamente antiguos, su creación pertenece al siglo pasado, pero esto no quiere decir que están en desuso u obsoletos, tanto el paradigma como el lenguaje siguen vigentes.

Este libro es para aprender a programar en el paradigma de la programación estructurada utilizando al lenguaje de programación C. Es resultado de un proceso evolutivo de diversos ajustes realizados al método de enseñanza utilizado, los cuales son producto de mi experiencia académica en la impartición tanto del paradigma, como del lenguaje de programación C, que dicho sea de paso, es uno de mis favoritos.

El método propuesto pretende guiar al estudiante paso a paso en la creación de programas, así como en la comprensión de la estructura de un programa estructurado escrito en el lenguaje C, lo cual se realiza a través de ejemplos que evolucionan progresivamente tanto en complejidad lógica, como en elementos estructurales.

Los ejercicios que acompañan a cada capítulo, tienen en su mayoría, la finalidad de reforzar los conceptos descritos en el texto a través de experimentos, que van desde la modificación más simple de los programas de ejemplo, hasta cambios o modificaciones más elaboradas, basadas también en su mayoría en los programas de ejemplo; de ahí que la recomendación desde ahora sea hacia la elaboración de los ejercicios, ya que están pensados como complemento al material descrito en el texto.

Libros del lenguaje de programación C hay tantos como bellezas naturales y arquitectónicas existen en México. Estos libros, en adición con la Internet, son material de consulta indispensable para complementar los conceptos aquí descritos. Con todo, las ventajas que tiene este libro en comparación con algunos de los existentes, es que no es una traducción, que está escrito en el castellano que se habla en México, y que ha pretendido concretizar los conceptos del paradigma estructurado y del lenguaje C en un texto de mediana extensión.

Este libro, es importante aclararlo, no es un manual de referencia, ni la guía última de la programación en C. Tampoco es un libro que enseñe técnicas avanzadas de programación, cualquiera que éstas sean y suponiendo que existan. Es poco probable que después de la lectura de este libro le den el premio Turing, que gane un concurso de programación, o que apruebe una certificación del lenguaje C. Sin embargo, si sigue el método propuesto, tendrá una muy buena base de los conocimientos esenciales del paradigma de la programación estructurada, y del lenguaje de programación C.

Es mi más sincero deseo que la lectura del libro resulte de interés, utilidad y agrado para el lector, ya que esos fueron los principales objetivos. Espero haberlos alcanzado.

La estructura del libro es la siguiente:

- **Capítulo 1:** presenta un panorama general de las etapas involucradas en el proceso de programación, así como su relación con el lenguaje de programación C. Así mismo, se muestra la estructura general que todo programa escrito en C debe tener.
- **Capítulo 2:** muestra la estructura de un programa en C e introduce al lenguaje a través de ejemplos sencillos. Adicionalmente, el capítulo familiariza al lector con algunas funciones de biblioteca.
- **Capítulo 3:** presenta y describe las estructuras de control que existen en C, así como su funcionamiento y uso a través de ejemplos que van progresivamente evolucionando en complejidad.
- **Capítulo 4:** describe brevemente el concepto de programación modular, su relación con las bibliotecas de funciones, y la creación de bibliotecas de funciones definidas por el programador.
- **Capítulo 5:** presenta una breve introducción al estigmatizado tema de la recursividad. Se abordan brevemente los conceptos y aspectos

inherentes a la recursividad como lo son el *overhead* y las relaciones de recurrencia.

- **Capítulo 6:** este capítulo presenta los pormenores fundamentales de la estructura de datos denominada arreglo. Así mismo, extiende la representación hacia arreglos de más de una dimensión.
- **Capítulo 7:** se desarrollan los fundamentos y la esencia de los apuntadores. Se describen y presentan ejemplos tanto de variables de tipo apuntador, como de apuntadores a funciones.
- **Capítulo 8:** este capítulo introduce al concepto de abstracción para la representación de tipos de datos definidos por el programador. Se presentan y discuten algunos de los pormenores de los **struct** de C, enfatizando particularmente la abstracción de datos para mejorar la gestión de información.
- **Capítulo 9:** introduce al lector a la gestión de archivos de texto y binarios en C. Se construyen aplicaciones sencillas pero útiles, con la finalidad de familiarizar progresivamente al lector con los llamados de función para el manejo de archivos.
- **Apéndice A:** presenta una breve introducción a los fundamentos de la programación: algoritmos, pseudocódigos y diagramas de flujo.
- **Apéndice B:** describe brevemente los tipos de compiladores de C, en particular el compilador de GNU GCC. También se describen los elementos básicos del uso del compilador GCC, para compilar programas en la línea de comandos.

En resumen, es un libro que siempre quise escribir y que, sin falsas modestias, me hubiera gustado tener en mi época de estudiante. Espero haber podido reflejar las expectativas que tuve como estudiante, y como académico, la retroalimentación que he tenido con mis estudiantes, los cuales han contribuido de manera significativa a la elaboración del método descrito.

My basic idea is that programming is the most powerful medium of developing the sophisticated and rigorous thinking needed for mathematics, for grammar, for physics, for statistics, and all the “hard” subjects. In short, I believe more than ever that programming should be a key part of the intellectual development of people growing up.

Seymour Papert

Capítulo 1

El proceso de programación en C

El proceso de diseño y creación de programas utilizando al lenguaje de programación C, incluye al menos las siguientes etapas:

1. Diseño del algoritmo.
2. Pruebas de escritorio para el algoritmo.
3. Edición del programa en código fuente.
4. Compilación.
5. Vinculación.
6. Carga y ejecución.
7. Depuración.

Las etapas 3 – 7 del proceso anterior, se gestionan de manera sumamente conveniente dentro de un **IDE** (*Integrated Development Environment*) y, si bien éste no es indispensable para el desarrollo de programas en C, resultan de mucha utilidad. La Figura 1.1 muestra la interacción con el disco y la memoria principal de cada una de las siete etapas, mismas que se describirán en secciones posteriores.

Un IDE es básicamente una herramienta que nos permite y facilita la gestión de las diferentes etapas del proceso de programación, mediante la

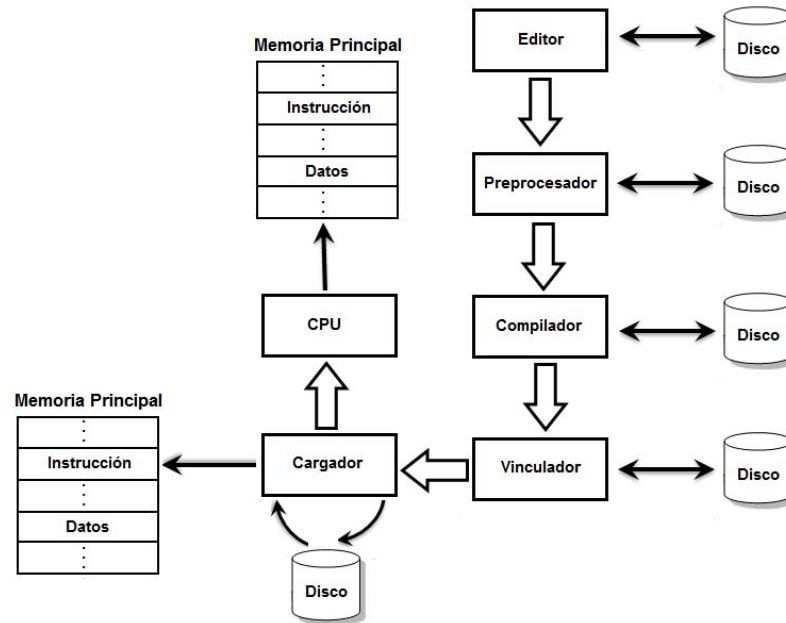


Figura 1.1: Etapas del proceso de compilación de programas en C

combinación de un conjunto de teclas, o de la selección de algunas opciones de menús por ejemplo, además de proporcionar un entorno de programación accesible y práctico, facilitando con ello las tareas de edición, compilación, vinculación de módulos, ejecución y depuración entre otras.

En las siguientes secciones se describe brevemente cada una de las etapas del proceso de creación de programas.

1.1. Diseño del algoritmo

Esta etapa es, por mucho, la más importante del proceso de programación, por lo que no es casualidad que sea la primera, ya que a partir de la descripción de un problema al cual se le quiera dar solución a través de un programa estructurado escrito en C, se realiza un análisis inicial para identificar:

- Los datos de entrada.
- Los datos de salida.

Ruiz Rodríguez, Ricardo

Una Introducción a la Programación Estructurada en C -1º ed.- El Cid Editor, 2013.

pdf

ISBN digital – pdf 978-144-9291-33-4

Fecha de catalogación: 17/09/2013

© Ricardo Ruiz Rodríguez

© El Cid Editor

ISBN versión digital pdf: 978-144-9291-33-4


El Cid Editor

de características extendidas del lenguaje C (C++), o hacen que el compilador interprete algunas sentencias del archivo como propias del paradigma de programación orientado a objetos por ejemplo, o que el compilador analice el código en base a otra gramática distinta a la de C, generando así mensajes que pudieran confundir al programador.

También es posible editar archivos que agrupen un conjunto de funciones que parezcan archivos de biblioteca personalizados. Técnicamente no lo son, pero es un estándar de *facto* el denominarlos de esta manera. Más adelante, en el Capítulo 4, se profundizará un poco más al respecto.

1.4. Compilación

La etapa de compilación puede resumirse como el **proceso de traducción** de un programa escrito en un lenguaje artificial¹ de alto nivel (código fuente), a un conjunto de instrucciones ejecutables por una computadora (programa en código objeto o binario).

El proceso de compilación se escucha simple, pero en realidad no lo es. De hecho, el compilador de C realiza al menos cuatro etapas:

1. **Pre procesamiento:** antes de cualquier cosa, el pre procesador de C ejecuta las directivas de inclusión de archivos (**#include**), y de definición de tipos (**#define**), que son las más comunes pero no las únicas. La etapa de pre procesamiento se realiza con la finalidad de definir y establecer las dependencias necesarias indicadas por éste tipo de directivas.
2. **Análisis léxico:** es la primera fase del proceso de traducción, y su función es la de descomponer el programa fuente en elementos léxicos o símbolos denominados *tokens*, los cuales serán utilizados en una etapa posterior durante el proceso de traducción. La salida del analizador léxico es la entrada para el analizador sintáctico.
3. **Análisis sintáctico:** este tipo de análisis se encarga, *grosso modo*, de verificar que las sentencias contenidas en el archivo fuente correspondan

¹Los lenguajes naturales se distinguen de los artificiales básicamente en el siguiente sentido: los lenguajes naturales son los que hablan las diferentes razas y culturas del mundo junto con su gramática, mientras que los lenguajes artificiales son aquellos que ha inventado el ser humano para algún propósito específico.

a la gramática del lenguaje C, es decir, que se cumplan con las reglas de estructura, escritura, y orden del lenguaje.

4. **Análisis semántico:** el analizador semántico analiza el significado, sentido o interpretación, así como la intención de los símbolos, elementos léxicos, sentencias y expresiones de la gramática del lenguaje C.

El resultado de estas cuatro etapas es, en el mejor de los casos, la traducción a código objeto del código fuente.

Por otro lado, existe el escenario de la detección de algún error identificado en alguna de las etapas anteriormente mencionadas del proceso de compilación. En éste sentido, pueden identificarse dos tipos de errores:

1. Error fatal (*Fatal error*).
2. Error preventivo (*Warning*).

1.4.1. Error fatal

Este tipo de errores no generan el código objeto derivado del proceso de traducción, debido a la detección de un problema grave durante el análisis del código fuente.

Los errores fatales pueden ser por ejemplo: la omisión de un punto y coma, la falta o inconsistencia de paréntesis o delimitadores de bloque, elementos (*tokens*) no definidos, etc.

En el *argot* computacional, es común denominar a este tipo de errores como: **errores en tiempo de compilación**.

1.4.2. Error preventivo

Los errores preventivos (*warnings*) sí producen código objeto, y su carácter es más bien informativo, respecto a posibles problemas detectados por el compilador durante el proceso de traducción.

La mayoría de este tipo de errores son del tipo semántico, por lo que en éste sentido, es responsabilidad del compilador indicar los elementos que le parecieron sospechosos; sin embargo, es responsabilidad del programador validar o no estas advertencias.

Algunos de estos errores, si no se corrigen, derivan en problemas durante la ejecución del programa, aunque no necesariamente, y se les denomina: **errores en tiempo de ejecución**.

Por otro lado, debe tenerse presente que no todos los errores en tiempo de ejecución son derivados de errores preventivos, la mayoría de los errores en tiempo de ejecución provienen de una lógica incorrecta.

1.5. Vinculación

El vinculador (*linker*) es el responsable de unir las distintas partes que conforman un programa en C.

Como parte del proceso de compilación de la etapa anterior, y asumiendo que el código fuente fue traducido con éxito en código objeto, se tienen, hasta ese momento, módulos relacionados pero independientes entre sí.

Incluso para el programa más simple (véase el capítulo 2), es común hacer uso de funciones, constantes simbólicas, o tipos de datos definidos en alguna de las bibliotecas de funciones, por lo que el vinculador es **responsable** de verificar la correspondencia y existencia de los módulos a los que se hace referencia en el programa, así como de establecer los vínculos o relaciones correspondientes con dichos módulos, para constituir y generar un módulo único y funcional, mismo que conformará el conjunto de instrucciones ejecutables o programa ejecutable.

En resumen, la función del vinculador es la de unir o vincular el código producido por el programador, con el código al que se hace referencia en las bibliotecas de funciones, con la finalidad de generar un archivo ejecutable: el programa ejecutable.

1.6. Carga y ejecución

Cada programa ejecutable derivado de un programa en C, cuando se ejecuta o procesa se denomina formalmente **proceso**. Un proceso es una instancia en ejecución de un programa.

Puede decirse en general que un programa y un proceso son conceptos intercambiables indistintamente, pero formalmente obedecen a aspectos diferentes.

- Un **programa** es un conjunto de instrucciones almacenadas en disco, listas para ser potencialmente ejecutadas como una tarea o aplicación por parte del sistema operativo.

- Un **proceso** es ese mismo conjunto de instrucciones cargadas ya en la memoria principal de la computadora, mismas que están en posibilidad real de ser procesadas o ejecutadas por el microprocesador. Todas las aplicaciones, tareas o procesos que se están ejecutando en una computadora, deben estar en memoria total o parcialmente.

El programa responsable de trasladar (cargar) las instrucciones de un programa a la memoria, para que sea susceptible de ser ejecutado como proceso se llama: **cargador** (*loader*).

1.7. Depuración

La depuración es propia del proceso de diseño de un programa, y es de hecho, el complemento de la etapa de pruebas de escritorio, ya que en esta fase, se prueban el conjunto de instrucciones escritas en lenguaje de alto nivel, con la finalidad de verificar y de validar que se cumplen los criterios y objetivos para los cuales fue diseñado el programa.

Es común asociar esta etapa a la corrección de errores o problemas (*bugs*) detectados durante la ejecución del programa y, aunque si bien es cierto que en esta etapa es en donde se tratan de identificar las sentencias que originan algún tipo de problema, no debe perder de vista que las pruebas constituyen una parte fundamental del proceso de diseño y programación, de tal forma que constituyen una parte fundamental de la verificación y validación del programa desarrollado.

Esta etapa incluye técnicas básicas de depuración como: la inclusión de puntos de ruptura (*break points*), ejecución paso a paso (*trace*), inspección de variables y expresiones (*watch*), ejecución de módulos completos (*step over*), y ejecución de módulos paso a paso (*step into*), entre otros.

1.8. Estructura de un programa en C

Un programa estructurado es lo opuesto a un programa desordenado.

Un **programa estructurado** es un programa con una distribución específica y un orden específico de las partes que lo componen, en donde dichas partes constituyen un conjunto de elementos relacionados pero independientes entre sí.

Con lo anterior en mente, un programa estructurado escrito en el lenguaje de programación C, tiene la estructura general descrita en el Ejemplo 1.1.

```
1 #include <.h> /* Inclusion de bibliotecas de funcion */
2
3 #define          /* Definicion de constantes simbolicas y macros */
4
5 union { };          /* Definicion de uniones */
6 struct { };         /* Definicion de estructuras */
7 typedef          /* Definicion de tipos */
8
9 tipo_de_dato funcion(lista); /* Prototipos de funcion */
10
11 int main(lista){ }; /* Funcion principal */
12
13 tipo_de_dato funcion(lista){ /* Definicion de funciones */
14 }
```

Ejemplo 1.1: Estructura general de un programa en C

Las líneas 5, 6 y 7 no siempre ni necesariamente van en ese orden, ya que depende de las características específicas del programa que se esté realizando. La regla de oro que sigue el lenguaje C en éste sentido es la siguiente: antes de usar *algo*, ese *algo* debe estar previamente definido, por lo que si una estructura (**struct**) requiere de una unión (**union**) por ejemplo, dicha unión deberá estar definida antes que la estructura.

Por ahora, todas o algunas de las partes mostradas en el Ejemplo 1.1 pueden parecer extrañas, pero se irán especificando y detallando a lo largo del libro.

En los capítulos siguientes se inicia con la especificación y detalle de cada una de las partes de la estructura mostrada en el Ejemplo 1.1. Por ahora basta con que no olvide su composición, y recurra a ella cada vez que se introduzca un nuevo elemento, con la finalidad de ir imprimiendo en su mente dicha estructura, por medio de un ejercicio repetido de identificación y relación.

1.9. Ejercicios

1. Dedique tiempo para investigar y conocer algunos de los IDEs más usuales para el desarrollo de programas en C. Tome en cuenta que los IDEs dependen del sistema operativo que se utilice.
2. Busque y lea la historia tan interesante que existe atrás del lenguaje de programación C, y conteste al menos las siguientes preguntas:
 - ¿Quién o quiénes fueron sus creadores?
 - ¿Cómo fue su evolución?
 - ¿En qué año surge?
 - ¿Cuántas versiones ha habido?
 - ¿A qué se refiere el ANSI C?

Este ejercicio, además de gratificador, constituirá una parte importante de su cultura general, y una parte fundamental de su cultura computacional.

3. Investigue el concepto de código espagueti, para reforzar el concepto de programación estructurada. En base a lo anterior, ¿cuáles considera que serían las ventajas de tener una estructura dentro de un código fuente?
4. Investigue qué lenguajes de programación además de C, soportan el paradigma estructurado.
5. Investigue el concepto de **Ingeniería de Software**.
6. Investigue el concepto de **Proceso de Desarrollo de Software**.
7. Revise el apéndice A para refrescar algunos de los conceptos ahí expuestos, antes de avanzar al siguiente capítulo.

Capítulo 2

Bienvenido a C!

Este capítulo muestra la estructura de un programa escrito en el lenguaje C. Se describen línea a línea ejemplos muy sencillos, con la finalidad de establecer la correspondencia entre la estructura de un programa en C (Ejemplo 1.1), y los programas presentados. Adicionalmente el capítulo va familiarizando al lector con algunas funciones de biblioteca, sentencias, y estructuras de control del lenguaje de programación C.

2.1. El primer programa en C

En base a la estructura general de un programa en el lenguaje C discutida en el Capítulo 1 (Ejemplo 1.1), el Ejemplo 2.1 muestra algunos de esos elementos.

```
1  /* El primer programa en C.
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  int main(){
7     printf("Bienvenido a C!\n");
8
9     return 0; /* Indica que el programa termino exitosamente */
10 }
```

Ejemplo 2.1: Primer programa en C

La estructura de control principal que gobierna el paradigma de la programación estructurada y al lenguaje de programación C, es la estructura de control secuencial.

La **estructura de control secuencial**, ejecuta o procesa las sentencias, instrucciones, expresiones, funciones, etc., en orden consecutivo, esto es, en la misma forma en que la mayoría de los habitantes de América Latina leemos: de izquierda a derecha y de arriba hacia abajo, por lo que el procesamiento de las líneas de los códigos de ejemplo seguirá dicha regla.

Las líneas 1-3 del Ejemplo 2.1, muestran el uso de comentarios. Los comentarios en C se inician con una diagonal seguida inmediatamente de un símbolo de asterisco (“/*”), y terminan con los mismos símbolos pero colocados en orden inverso, es decir: “*/”.

El uso de comentarios dentro de un programa, permite esclarecer algunos de los aspectos poco claros, o no tan evidentes. Los comentarios ayudan también a especificar cierta fórmula, una determinada expresión, o algún aspecto considerado durante la elaboración del algoritmo.

Los comentarios deberían ayudar a hacer más claro y entendible un programa. Un código bien escrito debe ser auto documentado, esto es, que el uso apropiado de identificadores, la claridad de expresiones, y la indentación deberían ser suficientes para hacer clara y comprensible la intención del código, pero cuando valga la pena hacer alguna aclaración por las razones anteriormente expuestas o alguna otra, los comentarios son una herramienta muy útil.

Ahora bien, aunque no es una regla del lenguaje, sí constituye una buena práctica de programación el iniciar los programas con un comentario que describa la funcionalidad del programa, así como algunas otras características que se consideren relevantes para su comprensión. En el Ejemplo 2.1 se muestra, además de lo anterior, el nombre del autor del código (línea 2). Se recomienda ampliamente seguir ésta práctica.

La línea 4 es una directiva de inclusión. Le indica al compilador que incluya la biblioteca de entrada y salida estándar (**stdio.h**), misma que contiene, entre otras cosas, a la función **printf** que se utilizará en la línea 7. La biblioteca de entrada y salida estándar se incluye en la mayoría de los programas en C.

La línea 6 define la función principal **main()**. El punto de entrada de todo programa en C es **main**, si esta función no existe, no es posible generar un programa ejecutable. La línea 6 indica también que **main** regresará un valor entero, mismo que está estrechamente relacionado con la línea 9. Todas las funciones en C deben especificar el tipo de dato de retorno; el tipo de dato por omisión es **int** (entero).

Por otro lado, la línea 7 muestra el uso más simple quizá de la función **printf**. La función **printf**, toma el argumento entre comillas (“ ”), y lo envía a la salida estándar, la cual está normalmente asociada con la pantalla. Observe el símbolo “\n”, a éste tipo de símbolos se les denomina **secuencias de escape**, y aunque es un símbolo compuesto por dos caracteres: “\” y “n”, se hace referencia a él como uno solo.

Técnicamente hablando, el símbolo “\n” representa un avance de línea y un retorno de carro, en analogía al mecanismo que utilizaban las máquinas de escribir mecánicas. El avance de línea y retorno de carro (\n) le indican al cursor que avance al siguiente renglón y se ponga al principio del mismo después de haber escrito: “Bienvenido a C!”. Las sentencias en C terminan con “;”.

La última línea del Ejemplo 2.1 (línea 9), regresa el valor cero (0) al invocador. En éste sentido, vale la pena hacer una mención comúnmente pasada por alto: como lo ilustra el proceso de diseño de programas descritos en el Capítulo 1, es común utilizar un IDE para la elaboración de programas en C, lo cual no debe llevar al lector a pensar que éste sea el proceso final, sino saber que es sólo una fase de diseño. Una vez que un programa ha sido probado, depurado, corregido y mejorado, se espera que trabaje como una aplicación más, independiente del entorno en el que se desarrolló. Ahora bien, no debe perderse de vista que las aplicaciones son ejecutadas por el Sistema Operativo (SO), cualquiera que éste sea, por lo que el valor de retorno se regresa al invocador de la función **main**, es decir, al SO y de ahí su importancia.

Si el SO recibe como valor de retorno de la aplicación el valor cero, se indica que la aplicación terminó normalmente y sin problemas; en otro caso, podría haber ocurrido algún tipo de error o situación anormal durante la ejecución de la aplicación. Supongamos que durante la ejecución de un programa, éste solicita memoria al SO y por alguna razón no se le concede; en éste caso, dicho programa no podría continuar su ejecución y tendría que terminar de manera anormal, regresando un valor distinto de cero, pues bien, éste valor precisamente es el que le serviría al SO para saber que hubo un problema, identificar de qué tipo (en base al valor recibido), e iniciar un mecanismo de recolección de basura o de compactación de memoria por ejemplo, entre otras muchas posibilidades, en función del valor de retorno procesado.

Finalmente, si se observan las líneas 6 y 10, dichas líneas contienen, respectivamente, los símbolos “{” y “}”, los cuales se denominan **delimitadores de bloque**, y su función es la de agrupar declaraciones y sentencias dentro

Bienvenido a C!

Figura 2.1: Salida del Ejemplo 2.1

de una **sentencia compuesta**, o **bloque** (cuerpo) de la función¹.

Los delimitadores de bloque pueden ser visualizados como el inicio (*begin*) y el fin (*end*) utilizado en la definición de algoritmos², pero en realidad son más que eso, ya que definen un **bloque de construcción** y en su momento se hará hincapié en ello, por ahora, resultará de utilidad que se visualicen de esta manera.

Una vez que se ha comprendido el código del Ejemplo 2.1, el paso siguiente sería compilarlo. El proceso de compilación y ejecución es muy dependiente de la forma de trabajo: utilizando un IDE o una compilación en línea. Consulte el apéndice B del libro para obtener más información en este sentido.

La salida del Ejemplo 2.1 muestra en la pantalla el mensaje: Bienvenido a C!, tal como se presenta en la Figura 2.1.

Considere ahora el Ejemplo 2.2.

```

1  /* El primer programa en C en su
2     version masoquista.
3     @autor Ricardo Ruiz Rodriguez
4  */
5
6  #include <stdio.h>
7
8  int main() {
9      printf("B"); printf("i"); printf("e");
10     printf("n"); printf("v"); printf("e");
11     printf("n"); printf("i"); printf("d");
12     printf("o"); printf(" "); printf("a");
13     printf(" "); printf("C!");
14     printf("\n");
15
16     return 0;
17 }
```

Ejemplo 2.2: Versión masoquista del primer programa en C

El Ejemplo 2.2 muestra los siguientes aspectos respecto del Ejemplo 2.1:

1. Pueden utilizarse varias funciones **printf** para imprimir un mismo mensaje en la pantalla, si ésto tiene o no sentido, es otra cosa.

¹También agrupan declaraciones y sentencias de estructuras de control, como se verá más adelante.

²Para profundizar un poco más al respecto, consulte el apéndice A.

2. Pueden ir más de una sentencia por renglón: en el Ejemplo 2.2 hay varios **printf** por línea, y el “;” separa y delimita las sentencias. Ésto obedece a estilo de programación y conveniencia, lo que no debe olvidar es que, aunque para la computadora sería lo mismo procesar todo el programa en una sola línea, para los humanos no lo es, ya que repercute en la legibilidad, y por consiguiente, en la comprensión del programa. No olvide que se requieren programas que sean fáciles de entender no sólo para quien lo escribió, sino para cualquiera que conozca la sintaxis del lenguaje.
3. El número de sentencias o funciones **printf** por línea, es independiente de la lógica del programa.
4. Si no se le indica explícitamente a la función **printf** que avance de línea y regrese el cursor al inicio, el cursor se queda en la posición siguiente al último símbolo puesto en pantalla.

Analice el programa del Ejemplo 2.2 y compárelo con el del Ejemplo 2.1. Estos programas son lógicamente equivalentes pero diferentes, ya que su código es distinto.

Antes de seguir, asegúrese de que comprende lo que sucede, y compruebe que tanto el Ejemplo 2.1 como el Ejemplo 2.2 producen la misma salida (Figura 2.1).

2.2. Lectura de datos

Es momento de pasar a hacer algo un poco más interesante que imprimir mensajes en pantalla y jugar con la función **printf**.

El Ejemplo 2.3 muestra un sencillo programa, que lee desde la entrada estándar, dos números enteros decimales (los operandos), y los suma. Como antes, se describirá línea a línea.

Las líneas 1-7 deberían resultar ahora familiares, en base a la explicación del Ejemplo 2.1.

Por otro lado, la línea 8 presenta un nuevo elemento: la **declaración de variables**.

La declaración de variables en C puede hacerse al iniciar un bloque ({ ... }), y tiene la siguiente estructura general:

```

{
    tipo_de_dato lista de variables;
    .
    .
    .
}

```

donde *tipo_de_dato* es alguno de los tipos de datos de C, y la *lista de variables* es una lista de identificadores separada por comas. En base a lo anterior, la línea 8 está declarando tres variables de tipo entero a través de la palabra reservada **int**.

```

1  /* Programa para sumar dos enteros.
2   @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main(){
8      int operando1, operando2, suma;    /* declaracion de variables */
9
10     printf("Operando1?: ");            /* Solicitud (prompt) */
11     scanf("%d", &operando1);           /* lee el primer operando */
12     printf("Operando2?: ");            /* Solicitud (prompt) */
13     scanf("%d", &operando2);           /* lee el segundo operando */
14     suma = operando1 + operando2;      /* Operacion de suma y asignacion */
15     printf("La suma es %d\n", suma);   /* Imprime suma */
16
17     return 0;
18 }

```

Ejemplo 2.3: Programa para sumar dos números enteros

Una **palabra reservada** es una palabra (*token*) que no puede ser utilizada como identificador, y que es propia del lenguaje, lo cual quiere decir que ninguna variable o función (ningún identificador) por ejemplo, puede llamarse “**int**”.

Por otro lado, un **identificador** es una secuencia de caracteres sin espacios que inicia con una letra. Los identificadores sirven para dar nombre a las variables que se declaran y que serán utilizadas dentro de un programa.

En C existen diferentes tipos de datos: **int**, **float** y **char** son algunos de ellos. La idea de tipo de dato se refiere a una categoría de datos pertenecientes a un dominio pero, desde el punto de vista del lenguaje de programación C ¿qué es un tipo de dato?

Lo que una computadora almacena en su memoria principal no es otra cosa que una secuencia de bits; pues bien, esa misma secuencia de bits tiene

diferentes interpretaciones, y la interpretación específica está en función del tipo de dato.

Considere la siguiente secuencia de bits: **1 1 0 1 0 1 0 1**. Suponga que éste *byte* representa, como número, al entero decimal **213**. Si esta misma secuencia de bits, se interpreta ahora como carácter (**char**) por ejemplo, podría representar a la letra '**R**' (no es que así sea, es sólo una suposición), mientras que interpretada como un número con punto decimal (**float**) podría representar al **69.5**.

En resumen, un **tipo de dato** es la manera en que una computadora *interpreta* un patrón de bits.

Continuando con el Ejemplo 2.3, la línea 10 debería resultar familiar. A este tipo de sentencias se les conoce en el *argot* computacional como *prompt*, es decir, un mensaje de solicitud de datos.

La línea 11 es nueva y muestra uno de los usos de la función **scanf**. La función **scanf** se compone básicamente de dos partes:

1. El (los) especificador(es) de formato indicados entre comillas.
2. Una lista de variables separadas por comas, en dónde se almacenarán los valores procesados (leídos) de la entrada estándar.

El especificador de formato “**%d**”, le indica a la función **scanf** que leerá un entero decimal, mientras que la expresión “**&operando1**”, especifica la variable en la que se almacenará el valor leído de la entrada estándar: *operando1*.

Debe observarse que el nombre de la variable ha sido antecedido por el símbolo “**&**”, la justificación de ello se comprenderá mejor cuando se revise el Capítulo 7, por ahora, no debe olvidar anteceder con dicho símbolo, el nombre de la variable en la que se desee almacenar el valor procesado, ya que su omisión no es detectada por el compilador como un error, debido a que no es una violación a la gramática del lenguaje C.

Las líneas 12 y 13 son análogas a las líneas 10 y 11 respectivamente. Asegúrese de comprender la analogía.

Por otro lado, la línea 14 presenta la expresión:

$$suma = operando1 + operando2;$$

Dicha expresión le indica a C que realice la operación aritmética suma, sobre los valores almacenados en *operando1* y *operando2*, y que el resultado se **asigne** (almacene) en la variable *suma*.

Operador	Descripción
/	División (cociente)
%	Módulo (residuo)
*	Multipliación
+	Adición
-	Sustracción

Tabla 2.1: Operadores aritméticos en C

La Tabla 2.1 muestra los operadores aritméticos en C.

Los operadores se evalúan siguiendo una precedencia. La **precedencia** de operadores le indica al compilador cuáles de los operadores en una expresión, deben ser evaluados primero.

Considere la siguiente expresión:

$$3 + 5 * 4$$

La expresión anterior es evaluada como **23**, y no como **32**, debido precisamente a que la multiplicación tiene una precedencia mayor sobre la adición, sin importar que la adición sea la operación que aparece primero.

Si lo que se desea es primero realizar la suma y luego el producto, la expresión debe escribirse como:

$$(3 + 5) * 4$$

Los paréntesis modifican la forma en que son evaluadas las expresiones, es decir, *modifican la precedencia* de cualquier operador y éste es su uso más común, aunque en muchas ocasiones, se utilizan sólo por claridad en la forma de representar una expresión, sin que modifiquen la precedencia de los operadores involucrados.

Los operadores aritméticos “/”, “%” y “*” tienen la misma precedencia, y es más alta que la de los operadores “+” y “-”. Si en una expresión existen operadores con la misma precedencia, la expresión es evaluada de izquierda a derecha.

Finalmente, la línea 15³ introduce una novedad respecto al Ejemplo 2.1, ya que muestra el uso de especificadores de formato dentro de la función **printf**.

Al igual que en la función **scanf**, el especificador de formato “%d” le indica a la función **printf** que en su lugar va a imprimir, en la salida estándar,

³La línea 17 se explicó a detalle en el Ejemplo 2.1.

```
Operando1?: -45
Operando2?: 32
La suma es -13
```

Figura 2.2: Salida del Ejemplo 2.3

un entero decimal cuyo valor está almacenado en la variable *suma*, la cual se encuentra después de la coma. En este sentido, por cada especificador de formato debe existir su correspondiente variable asociada, de tal forma que tanto el número de especificadores de formato, como el de variables, debe corresponder.

Una posible salida del Ejemplo 2.3 se muestra en la Figura 2.2. Asegúrese de comprender la descripción realizada hasta el momento, así como de entender lo que sucede con la ejecución y la salida correspondiente antes de continuar.

Considere ahora el Ejemplo 2.4 y compárelo con el Ejemplo 2.3. Note que todas las líneas son iguales, excepto por la línea 15.

```
1  /* Programa para sumar dos enteros (segunda version).
2     @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main(){
8      int operando1, operando2, suma;    /* declaracion de variables */
9
10     printf("Operando1?: ");            /* Solicitud (prompt) */
11     scanf("%d", &operando1);           /* lee el primer operando */
12     printf("Operando2?: ");            /* Solicitud (prompt) */
13     scanf("%d", &operando2);           /* lee el segundo operando */
14     suma = operando1 + operando2;      /* Operacion de suma y asignacion */
15     printf("%d + %d = %d\n", operando1, operando2, suma);
16
17     return 0;
18 }
```

Ejemplo 2.4: Programa para sumar dos números enteros (segunda versión)

La línea 15 del Ejemplo 2.4, contiene tres especificadores de formato “%d”. Cada especificador de formato, le indica a la función **printf** que imprima en orden, el entero decimal correspondiente almacenado en cada una de las variables correspondientes: *operando1*, *operando2* y *suma*.

La lógica del programa del Ejemplo 2.4 no cambia en nada con respecto de la lógica del Ejemplo 2.3, sólo se ha cambiado el estilo en la forma de

```
Operando1?: -45  
Operando2?: 32  
-45 + 32 = -13
```

Figura 2.3: Salida del Ejemplo 2.4

presentar los resultados, la cual es más *ad hoc* con la intención del programa.

El Ejemplo 2.4 muestra la forma de incluir más de un especificador de formato en la función **printf**, de tal forma que puede observarse que por cada especificador de formato, existe su correspondiente variable asociada. Es importante mencionar también que es responsabilidad del programador el asegurar que exista una *correspondencia* entre el especificador de formato y el tipo de dato de la variable, C no realiza esta verificación.

Los especificadores de formato para **printf** tienen la siguiente forma general:

$$\%[-]m.nx$$

donde % delimita el inicio del especificador de formato y x representa el especificador de formato a utilizar. El guión o signo de menos, justifica el campo a la izquierda, si se omite, el campo se justifica (alineá) a la derecha. Ahora bien, dependiendo del valor de x , los número enteros representados por m y n se interpretan de manera diferente:

- Usualmente, m es la longitud mínima y n es la longitud máxima del campo (ancho de campo) que se utilizará para imprimir x .
- Si x representa el especificador de formato para un número con punto decimal, n es interpretado como la precisión que deberá ser utilizada (número de decimales después del punto).

Una posible salida para el Ejemplo 2.4 se muestra en la Figura 2.3. Pruebe los Ejemplos 2.3 y 2.4 con distintos valores y observe sus resultados; repita el procedimiento hasta que se sienta cómodo y entienda por completo el mecanismo de funcionamiento de los programas. Revise la sección de ejercicios para realizar modificaciones derivadas de los ejemplos.

2.3. Consideraciones adicionales

Cuando un proceso se ejecuta, tiene asociados tres flujos (*streams*) de manera automática, sin que se haga nada especial para su creación:

1. **Entrada estándar** (*stdin*): asociada al teclado, pero puede ser redireccionado.
2. **Salida estándar** (*stdout*): asociada a la pantalla, pero puede ser redireccionado.
3. **Error estándar** (*stderr*): asociada a la pantalla y no puede ser redireccionado.

El redireccionamiento se refiere a la capacidad de un proceso de tomar o enviar datos de, o hacia un archivo. Ésto quiere decir, que un programa no siempre lee sus datos desde el teclado, o que no siempre envía sus datos a la pantalla, sino que los datos pueden ser leídos de, o enviados a archivos, sin incluir explícitamente sentencias o funciones para la gestión de dichos archivos.

Usando redireccionamiento, el proceso no se “enteraría” de que los datos que lee o escribe, no están siendo obtenidos desde el teclado o presentados en la pantalla respectivamente, ya que el mecanismo de redireccionamiento es transparente para los procesos, favoreciendo así su versatilidad.

Por último, es importante que sepa desde ahora, que en general los flujos están estrechamente relacionados con los archivos, por lo que en el Capítulo 9 se retomará este aspecto; por ahora, basta con lo descrito hasta aquí⁴.

⁴En el apéndice B se describe un poco más el mecanismo de redireccionamiento para los programas desde la línea de comandos.

2.4. Ejercicios

1. Compare y relacione la estructura del Ejemplo 2.1 con la estructura general de un programa en C del Ejemplo 1.1.
2. En base al Ejemplo 2.1 ¿Qué sucede si elimina la secuencia de escape “\n” del final de “Bienvenido a C!\n”? No olvide compilar después de su modificación.
3. En base al Ejemplo 2.1 pruebe colocando en diferentes partes del mensaje “Bienvenido a C!” la secuencia de escape “\n” y analice lo que sucede. No olvide compilar después de cada modificación.
4. Investigue qué otras secuencias de escape existen y para qué sirven. Escriba un programa que las incluya, y pruebe cada una de ellas.
5. Modifique el Ejemplo 2.1 y pruebe lo que sucede si se introducen otras secuencias de escape. No olvide compilar después de cada modificación.
6. Escriba un programa en C que imprima su nombre completo en la pantalla en una sola línea.
7. Escriba un programa en C que imprima su nombre completo en la pantalla pero dividido en tres líneas:
 - a) En la primera línea su(s) nombre(s).
 - b) En la segunda línea su primer apellido.
 - c) En la tercera línea su segundo apellido.
8. Diferentes culturas y pueblos tienen distintos tipos de leyendas, y la disciplina de la computación no escapa a ellas. Se dice que todo aquel que se inicie en las maravillosas artes de la programación estructurada usando al lenguaje de programación C debe, si no quiere recibir la maldición de ser un mal programador toda su vida, escribir un programa que imprima, en la salida estándar el mensaje:

Hola Mundo!

Independientemente de sus creencias, evite el riesgo de la maldición y colabore con la perpetuidad de esta tradicional leyenda realizando este simple pero bello ejercicio, de tal manera que, si el augurio se cumple, se dé por descartada, al menos, la maldición por la omisión del ejercicio.

9. Investigue qué otros tipos de datos existen en C, qué representación tienen, cuál es el rango de números que pueden almacenar, y cuál es el especificador de formato que utilizan para las funciones **printf** y **scanf** respectivamente. **Nota:** no siempre es el mismo.
10. Extienda el ejemplo que se muestra a continuación (Ejemplo 2.5) para que considere todos los tipos de datos en C que investigó en el ejercicio anterior. Note que el Ejemplo 2.5 muestra el uso del operador unario⁵ **sizeof** para determinar el tamaño en *bytes* que ocupa el operando asociado.

```

1  /* Ejemplo de uso del operador sizeof para determinar el numero de
2     bytes requeridos para representar al operando asociado.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int main(){
8      int i;
9      double d;
10
11     printf("sizeof(i) = %ld\n", sizeof(i));
12     printf("sizeof(int) = %ld\n", sizeof(int));
13     printf("\nsizeof(d) = %ld\n", sizeof(d));
14     printf("sizeof(double) = %ld\n", sizeof(double));
15
16     return 0;
17 }

```

Ejemplo 2.5: Uso del operador unario **sizeof**

Observe también que el operador **sizeof** trabaja, no sólo sobre tipos de datos (líneas 12 y 14), sino también sobre variables (líneas 11 y 13).

Este ejercicio y el anterior, debería llevar al lector a deducir, que el **rango** de los números representados por un tipo de dato depende del tamaño en *bits*⁶ del tipo de dato.

⁵Un operador unario se refiere a que el operador sólo necesita un operando para realizar su función; el operador **+** por ejemplo, es un operador binario porque necesita de dos operandos para realizar su función.

⁶Para determinar el tamaño en *bits*, multiplique el tamaño en *bytes* por ocho, que es el número de *bits* que tiene cada *byte*.

```
sizeof(i) = 4  
sizeof(int) = 4  
  
sizeof(d) = 8  
sizeof(double) = 8
```

Figura 2.4: Salida del Ejemplo 2.5

La salida del Ejemplo 2.5 se muestra en la Figura 2.4.

11. Experimente omitiendo intencionalmente el uso del operador “&” en la lectura de una variable como la que se hace en el Ejemplo 2.3. ¿Qué sucede cuando compila? ¿qué pasa cuando se ejecuta?, si el programa es ejecutado ¿qué valor se guarda en la variable?
12. Escriba una programa que, basándose en el Ejemplo 2.3, realice la resta de dos números enteros decimales.
13. Escriba una programa que, basándose en el Ejemplo 2.3, realice la multiplicación de dos números enteros decimales.
14. Escriba una programa que, basándose en el Ejemplo 2.3, realice el módulo de dos números enteros decimales. ¿Qué podría pasar en este caso?
15. Escriba una programa que, basándose en el Ejemplo 2.3, realice la división de dos números enteros decimales ¿Qué podría pasar en este caso?
16. Repita los ejercicios 12-15 con las consideraciones descritas en la sección 2.2, e implementadas en el Ejemplo 2.4.
17. Dependiendo del IDE que esté utilizando, investigue y documéntese acerca de la depuración de programas. Si está compilando en línea, busque un tutorial del programa **gdb**; en cualquier caso, debe saber que la depuración, no sólo le será de suma utilidad, sino que es una tarea fundamental de la programación.

Capítulo 3

Estructuras de control

El lenguaje de programación C incorpora tres estructuras de control:

1. Estructura de control secuencial.
2. Estructuras de control para la selección de sentencias.
3. Estructuras de control para la repetición de sentencias.

La **estructura de control secuencial** se encarga de que se ejecuten en orden y en secuencia (de ahí su nombre), las sentencias, operaciones y expresiones escritas en la gramática del lenguaje C, esto es, de izquierda a derecha y de arriba hacia abajo.

Por otro lado, las estructuras de selección y de repetición de sentencias se describen a lo largo del presente capítulo, en el cual también se muestra el funcionamiento y uso de dichas estructuras de control, en el contexto del lenguaje de programación C.

3.1. Estructuras de control para la selección

Si realizó los ejercicios propuestos en el Capítulo 2, habrá notado que para el caso del programa de la división y el módulo puede presentarse un problema, debido a la indeterminación latente en la división.

El resultado de la división se indetermina si el denominador es cero, ¿qué hacer en estos casos?. Para prevenir este tipo de situaciones, se necesita de una estructura de control que permita procesar o no un grupo de

Operador	Descripción
==	Igual que
!=	Distinto de
<	Menor estricto que
>	Mayor estricto que
<=	Menor igual que
>=	Mayor igual que

Tabla 3.1: Operadores relacionales en C

sentencias, en función de alguna condición. A éste tipo de estructuras de control se les conoce como **estructuras de control para la selección de sentencias**.

3.1.1. La estructura de control **if**

La estructura de control **if** tiene la siguiente estructura general:

```
if (expresión) {
    sentencia(s);
}
```

La estructura de selección **if** procesa o ejecuta el grupo de *sentencia(s)*¹ delimitadas por su bloque, **si** al ser evaluada la *expresión* ésta es distinta de cero, es decir: *expresión* $\neq 0$.

La llave izquierda “{” indica el inicio del bloque correspondiente a la estructura, mientras que la llave derecha “}” delimita su final.

Si al ser evaluada la *expresión* ésta es igual a cero, se ignora el grupo de *sentencia(s)* delimitadas por el bloque de la estructura, es decir, no se procesan (no son ejecutadas), y la ejecución continúa con la primera sentencia que se encuentre después del delimitador de fin de bloque del **if**.

El Ejemplo 3.1, además de mostrar el funcionamiento de la estructura de selección **if**, muestra el uso de los operadores relacionales presentados en la Tabla 3.1.

¹*sentencia(s)* puede ser cualquier sentencia(s) válida(s) en el lenguaje de programación C, como por ejemplo, otras estructuras de control (de selección y de repetición), incluida ella misma, a lo cual se le denomina: **estructuras de control anidadas**.

A partir del Ejemplo 3.1, sólo se hará mención explícita de las líneas de código que introduzcan algún elemento nuevo o diferente respecto de lo hasta ahora comentado en los ejemplos anteriores, con la finalidad de hacer más ágil la explicación.

```
1  /* Programa que ilustra el uso de operadores relacionales, de
2     comparacion y la estructura de control de seleccion if.
3     @autor Ricardo Ruiz Rodriguez
4  */
5
6  #include <stdio.h>
7
8  int main() {
9      int num1, num2;
10
11     printf("Proporcione dos numeros enteros: ");
12     scanf("%d%d", &num1, &num2);
13
14     if(num1 == num2)
15         printf("Los numeros son iguales %d == %d\n", num1, num2);
16
17     if(num1 != num2)
18         printf("Los numeros son distintos %d != %d\n", num1, num2);
19
20     if(num1 < num2)
21         printf("%d es menor estricto que %d\n", num1, num2);
22
23     if(num1 > num2)
24         printf("%d es mayor estricto que %d\n", num1, num2);
25
26     if(num1 <= num2)
27         printf("%d es menor o igual que %d\n", num1, num2);
28
29     if(num1 >= num2)
30         printf("%d es mayor o igual que %d\n", num1, num2);
31
32     return 0;
33 }
```

Ejemplo 3.1: Uso de la estructura de selección **if** y los operadores relacionales

Lo primero a comentar aparece en la línea 12, la cual introduce una nueva modalidad en la lectura de datos a través de la función **scanf**: es posible leer más de un dato en un solo llamado a la función. Los dos especificadores de formato “**%d**”, le indican a la función **scanf** que lea dos enteros decimales de la entrada estándar.

Observe que al igual que con la función **printf**, por cada especificador de formato debe existir su correspondiente variable asociada para su almacenamiento, y es responsabilidad del programador el que también exista una correspondencia entre el especificador de formato, y el tipo de dato de la variable.

```
Proporcione dos numeros enteros: 56 65
Los numeros son distintos 56 != 65
56 es menor estricto que 65
56 es menor o igual que 65
```

Figura 3.1: Salida del Ejemplo 3.1

Por otro lado, la línea 14 muestra el uso de la estructura de selección **if** y el operador de igualdad “==”. La forma de interpretar en nuestro lenguaje dicha sentencia es: “**Si** *num1* **es igual que** *num2*, ejecuta la sentencia de la línea 15”.

Note que intencionalmente se ha omitido el uso de los delimitadores de bloque para las estructuras de selección **if**, lo cual es una práctica muy común en C y se recomienda usarse con precaución, ya que si se omiten las llaves, únicamente se considera la sentencia inmediata posterior a la estructura, sin importar que la indentación de las demás sentencias sugieran otra cosa.

Continuando con el ejemplo, la línea 17 se puede interpretar como: “**Si** *num1* **es distinto de** *num2*, ejecuta la sentencia de la línea 18”; mientras que la línea 20: “**Si** *num1* **es menor estricto que** *num2*, ejecuta la sentencia de la línea 21”; las líneas 23, 26 y 29 tienen una interpretación análoga.

Las líneas 15, 18, 21, 24, 27 y 30 deberían ser ya totalmente comprendidas. Una posible salida del Ejemplo 3.1 se muestra en la Figura 3.1.

Es importante que pruebe y experimente con otros datos, y que se asegure de cubrir **todos** los casos considerados en las expresiones condicionales de las estructuras de selección **if**. Esta actividad es conocida como **pruebas de caja blanca o transparente**, y la validación de todas las sentencias dentro del código de un programa, es una labor relacionada con el concepto de **complejidad ciclomática**, y aunque la explicación de estos conceptos está fuera de los alcances de este libro, su mención y presentación no.

Ahora bien, en base al funcionamiento de la estructura de selección **if** y la explicación realizada, y considerando el caso del intento de división por cero al que se enfrentó en los ejercicios de la sección 2.4, ¿se le ocurre algo para prevenir dicha indeterminación?

Si no ha realizado dichos ejercicios, este es un buen momento para hacerlos, recuerde que el método del libro requiere que se trabajen los ejercicios de cada capítulo como parte del aprendizaje y del reforzamiento, tanto de los conceptos como de los ejemplos presentados.

En este sentido, el Ejemplo 3.2 es una primera propuesta de solución a la indeterminación matemática, y está dada esencialmente por la línea 15, ya que en ella se realiza la verificación del denominador utilizando la estructura de selección **if**, y el operador de comparación “!=”.

```
1  /* Programa para dividir dos enteros.
2     @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main() {
8      int num, den, div;
9
10     printf("Numerador?: ");
11     scanf("%d", &num);
12     printf("Denominador (!=0)?: ");
13     scanf("%d", &den);
14
15     if(den != 0) {
16         div = num / den;
17         printf("%d / %d = %d\n", num, den, div);
18     }
19
20     return 0;
21 }
```

Ejemplo 3.2: División de dos números enteros (primera versión)

Si el denominador (*den*) es **distinto de cero** (línea 15), entonces es posible realizar la división (línea 16), y presentar el resultado (línea 17).

Observe que si el denominador es cero, simplemente no se realiza la división y no se presenta ningún tipo de información en la salida estándar, lo cual, además de parecer poco cordial e informativo es incorrecto, no desde la perspectiva de la lógica de funcionamiento, sino desde la perspectiva del usuario del programa, ya que no basta con escribir programas funcionales, sino también útiles.

Por lo anterior, sería bueno proporcionar al usuario algún tipo de notificación acerca de lo que ha ocurrido, para ello, se requiere de *algo* que permita al programador seleccionar entre un grupo de sentencias u otro en función de una determinada situación, y ese *algo* es la estructura de control para la selección de sentencias **if-else**, misma que se describirá en la siguiente sección. Por ahora, el Ejemplo 3.2 valida y previene una operación de indeterminación matemática, haciéndolo lógicamente funcional.

Una posible salida para el programa del Ejemplo 3.2 se muestra en la Figura 3.2. Al igual que antes, pruebe y experimente con otros datos, genere

```
Numerador?: 10
Denominador (!=0)?: 3
10 / 3 = 3
```

Figura 3.2: Salida del Ejemplo 3.2

otras salidas, y verifique qué hace el programa cuando el denominador es cero.

Por último, es importante mencionar que en el lenguaje de programación C la división de enteros es entera, sin importar que la división no sea exacta, esto quiere decir, que en la división de enteros la parte fraccionaria es descartada, y sólo se conserva la parte entera. Más adelante se introducirán nuevos tipos de datos (**float** y **double**), los cuales proporcionarán un mejor manejo de números fraccionarios.

3.1.2. La estructura de control if-else

La estructura de control para la selección if-else tiene la siguiente estructura general:

```
if (expresión) {
    sentencia(s)1;
} else {
    sentencia(s)2;
}
```

La estructura de selección **if-else** procesa o ejecuta el grupo de *sentencia(s)1*² delimitadas por su bloque, **si** al ser evaluada la *expresión* ésta es distinta de cero, es decir: $expresión \neq 0$. Pero si al ser evaluada la *expresión* ésta es igual a cero, se procesa o ejecuta el grupo de *sentencia(s)2* delimitadas por su respectivo bloque. Las llaves izquierdas “{” delimitan el inicio del bloque, mientras que las llaves derechas “}” delimitan su fin.

Antes de continuar, compare los Ejemplos 3.2 y 3.3, y compruebe que son esencialmente iguales. Note que la diferencia se presenta en la línea 18, ya que el Ejemplo 3.3 muestra el uso de la estructura de selección **if-else**.

²*sentencia(s)1* o *sentencia(s)2* puede ser cualquier sentencia(s) válida(s) en el lenguaje de programación C, como por ejemplo, otras estructuras de control (de selección y de repetición) incluida ella misma, a lo cual se le denomina: **estructuras de control anidadas**.

```
Numerador?: 45
Denominador (!=0)? : 0
Intento de division por 0
```

Figura 3.3: Salida del Ejemplo 3.3

La estructura de selección **if-else** para este ejemplo, podría interpretarse en nuestro lenguaje de la siguiente forma: “**Si** *den* es distinto de cero, procesa la división (línea 16) y presenta su resultado (línea 17); **en caso contrario**, escribe un mensaje que notifique el intento de división por cero”.

```
1  /* Programa para dividir dos enteros (segunda version).
2   @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main(){
8      int num, den, div;
9
10     printf("Numerador?: ");
11     scanf("%d", &num);
12     printf("Denominador (!=0)? : ");
13     scanf("%d", &den);
14
15     if(den != 0){
16         div = num / den;
17         printf("%d / %d = %d\n", num, den, div);
18     }else{
19         printf("Intento de division por 0\n");
20     }
21
22     return 0;
23 }
```

Ejemplo 3.3: División de dos números enteros (segunda versión)

La Figura 3.3 muestra una posible entrada de datos para generar la salida de indeterminación. Al igual que antes, pruebe con otros datos, y asegúrese de entender el funcionamiento de la estructura de selección **if-else** antes de continuar.

El Ejemplo 3.3 pudo haber sido escrito al menos de otra forma. Considere el Ejemplo 3.4, y compárelo con el primero.

Probablemente se esté preguntando, ¿cuál de estas dos formas es mejor?, y la respuesta es ambas, ya que las dos son igual de eficientes y previenen la indeterminación.

Tome en consideración que tanto el Ejemplo 3.3 como el Ejemplo 3.4 son lógicamente equivalentes, que ambos producirán la misma salida (como la de la Figura 3.3), y que los dos son un reflejo de la forma en que los seres humanos analizamos, pensamos y resolvemos las cosas: desde distintas perspectivas.

```
1  /* Programa para dividir dos enteros (tercera version).
2     @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main(){
8      int num, den, div;
9
10     printf("Numerador?: ");
11     scanf("%d", &num);
12     printf("Denominador (!=0)?: ");
13     scanf("%d", &den);
14
15     if(den == 0){
16         printf("Intento de division por 0\n");
17     }else{
18         div = num / den;
19         printf("%d / %d = %d\n", num, den, div);
20     }
21
22     return 0;
23 }
```

Ejemplo 3.4: División de dos números enteros (tercera versión)

Ahora bien, una vez que haya comprendido el funcionamiento de la estructura de selección **if-else**, responda lo siguiente:

1. ¿Qué sucede si el Ejemplo 3.1 se reescribe utilizando una estructura de selección **if-else** anidada?.
2. En base a lo anterior ¿se generaría la misma salida mostrada en la Figura 3.1?.
3. Si genera la misma salida ¿por qué?, y si no genera la misma salida ¿por qué?.

Lea, comprenda, y analice el Ejemplo 3.5 para determinar sus respuestas, y asegúrese de entender lo que sucede. Trate de determinar sus respuestas antes de ejecutar el programa.

```

1  /* Programa que ilustra el uso de operadores relacionales, de comparacion y
2     estructuras de control de seleccion if-else anidadas.
3     @autor Ricardo Ruiz Rodriguez
4  */
5
6  #include <stdio.h>
7
8  main() {
9      int num1, num2;
10
11     printf("Proporcione dos numeros enteros: ");
12     scanf("%d%d", &num1, &num2);
13
14     if(num1 == num2)
15         printf("Los numeros son iguales %d == %d\n", num1, num2);
16     else if(num1 != num2)
17         printf("Los numeros son distintos %d != %d\n", num1, num2);
18     else if(num1 < num2)
19         printf("%d es menor estricto que %d\n", num1, num2);
20     else if(num1 > num2)
21         printf("%d es mayor estricto que %d\n", num1, num2);
22     else if(num1 <= num2)
23         printf("%d es menor o igual que %d\n", num1, num2);
24     else if(num1 >= num2)
25         printf("%d es mayor o igual que %d\n", num1, num2);
26
27     return 0;
28 }

```

Ejemplo 3.5: Ejemplo 3.1 reescrito con estructuras **if-else** anidadas

3.1.3. El operador ternario ? :

El lenguaje de programación C incorpora un operador muy versátil, el cual es sumamente útil para escribir expresiones condicionales compactas: el operador ternario (? :).

El operador ternario tiene la siguiente estructura general:

$$\text{valor} = \text{expresión1} ? \text{expresión2} : \text{expresión3}$$

donde la *expresión1* es evaluada primero, y si es distinta de cero (*expresión1* $\neq 0$), entonces se evalúa la *expresión2* y su resultado es asignado a *valor*; en caso contrario, se evalúa la *expresión3* y su resultado es asignado a *valor*.

Note que el operador ternario es una estructura de selección **if-else** compacta. En este sentido, es importante señalar que el operador ternario tiene su correspondiente equivalencia representada en dicha estructura de selección. Asegúrese de comprender la equivalencia:

```
Introduzca dos numeros distintos: 19 74
El mayor es: 74
```

Figura 3.4: Salida del Ejemplo 3.6

```
if (expresión1)
    valor = expresión2;
else
    valor = expresión3;
```

La principal ventaja radica en lo compacto y la expresividad del operador ternario, pero en general es posible representar en una estructura de selección **if-else**, cualquier expresión que haya sido escrita utilizando dicho operador.

Considere el Ejemplo 3.6, el cual determina el mayor de dos números enteros.

```
1  /* Programa mostrar el uso del operador ternario "? :".
2   @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  int main() {
7      int num1, num2, mayor;
8
9      printf("Introduzca dos numeros distintos: ");
10     scanf("%d %d", &num1, &num2);
11
12     mayor = num1 > num2 ? num1 : num2;
13     printf("El mayor es: %d\n", mayor);
14
15     return 0;
16 }
```

Ejemplo 3.6: Uso del operador ternario ?:

La línea 12 y 13 del Ejemplo 3.6 pudieron haber sido escritas en una sola, es decir, el valor retornado por el operador ternario “?:” pudo haber sido utilizado directamente como el valor a imprimir por el especificador de formato “%d” de la función **printf** de la línea 13.

Revise la sección de ejercicios al final de este capítulo, ahí se propone realizar dicha modificación y comprobar su equivalencia.

Una posible salida para el Ejemplo 3.6 se muestra en la Figura 3.4.

3.1.4. La estructura de control **switch**

La estructura de control para la selección **switch** tiene la siguiente estructura general:

```
switch (expresión) {
    case expresión_constante1:
        sentencia(s)1;
        break;
    case expresión_constante2:
        sentencia(s)2;
        break;
    .
    .
    .
    case expresión_constanteN:
        sentencia(s)N;
        break;
    default:
        sentencia(s)N + 1;
}
```

La estructura de control **switch** evalúa la *expresión* y compara el resultado con el valor de la *expresión_constante1* después de la palabra reservada **case**, **si** coinciden, entonces se procesan o ejecutan el grupo de *sentencia(s)1*³ **hasta** encontrar la palabra reservada **break**. En caso de no coincidir, la estructura **switch** compara el resultado de la *expresión* con el de *expresión_constante2*, si coincide, se procesan o ejecutan el grupo de *sentencia(s)2* **hasta** encontrar la palabra reservada **break**. El proceso descrito continúa de manera análoga para todos los valores de *expresión_constante* que contenga la estructura **switch**.

Por otro lado, el grupo de *sentencia(s)N + 1* se procesan o ejecutan, sólo si el resultado de la *expresión* no coincidiera con ninguna de las **N** *expresiones_constantes*, lo cual representa el caso por omisión (**default**).

³Una vez más, *sentencia(s)1*, *sentencia(s)2*, *sentencia(s)N* o *sentencia(s)N + 1* puede ser cualquier *sentencia(s)* válida(s) en el lenguaje de programación C, como por ejemplo, otras estructuras de control (de selección y de repetición) incluida ella misma, a lo cual se le denomina: **estructuras de control anidadas**.

Debe resaltarse la importancia de la palabra reservada **break**, ya que si se omite, el compilador no genera ningún tipo de error, pero una posible consecuencia sería por ejemplo, algunos minutos de entretenidas sesiones de depuración.

```
1  /* Programa para traducir numeros a palabras.
2  @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  int main(){
7      int num;
8
9      printf("Numero(1-3)? ");
10     scanf("%d", &num);
11
12     switch(num){
13         case 1:
14             printf("El numero es UNO\n");
15             break;
16         case 2:
17             printf("El numero es DOS\n");
18             break;
19         case 3:
20             printf("El numero es TRES\n");
21             break;
22         default:
23             printf("Indescifrable...\n");
24     }
25
26     return 0;
27 }
```

Ejemplo 3.7: Uso de la estructura de selección **switch**

Al igual que antes, observe que la llave izquierda “{” delimita el inicio del bloque de la estructura **switch**, mientras que la llave derecha “}” delimita su final.

El Ejemplo 3.7 muestra el uso de la estructura **switch**. El programa lee un número entero, y a través de ésta estructura de control, “convierte” el número en su correspondiente palabra.

La estructura de selección **switch** trabaja de la siguiente manera: la línea 12 evalúa el valor de la variable *num* y lo compara con el valor que se encuentra después de la primera palabra reservada **case** (línea 13), si coincide, entonces se procesa la sentencia de la línea 14 y a continuación la de la línea 15, la cual le indica a la estructura de selección **switch** que termine⁴, y pro-

⁴Procesar el **break** sería equivalente a procesar la llave de fin de bloque de la estructura **switch**.


```
Numero(1-3)? 2  
El numero es DOS
```

Figura 3.5: Salida del Ejemplo 3.7

cese la primera sentencia que se encuentre después del final de su bloque. En caso de no coincidir, el proceso de comparación se repetiría con los valores de las líneas 16 y 19, para finalmente procesar la sentencia **default**, en caso de no existir coincidencia.

Experimente eliminando del Ejemplo 3.7 una, dos, o las tres sentencias **break** en diferentes combinaciones, no olvide compilar de nuevo el programa en cada modificación que realice. La intención de esta prueba controlada, es la de proporcionarle valiosa experiencia en la omisión intencional de la sentencia **break**.

Una posible salida para el Ejemplo 3.7 se muestra en la Figura 3.5.

3.2. Estructuras de control para la repetición

En lenguaje de programación C contempla, como parte de su gramática, tres estructuras de control para la repetición de sentencias, comúnmente denominadas **ciclos**:

1. La estructura de control **while**.
2. La estructura de control **do-while**.
3. La estructura de control **for**.

Las secciones siguientes describen cada una de ellas.

3.2.1. La estructura de control **while**

La estructura de control o ciclo **while**, tiene la siguiente estructura general:

```
while (expresión) {  
    sentencia(s);  
}
```

El ciclo **while** procesa o ejecuta el grupo de *sentencia(s)* delimitadas por su bloque de manera repetida, **mientras** la *expresión*, al ser evaluada, sea distinta de cero, es decir: *expresión* $\neq 0$.

La llave izquierda “{” delimita el inicio de su bloque, mientras que la llave derecha “}” delimita su final.

Cuando el conjunto de *sentencia(s)* delimitadas por el bloque del ciclo **while** se procesan, y la secuencia de ejecución alcanza el delimitador de fin del bloque, la estructura de repetición **while** modifica el flujo secuencial de ejecución, de tal forma que el flujo de ejecución *brinca* hacia *expresión* para que ésta vuelva a ser evaluada, y nuevamente, si *expresión* $\neq 0$, se **repite** el procesamiento de la(s) *sentencia(s)* delimitadas por su bloque. En caso contrario, se procesa la siguiente sentencia que se encuentre después de su delimitador de fin de bloque.

```
1  /* Ejemplo de la estructura de control de repeticion while.
2   @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  int main(){
7      int i;          /* variable de control */
8
9      i = 1;          /* inicializacion de la variable de control */
10     while(i <= 10){ /* expresion condicional */
11         printf("i = %d\n", i);
12         i = i + 1; /* modificacion de la variable de control */
13     }
14     return 0;
15 }
```

Ejemplo 3.8: Uso de la estructura de control **while**

El Ejemplo 3.8 imprime en la salida estándar los números del uno al diez, uno por renglón. La salida se muestra en la Figura 3.6.

La expresión condicional de la línea 10 está compuesta por una constante (10), y aunque ayuda a entender mejor la intención del ejemplo, en la práctica de la programación a éste tipo de constantes se les conoce como “**números mágicos**”, y en general no se consideran una buena práctica de programación.

Quizá se pregunte por qué se considera como una mala práctica de programación. Reflexione en lo siguiente: Suponga que está escribiendo un programa con un número considerable de líneas de código, digamos unas 300, 5 000 o 13 000 líneas de código, y en algunas de ellas hace uso de constantes o números mágicos para controlar ciclos, longitudes, capacidades o algo parecido, y que

```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

Figura 3.6: Salida de los ejemplos de ciclos **while**, **do-while** y **for**

por alguna razón (y esa razón aparecerá), las constantes o números mágicos utilizados tienen que cambiar, quizá todos, quizá sólo algunos, ¿a qué situación se enfrenta?, ¿es posible realizar los cambios sin efectos colaterales?, ¿recordará todas las líneas que contenían la constante a modificar?

Si bien es cierto que puede utilizar el IDE o un editor de texto para realizar búsquedas y apoyarse de ellas para realizar los cambios, la latencia a introducir errores sigue presente, debido a que no deja de ser una búsqueda manual, y a que los seres humanos somos propensos a cometer errores, por ello, es mejor dejarle toda la responsabilidad a la computadora, específicamente al compilador, y siendo más precisos y específicos: al pre procesador del compilador del lenguaje C.

Tomando en cuenta esta propuesta, el Ejemplo 3.9 muestra el uso de la directiva del pre procesador **#define**, en base a lo descrito con anterioridad y al Ejemplo 3.8.

Compile el ejemplo, ejecútelo (no lo mate, sólo póngalo a funcionar), y asegúrese que la salida es la misma que la indicada en la Figura 3.6.

Tanto el Ejemplo 3.8 como el Ejemplo 3.9 hacen uso en las líneas 12 y 15 respectivamente, de la expresión:

$$i = i + 1;$$

la cual es una operación de **incremento**, también conocida como expresión de **acumulación**. Dicha expresión se utiliza normalmente para modificar la variable de control, que para el caso de ambos ejemplos es *i*. Se le denomina **variable de control** porque en función de ella se escribe la expresión condicional del ciclo, y de alguna manera controla la continuación o la terminación del ciclo.

```

1  /* Ejemplo de la estructura de control de repeticion while
2     y la directiva define.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  #define NUM 10
8
9  int main() {
10     int i;           /* variable de control */
11
12     i = 1;           /* inicializacion de la variable de control */
13     while( i <= NUM) { /* expresion condicional */
14         printf("i = %d\n", i);
15         i = i + 1;    /* modificacion de la variable de control */
16     }
17     return 0;
18 }

```

Ejemplo 3.9: Uso de la estructura de control **while** y la directiva **define**

El lenguaje de programación C incorpora el **operador de incremento** (**++**) para simplificar éste tipo de expresiones de incremento, por lo que las líneas 12 y 15 de los Ejemplos 3.8 y 3.9 respectivamente, pueden ser substituidas por:

$$\begin{array}{c}
 i++; \\
 \text{ó} \\
 ++i;
 \end{array}$$

En la sección de ejercicios se le pedirá que realice dicho cambio, para que compruebe el funcionamiento del operador.

El operador de incremento puede ser utilizado como **prefijo** (antes de la variable) o como **postfijo** (después de la variable), y en ambos casos el efecto es el mismo⁵: **incrementar la variable en una unidad**.

Si el operador de incremento se utiliza como prefijo, a la expresión se le denomina de **pre incremento**, debido a que el valor de la variable es incrementado **antes** de que su valor se utilice, en tanto que si el operador es utilizado como postfijo, el valor de la variable es incrementado **después** de que su valor se ha utilizado, y a esta expresión se le denomina de **pos incremento**. Lo anterior significa que, en un contexto en donde el valor de la variable afectada está siendo utilizado, el pre incremento y el pos incremento tienen resultados distintos.

⁵Utilizado como expresión aislada (es decir, sin usarse dentro de otra expresión).

```
1  /* Programa para mostrar la diferencia del
2     pre-incremento y el post-incremento.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int main() {
8      int m, n;
9
10     n = 5;
11     m = n++;
12     printf("m = %d y n = %d\n", m, n);
13
14     n = 5;
15     m = ++n;
16     printf("m = %d y n = %d\n", m, n);
17
18     return 0;
19 }
```

Ejemplo 3.10: Programa para mostrar la diferencia entre el pre incremento y el pos incremento

El Ejemplo 3.10 muestra la diferencia de utilizar el operador de incremento como prefijo y como postfijo. En base a la explicación del párrafo anterior, determine la salida del programa antes de ejecutarlo, y asegúrese de entender lo que sucede antes de seguir.

Finalmente, se debe mencionar que C incorpora también un operador que funciona de manera análoga para los decrementos, el **operador de decremento** (`--`), el cual sigue las mismas reglas que se comentaron para el operador de incremento. En la sección 3.4, se revisarán otros ejemplos y se trabajará más con la estructura de repetición **while** y con los operadores de incremento y decremento.

3.2.2. La estructura de control do-while

La estructura de control o ciclo **do-while**, tiene la siguiente estructura general:

```
do {
    sentencia(s);
} while (expresión);
```

El ciclo **do-while** procesa o ejecuta el grupo de *sentencia(s)* delimitadas por su bloque de manera repetida, **mientras** la *expresión*, al ser evaluada sea distinta de cero, es decir: *expresión* $\neq 0$.

La llave izquierda “{” delimita el inicio de su bloque, mientras que la llave derecha “}” delimita su final.

El conjunto de *sentencia(s)* delimitadas por el bloque del ciclo **do-while** se procesan **al menos una vez**, y cuando la secuencia de ejecución alcanza el indicador de fin del bloque y la palabra reservada **while**, se evalúa la *expresión*. Si *expresión* $\neq 0$, la estructura **do-while** modifica el flujo de ejecución secuencial para que brinque automáticamente a la palabra reservada **do**, y se repita el procesamiento de las *sentencia(s)* delimitadas por su bloque. En caso contrario, se procesa la siguiente sentencia que se encuentre después de la palabra reservada **while**.

```

1  /* Ejemplo de la estructura de control de
2     repetición do-while
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  #define NUM 10
8
9  int main(){
10     int i;    /* variable de control */
11
12     i = 1;    /* inicialización */
13     do{
14         printf("i = %d\n", i);
15     }while(i++ < NUM); /* expresión condicional
16                        y modificación */
17     return 0;
18 }

```

Ejemplo 3.11: Uso de la estructura de control **do-while**

Observe que la descripción y el funcionamiento de la estructura **do-while** es similar al de la estructura **while**, con la diferencia de que la *expresión* en el ciclo **while** es evaluada **al principio**, mientras que en el ciclo **do-while** es evaluada **al final**. En base a esto, se tiene que:

- Las sentencias de la estructura de repetición **while** se repiten de 0 a *n-veces*.
- Las sentencias de la estructura de repetición **do-while** se repiten de 1 a *n-veces*.

En esencia, ésta es la única diferencia que existe entre las estructuras de repetición **while** y **do-while**. No olvide tenerlo en mente.

Un ejemplo de uso de la estructura de repetición **do-while** se muestra en el Ejemplo 3.11. Observe la semejanza con el Ejemplo 3.9, y note que la

principal diferencia radica, aparte del uso del ciclo **do-while**, en la expresión condicional de la línea 15, ya que se hace uso del operador de pos incremento para la variable de control en la expresión condicional, sin embargo, compruebe que la salida es exactamente la misma que la mostrada en la Figura 3.6.

La estructura de repetición **do-while** ha sido relegada y poco apreciada por muchos, pero tiene nichos de aplicación muy específicos y concretos en los que resulta más que conveniente.

```
1  /* Programa para dividir dos enteros (cuarta version).
2     @autor Ricardo Ruiz Rodriguez
3  */
4
5  #include <stdio.h>
6
7  int main() {
8      int num, den, div;
9
10     printf("Numerador?: ");
11     scanf("%d", &num);
12     do{
13         printf("Denominador (!=0)?: ");
14         scanf("%d", &den);
15     }while(den == 0);
16
17     div = num / den;
18     printf("%d / %d = %d\n", num, den, div);
19
20     return 0;
21 }
```

Ejemplo 3.12: Uso de la estructura de control **do-while** para la validación de datos

El Ejemplo 3.12 retoma el problema de indeterminación para la división discutido en la sección 3.1.1, pero solucionado desde otro enfoque: repetir la solicitud del denominador y su respectiva lectura, **mientras** el denominador sea igual a cero, que es precisamente lo contrario a lo que se requiere.

Aunque al principio lo anterior podría parecer contradictorio, si se analiza y reflexiona con detenimiento, la aparente contradicción desaparece: el ciclo **do-while** de las líneas 12-15 repiten el *prompt* cuando no se obtiene lo deseado (que el denominador sea distinto de cero), es una lógica inversa, pero el truco está en entender que **no se desea que se repita el *prompt***, pero si el denominador no es como se espera (distinto de cero), entonces se tiene que repetir el *prompt*. Por lo tanto, la expresión condicional se expresa en términos de dicha situación, la negación de *distinto de cero* es *igual a cero*.

```

Numerador?: 10
Denominador (!=0)?: 0
Denominador (!=0)?: 0
Denominador (!=0)?: 0
Denominador (!=0)?: 3
10 / 3 = 3

```

Figura 3.7: Salida del Ejemplo 3.12

Observe que la línea 17 puede ejecutarse ya sin ningún tipo de verificación, ya que cuando la estructura de control secuencial la procese, es porque la validación hecha en el ciclo **do-while** se aseguró de que el denominador es distinto de cero, ya que en caso contrario, el flujo de control seguiría dentro del ciclo.

Por último, es importante mencionar que es posible reescribir todo lo que se escriba con un ciclo **while** con un ciclo **do-while** y viceversa, pero existen nichos específicos en los que uno conviene más que el otro, y la lectura de datos acompañada de su respectiva validación se expresan de manera más natural con una estructura de repetición **do-while**. En la sección de ejercicios tendrá la oportunidad de corroborar la afirmación hecha en este párrafo y generar su propia conclusión.

3.2.3. La estructura de control for

La estructura de control o ciclo **for** tiene la siguiente estructura general:

```

for ( expresión1; expresión2; expresión3 ) {
    sentencia(s);
}

```

El ciclo **for** procesa o ejecuta el grupo de *sentencia(s)* delimitadas por su bloque de manera repetida, **mientras** la *expresión2*, al ser evaluada, sea distinta de cero, es decir: $expresión2 \neq 0$.

La llave izquierda “{” delimita el inicio de su bloque, mientras que la llave derecha “}” delimita su final.

Si se procesan el conjunto de *sentencia(s)* delimitadas por el bloque del ciclo **for**, cuando la secuencia de ejecución alcanza el indicador de fin del bloque, la estructura de repetición **for** modifica el flujo de ejecución secuencial

y brinca automáticamente a la evaluación de la *expresión3*, la cual habitualmente modifica la(s) variable(s) de control; después de esto, el flujo de control se pasa a *expresión2*, y si *expresión2* $\neq 0$, se repite el procesamiento de las *sentencia(s)* delimitadas por el bloque. En caso contrario, se procesa la siguiente sentencia que se encuentre después del indicador de fin de bloque del ciclo **for**⁶.

Cabe mencionar aquí que en un ciclo **for**, cualquiera de las expresiones: *expresión1*, *expresión2*, o *expresión3*, pueden ser vacías. Si las tres son vacías, se genera un **ciclo infinito**.

```
1  /* Ejemplo de la estructura de repeticion for.
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  #define NUM 10
7
8  int main() {
9      int i;
10
11     /* inicializacion, expresion, y modificacion */
12     for(i = 1; i <= NUM; i++)
13         printf("i = %d\n", i);
14
15     return 0;
16 }
```

Ejemplo 3.13: Uso de la estructura de repetición **for**

La estructura de repetición **for** es equivalente a la estructura de repetición **while** en el siguiente sentido:

```
expresión1;
while (expresión2) {
    sentencia(s);
    expresión3;
}
```

por lo que puede decirse, que cualquier ciclo escrito en una estructura de repetición **while** puede ser reescrito en la estructura de repetición **for** y viceversa,

⁶Es un error común de programación terminar por inercia una estructura de repetición **for** con “;”, esto no es ningún error respecto a la gramática del lenguaje, por lo que el compilador no identificará nada extraño. Revise la sección de ejercicios para profundizar un poco más en éste aspecto.

de tal forma que es posible mantener la equivalencia lógica y funcional entre una y otra estructura.

El Ejemplo 3.13 muestra el uso de la estructura de repetición **for**, el cual genera la misma salida mostrada en la Figura 3.6. Asegúrese de comprender su funcionamiento.

$$\sum_{n=1}^n i \quad (3.1)$$

Considere ahora el Ejemplo 3.14, el cual utiliza una estructura **for** un poco más elaborada que la anterior, para implementar la sumatoria de los números entre 1 y n , expresada en la Ecuación 3.1

```

1  /* Sumatoria de n numeros.
2  @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  int main() {
7      int sum, n, i;
8
9      printf("Programa para calcular la sumatoria\n");
10     printf("de i, con i = 1 hasta n.\n");
11     do{
12         printf("n (> 0)? : ");
13         scanf("%d", &n);
14     }while(n < 1); /* verifica que n sea valido */
15
16     for(i = 1, sum = 0; i <= n; i++){
17         sum += i;
18
19     printf("La sumatoria de i, con i = 1 hasta %d es: %d\n", n, sum);
20
21     return 0;
22 }
```

Ejemplo 3.14: Sumatoria de n números enteros

Observe que es posible inicializar más de una variable en la *expresión1* del ciclo **for** (línea 16): la variable de control i , que sirve para ir generando la serie de números, mientras que la variable sum , se utiliza como acumulador. De hecho, es posible tanto inicializar como modificar más de una variable de control en *expresión1* y *expresión3* respectivamente, y la forma de hacerlo es escribir dichas expresiones con una lista de variables separadas por comas.

La expresión de la línea 17 del Ejemplo 3.14 puede interpretarse en nuestro lenguaje como: “A la variable sum incrementala en i ”, debido a que en C, la expresión:

i = 1	j = 10
i = 2	j = 9
i = 3	j = 8
i = 4	j = 7
i = 5	j = 6
i = 6	j = 5
i = 7	j = 4
i = 8	j = 3
i = 9	j = 2
i = 10	j = 1

Figura 3.8: Salida del Ejemplo 3.15

variable = variable *operador* expresión;

es equivalente a

variable *operador* = expresión;

donde *operador* es cualquier operador aritmético válido en C (véase la Tabla 2.1), y se le denomina **notación compacta**. Es muy común escribir éste tipo de expresiones usando dicha notación.

```

1  /* Ejemplo de uso del ciclo for con dos variables de control.
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  #define M 10
7
8  int main() {
9      int i, j;
10
11     for(i = 1, j = M; i <= M; i++, j--)
12         printf("\ti = %d\tj = %d\n", i, j);
13
14     return 0;
15 }
```

Ejemplo 3.15: Estructura de repetición **for** con dos variables de control

Para finalizar esta sección se describirán dos ejemplos más. El Ejemplo 3.15, muestra el uso de la estructura de repetición **for** con dos variables de control *i* y *j*, el ciclo puede ser gobernado por ambas variables si el problema a resolver así lo requiere, pero para este ejemplo, sólo se imprimen las variables de control en orden ascendente y descendente respectivamente, como se muestra en la Figura 3.8.

Observe que la *expresión1* del ciclo **for** (línea 11), contiene la inicialización de las dos variables de control separadas por una coma (,). De manera

análoga, la *expresión*3 contiene la modificación, incremento ($i++$) y decremento ($j--$) respectivamente, de las variables de control, separadas también por una coma.

```

1  /* Uso de rand para generar numeros aleatorios.
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #define M 15
9
10 int main() {
11     int i;
12
13     srand(time(NULL)); /* Inicializa la semilla */
14     for(i = 1; i <= M; i++)
15         printf("%c", 'a' + rand() % 26);
16     printf("\n");
17
18     return 0;
19 }
```

Ejemplo 3.16: Uso de la funciones **srand** y **rand** para generar números aleatorios

Por otro lado, el Ejemplo 3.16 genera una secuencia de quince caracteres (véanse las línea 8, 14 y 15) aleatorios. El programa incluye en la línea 5 la biblioteca estándar **stdlib.h**, la cual es necesaria para poder utilizar las funciones **srand** y **rand** de las líneas 13 y 15 respectivamente.

La biblioteca **time.h** de la línea 6, se ha incluido para poder hacer uso de la función **time** de la línea 13, la cual permite leer la fecha y hora del sistema en donde se ejecuta el programa, y regresa un número de tipo **size_t** el cual es capaz de almacenar los segundos que han transcurrido desde las cero horas del primero de Enero de 1970, y que es utilizado por la función **srand** para inicializar la semilla de números aleatorios con un valor distinto en cada ejecución.

Los números aleatorios son generados por alguna función matemática, y para generarse, se basan en un valor inicial (semilla), si éste valor no cambia, la secuencia de números generados tampoco. Por lo tanto, la función **srand** es la responsable de cambiar la secuencia de generación de números aleatorios a partir del valor semilla, que para el caso del Ejemplo 3.16, utiliza la hora del sistema cada vez que se ejecuta el programa.

Ahora bien, la función encargada de generar el número aleatorio a partir de la inicialización que hizo la función **srand**, es la función **rand**, (línea 15).

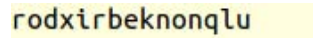


Figura 3.9: Salida del Ejemplo 3.16

La función **rand** genera un número aleatorio entre **0** y **RAND_MAX**, el cual es un valor dependiente de la implementación de la biblioteca (debido a las diferentes arquitecturas de *hardware* y *software*), pero se garantiza que es de al menos 32,767 en cualquier implementación de la biblioteca que se base en el estándar ANSI C.

Observe que el valor de la función **rand** es dividido por 26, y que el residuo de dicha división es sumado al carácter 'a' (línea 15). Por extraño que parezca (sumarle un número a una letra), esto tiene sentido al menos en C, y se explica de la siguiente manera:

Sea n el número que representa el código del carácter 'a'⁷, y sea d definido como:

$$d = \text{rand}() \% 26$$

y c como:

$$c = n + d$$

entonces:

$$c \in \{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'\}$$

El valor de c representado como carácter, es lo que se imprime en el especificador de formato “%c”, el cual es el especificador de formato en C para los caracteres (**char**).

En resumen, el Ejemplo 3.16 genera una secuencia de quince caracteres aleatorios. Antes de avanzar a la siguiente sección, asegúrese de comprender lo que se ha descrito hasta aquí, y por qué el Ejemplo 3.16 genera una salida como la de la Figura 3.9. Pruebe con varias ejecuciones, quizá alguna de ellas contenga una palabra medianamente comprensible.

⁷Recuerde que lo que la computadora entiende, procesa y “ve” son sólo números, de hecho el valor del carácter 'a' es traducido por el compilador a su representación numérica, pero para nosotros como seres humanos, es más fácil y claro de entender el símbolo 'a', que el código numérico que lo representa; y aunque se podría poner directamente el código del carácter 'a', el precio a pagar sería la portabilidad a otros sistemas, así como la problemática de los números mágicos discutida con anterioridad.

3.3. Estructuras de control combinadas

Pasemos ahora a un programa un poco más elaborado. El Ejemplo 3.17, utiliza la estructura de selección **switch** anidada dentro de una estructura de repetición **while**; también se utiliza la función **getchar** (línea 13), la cual regresa un carácter leído desde la entrada estándar.

```

1  /* Este programa cuenta las vocales a partir de un conjunto
2  de simbolos procesados de la entrada estandar.
3  @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int main(){
8      int simbolo;
9      int as = 0, es = 0, is = 0, os = 0, us = 0;
10
11     printf("Introduza un texto (EOF para terminar):\n");
12
13     while((simbolo = getchar()) != EOF){ /* EOF = End Of File */
14         switch (simbolo) { /* switch anidado dentro de while */
15             case 'A': case 'a': /* Cuenta las a's */
16                 as++;
17                 break;
18             case 'E': case 'e': /* Cuenta las e's */
19                 es++;
20                 break;
21             case 'I': case 'i': /* Cuenta las i's */
22                 is++;
23                 break;
24             case 'O': case 'o': /* Cuenta las o's */
25                 os++;
26                 break;
27             case 'U': case 'u': /* Cuenta las u's */
28                 us++;
29                 break;
30             default: /* ignora todos los otros caracteres */
31                 break;
32         }
33     }
34
35     printf("\nTotal de vocales:\n");
36     printf("a's: %d\te's: %d\ti's: %d\t o's: %d\tu's: %d\n",
37         as, es, is, os, us);
38
39     return 0;
40 }

```

Ejemplo 3.17: Programa para mostrar la combinación y anidamiento de estructuras de control

El *prompt* de la línea 11 indica que para terminar la entrada de datos se escriba *EOF*, el cual es un acrónimo de *End Of File*.

```
Introduza un texto (EOF para terminar):  
Esto es una línea de prueba  
  
Total de vocales:  
a's: 3 e's: 5 i's: 1 o's: 1 u's: 2
```

Figura 3.10: Salida del Ejemplo 3.17

Si los datos se procesan desde un archivo (y para este ejemplo es posible sin realizar cambio alguno en el código), la marca EOF se encontrará al final del mismo; pero si los datos se procesan desde el teclado, la marca de EOF tendrá que ser simulada: en GNU/Linux se simula con la combinación de teclas *Ctrl+D*, mientras que en Windows la combinación es *Ctrl+Z*.

Observe la sentencia de la línea 13, la cual es un ejemplo muy usual de la forma de escribir expresiones y sentencias en C, ya que la expresión condicional del ciclo **while** está combinada con una expresión de asignación. La sentencia podría interpretarse en nuestro lenguaje como: “Obtén el siguiente carácter de la entrada estándar, asígnalo a *simbolo*, y si es distinto de EOF, ejecuta el grupo de sentencias del ciclo”.

Note que la variable *simbolo* es de tipo entero (**int**). Esto se debe a que la función **getchar** regresa el número entero que representa al símbolo (carácter) procesado.

Por otro lado, la estructura **switch** se encarga de hacer una comparación del dato contenido en *simbolo*, contra los casos (**case**) de interés: las vocales mayúsculas o minúsculas. Note también la forma en que se han expresado los casos (**case**), ya que no se ha hecho uso de ningún operador lógico (de hecho, intencionalmente no han sido presentados), pero las sentencias de las líneas 15, 16 y 17 podrían interpretarse en nuestro lenguaje como: “En caso de que *simbolo* sea una 'A' o una 'a', incrementa el contador de a's y termina la comparación”. Las líneas 18-29 tendrían una interpretación análoga.

Observe también que los símbolos 'A' y 'a' han sido puestos entre comillas simples, lo cual, además de auto documentar mejor el código fuente, sirve para que el compilador traduzca el código de dichos símbolos, en su correspondiente representación numérica, y puedan así ser comparados con el valor regresado por la función *getchar*.

En resumen, la estructura **while** se encarga de repetir la lectura de datos y de enviarle los datos a la estructura **switch** mientras el dato procesado sea distinto de EOF; por otro lado, la estructura **switch** se encarga de determinar

Operador	Descripción
&&	Conjunción (AND)
	Disyunción (OR)
!	Negación (NOT)

Tabla 3.2: Operadores lógicos en C

si el dato procesado es o no una vocal. Si lo es, la contabiliza en el contador correspondiente, y si no, lo ignora (**default** de la línea 30).

Una muestra de la ejecución del Ejemplo 3.17 con datos procesados desde el teclado se muestra en la Figura 3.10.

Hasta ahora ha sido posible resolver todos los ejemplos y ejercicios sin el uso de operadores lógicos, de hecho esa ha sido la intención, sin embargo, el uso de operadores lógicos resulta indispensable para resolver de manera más sencilla diferentes tipos de problemas. La Tabla 3.2 muestra los **operadores lógicos** en C.

A partir de ahora, siéntase libre de usar los operadores lógicos para resolver los ejercicios de este capítulo. Si los usó antes, ha cometido una violación a la metodología de aprendizaje del libro, por lo que si no quiere ser condenado por toda la eternidad, debería repetir los ejercicios en los que utilizó los operadores lógicos, suponiendo que no existen, el reto le resultará, espero, interesante.

3.4. Tipos de repetición

Independientemente del tipo de estructura de control que se utilice para la repetición de un grupo de sentencias, existen básicamente dos **tipos de repetición**:

1. Repetición controlada por contador.
2. Repetición controlada por centinela.

Ambos tipos de repetición pueden ser implementados en C utilizando cualquiera de las estructuras de control **while**, **do-while** y **for**, por lo que no hay una relación de correspondencia única entre el tipo de repetición y la estructura de control de repetición que se utilice. Sin embargo, debido a la

naturaleza y al funcionamiento de las estructuras de control de repetición, el enfoque de repetición controlada por contador se implementa de manera más natural utilizando los ciclos **while** y **for**; mientras que la repetición controlada por centinela, se implementa de forma mucho más natural utilizando la estructura de control de repetición **do-while**, pero esto es más una conveniencia que una asociación.

3.4.1. Repetición controlada por contador

Los Ejemplos 3.8 y 3.9, además de mostrar el funcionamiento de la estructura de control de repetición **while**, son ejemplos clave del tipo de repetición controlada por contador. La variable *i* tiene el rol de **contador** debido a que registra cuántas veces se ha repetido el ciclo.

```
1  /* Programa que calcula el promedio de unas calificaciones
2     usando repeticion controlada por contador.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define N 10
7
8  int main(){
9      int contador;
10     float calificacion , total;
11
12     total = 0.0;
13     contador = 1;
14
15     while(contador++ <= N){
16         printf(" Calificacion?: ");
17         scanf("%f", &calificacion);
18         total += calificacion;
19     }
20
21     printf("Promedio de las calificaciones: %.1f\n", total/N);
22
23     return 0;
24 }
```

Ejemplo 3.18: Promedio de calificaciones utilizando repetición controlada por contador

A manera de guía, siempre que se sepa o se pueda intuir a partir de la descripción del problema a resolver cuántas veces se va a repetir un grupo de sentencias, sin importar que el número de repeticiones sea muy pequeño o muy grande, tendremos la situación de un tipo de **repetición controlada por contador** , como por ejemplo:

```
Calificacion?: 1.1
Calificacion?: 1.2
Calificacion?: 1.3
Calificacion?: 1.4
Calificacion?: 1.5
Calificacion?: 1.6
Calificacion?: 1.7
Calificacion?: 1.8
Calificacion?: 1.9
Calificacion?: 2
Promedio de las calificaciones: 1.5
```

Figura 3.11: Una posible salida del Ejemplo 3.18

- Determinar el promedio de n calificaciones, donde $n > 1$.
- Determinar el mayor de n números, donde $n > 1$.
- Determinar la suma de los primeros n números enteros positivos ($n > 1$).
- Determinar la suma de los primeros n números pares, donde $n > 1$.
- Determinar la suma de números pares contenidos entre 1 y n , donde $n > 1$.
- Identificar los números primos entre 1 y n , donde $n > 1$, etc.

En todos estos ejemplos es posible conocer, antes de escribir una sola línea de código, el número de repeticiones que se tendrá.

El Ejemplo 3.18 es una muestra clave de la repetición controlada por contador. Todas las líneas deberían ser completamente comprendidas, por lo que debería responder ¿cuántas veces se repetirá el ciclo de la línea 15?, ¿puede determinar esto sin ejecutar el programa?

Observe que en la línea 17 se ha puesto el especificador de formato “%f”, el cuál le indica a la función **scanf** que se va a leer un número de tipo flotante (**float**). Por otro lado, el mismo especificador en la línea 21, le indica a la función **printf** que va a imprimir un número de tipo **float** en la salida estándar, pero note que entre el “%” y la “f” hay un valor: “.1”, el cual le indica a la función **printf** que imprima el número con **un** decimal después del punto. Pruebe con otros valores, como “.2”, “.3” o “.4”, recompile y vea lo que sucede (vea la sección 2.2).

Una posible salida para el Ejemplo 3.18 se muestra en la Figura 3.11.

3.4.2. Repetición controlada por centinela

No en todos los problemas podemos conocer de manera anticipada, cuantas veces se repetirá un grupo de sentencias. Es en este tipo de situaciones cuando tenemos el tipo de **repetición controlada por centinela**.

El **centinela** es un valor que se utiliza para determinar la continuidad o no del ciclo, a partir de la evaluación de la expresión condicional del mismo, es decir, el centinela determina si el grupo de sentencias asociadas se procesarán o no nuevamente. Por lo general, el valor del centinela forma parte de la expresión condicional de la estructura de control de repetición, y puede estar dado de manera explícita o implícita.

Considere el ejercicio de la división y su solución del Ejemplo 3.12, ahí el centinela está implícito, debido a que la descripción del problema no contempla la designación del cero como centinela, sino que la naturaleza misma del problema lo requiere, ante la posible indeterminación de la división.

En otros casos, el centinela se describe de manera explícita, como en los siguientes ejemplos:

- Determinar el promedio de una lista de calificaciones, en donde cada calificación será introducida desde el teclado una a la vez. El fin de la lista de calificaciones será indicado con -1, debido a que la lista es variable.
- Determinar la suma de los números enteros proporcionados desde el teclado, hasta que el número proporcionado sea cero.
- Dada una lista de números enteros positivos, determine el mayor y el menor de ellos, la lista termina al proporcionar cualquier número negativo.

Aunque la repetición controlada por centinela es menos frecuente, no es menos necesaria ni menos importante que la repetición controlada por contador, de ahí la importancia de presentarla, conocerla y dominarla, como parte de su repertorio como programador, y de las técnicas utilizadas en la resolución de problemas.

Considere el Ejemplo 3.19, el cual resuelve el mismo problema que el Ejemplo 3.18 pero utilizando un enfoque diferente: el de la repetición controlada por centinela. Una vez más, todas las sentencias de este programa deben ser completamente comprendidas.

```
1  /* Programa que calcula el promedio de unas calificaciones
2     usando repetición controlada por centinela.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define CENTINELA -1
7
8  int main(){
9      int contador;
10     float calificacion , total;
11
12     total = 0;
13     contador = 0;
14
15     printf(" Calificacion? (-1 para terminar): ");
16     scanf("%f", &calificacion);
17
18     while(calificacion != CENTINELA){
19         total += calificacion;
20         contador++;
21         printf(" Calificacion? (-1 para terminar): ");
22         scanf("%f", &calificacion);
23     }
24
25     if(contador != 0)
26         printf("Promedio de las calificaciones: %.1f\n", total/contador);
27     else
28         printf("No se proporcionaron calificaciones\n");
29
30     return 0;
31 }
```

Ejemplo 3.19: Promedio de calificaciones utilizando repetición controlada por centinela

Note que en la repetición controlada por centinela, podría ser que el primer dato procesado fuera el centinela, por lo que no habría calificaciones a promediar, lo cual no es posible de hacer en la repetición controlada por contador, en donde se está obligado a proporcionar las N (diez) calificaciones, sin embargo, es posible simular el mismo tipo de ejecución, tal y como se muestra en la Figura 3.12. Compare los ejemplos y sus correspondientes salidas, y asegúrese de entenderlos.

```
Calificacion? (-1 para terminar): 1.1
Calificacion? (-1 para terminar): 1.2
Calificacion? (-1 para terminar): 1.3
Calificacion? (-1 para terminar): 1.4
Calificacion? (-1 para terminar): 1.5
Calificacion? (-1 para terminar): 1.6
Calificacion? (-1 para terminar): 1.7
Calificacion? (-1 para terminar): 1.8
Calificacion? (-1 para terminar): 1.9
Calificacion? (-1 para terminar): 2
Calificacion? (-1 para terminar): -1
Promedio de las calificaciones: 1.5
```

Figura 3.12: Salida del Ejemplo 3.19

3.4.3. Repeticiones híbridas

El lenguaje de programación C incorpora dos sentencias bastante prácticas y útiles en diferentes contextos: **break** y **continue**. Dichas sentencias permiten implementar un tipo de repetición híbrida, en cuanto a que es posible construir ciclos que incorporen los mecanismos de repetición controlada por contador y por centinela de manera combinada.

```
1  /* Programa que muestra el uso de la sentencia continue.
2  @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #define NUM 10
6  #define CENTINELA 5
7
8  int main() {
9      int i;
10
11      for(i = 1; i <= NUM; i++){
12          if(i == CENTINELA)
13              continue;
14          printf("\ti = %d\n", i);
15      }
16
17      return 0;
18 }
```

Ejemplo 3.20: Uso de la sentencia **continue**

El programa del Ejemplo 3.20 muestra el uso de la sentencia **continue** dentro de un ciclo **for**.

```
i = 1  
i = 2  
i = 3  
i = 4  
i = 6  
i = 7  
i = 8  
i = 9  
i = 10
```

Figura 3.13: Salida del Ejemplo 3.20

En principio, la estructura de repetición **for** está gobernada por un tipo de repetición controlada por contador, y en principio, dicho ciclo se repetirá *NUM* veces. Sin embargo, la sentencia **continue** de la línea 13 indica un cambio en el flujo de ejecución secuencial, de tal forma que cuando el flujo de ejecución procesa la sentencia **continue**, el flujo de ejecución brinca a la verificación de la *expresión3*, para después seguir con la secuencia de ejecución usual de la estructura **for**, es decir, hacia la validación de la expresión condicional (*expresión2*)⁸.

En resumen, el ciclo **for** se repite *NUM* veces, pero no así todas las sentencias del ciclo, ya que cuando el flujo de ejecución alcanza la sentencia **continue**, las sentencias posteriores a ella serán ignoradas, por lo que la salida del Ejemplo 3.20, es la impresión de los números del uno al diez, uno por renglón, excepto el *CENTINELA*, tal y como se muestra en la Figura 3.13.

Ahora bien, el Ejemplo 3.21 muestra una situación análoga. Para empezar, note que dicho ejemplo es exactamente el mismo que el Ejemplo 3.20 excepto por la línea 13, donde se ha cambiado la sentencia **continue** por la sentencia **break**.

La sentencia **break** ya había sido discutida en la sección 3.1.4, donde se utilizaba en el contexto de la estructura de selección **switch**, y servía para romper el flujo de ejecución y brincar al final de dicha estructura de selección. Para el caso del ciclo **for** funciona de manera análoga, ya que cuando el flujo de ejecución la procesa, es como si se diera un salto a la expresión condicional (*expresión2*) del ciclo y ésta fuera evaluada como cero (falsa), lo que implicaría la terminación del ciclo.

⁸Recuerde la estructura general y el funcionamiento del ciclo **for**.

```
i = 1  
i = 2  
i = 3  
i = 4
```

Figura 3.14: Salida del Ejemplo 3.21

En términos prácticos es válido decir que la sentencia **break** rompe el ciclo **en el que se encuentre la sentencia** sin importar la expresión condicional que lo gobierna.

Observe que en principio el ciclo **for** del Ejemplo 3.21 está también gobernado, al menos en su expresión condicional, por un tipo de repetición controlada por contador, y que en principio se repetirá *NUM* veces. Sin embargo, cuando el valor del contador coincida con el del *CENTINELA*, la sentencia **break** será procesada y el ciclo terminará, por lo que el efecto del programa es ver los números del uno al *CENTINELA-1*, uno por renglón, tal y como se muestra en la Figura 3.14.

```
1  /* Programa que muestra el uso de la sentencia break  
2     @autor Ricardo Ruiz Rodriguez  
3  */  
4  #include <stdio.h>  
5  #define NUM 10  
6  #define CENTINELA 5  
7  
8  int main() {  
9      int i;  
10  
11     for(i = 1; i <= NUM; i++){  
12         if(i == CENTINELA)  
13             break;  
14         printf("\ti = %d\n", i);  
15     }  
16  
17     return 0;  
18 }
```

Ejemplo 3.21: Uso de la sentencia **break**

3.5. Ejercicios

1. Escriba el Ejemplo 3.1 utilizando los delimitadores de bloque para cada una de las estructuras de selección **if**.
2. Siguiendo la idea expresada en el texto de la Sección 3.1.1, escriba en nuestro lenguaje la descripción de las sentencias de las líneas 23, 26 y 29 del Ejemplo 3.1.
3. Reescriba el Ejemplo 3.6 usando la simplificación de líneas descrita en la Sección 3.1.3 y compruebe su equivalencia, ¿cuál es mejor y por qué?, ¿en qué casos convendría utilizar una u otra?.
4. Reescriba el Ejemplo 3.6 usando una estructura de selección **if-else**.
5. Escriba un programa que utilice el operador ternario para determinar el menor de dos números enteros. Este programa es similar al del Ejemplo 3.6.
6. Escriba un programa que determine si un número entero positivo es par o impar. Un número par es aquel que es divisible por dos.
7. Escriba un programa que utilice una estructura de selección **if-else** y los operadores relacionales, para determinar el **menor** de tres números enteros (asuma que los números son distintos).
8. Escriba un programa que utilice una estructura de selección **if-else** y los operadores relacionales, para determinar el **mayor** de tres números enteros (asuma que los números son distintos).
9. Escriba un programa que utilice una estructura de selección **if-else** y los operadores relacionales, para determinar el **número central** (en base a su ordinalidad) de tres números enteros (asuma que los números son distintos).
10. Extienda el Ejemplo 3.7 para que el rango de conversión de números a palabras sea más amplio: agregue las sentencias necesarias de tal forma que el programa considere al menos números del uno al diez, ¿qué sucede si en lugar de un número escribe una letra?.
11. Convierta el Ejemplo 3.7 y el ejercicio anterior, a su correspondiente representación con una estructura de selección if-else anidada.

12. Reescriba los Ejemplos 3.8 y 3.9 utilizando el operador de incremento como **prefijo**.
13. Reescriba los Ejemplos 3.8 y 3.9 utilizando el operador de incremento como **postfijo**.
14. Escriba un programa que imprima en la salida estándar los números del uno al diez, uno por renglón pero en orden inverso, es decir: 10, 9, 8, 7, ..., 1. **No** utilice el operador de decremento.
15. Escriba un programa que imprima los números del uno al diez, uno por renglón pero en orden inverso, es decir: 10, 9, 8, 7, ..., 1. Utilice el operador de decremento.
16. Reescriba el Ejemplo 3.9 usando la estructura de repetición **do-while**.
17. Reescriba todos los ejemplos y ejercicios que utilizan la estructura de repetición **while**, utilizando ahora la estructura de repetición **for**.
18. Considere el Ejemplo 3.13, y coloque al final de la línea 12 un “;”, compile el programa y ejecútelo. Lo único que verá en la salida estándar es: “**i = 11**”. ¿Por qué?
19. Considere el programa del Ejemplo 3.22 y determine su salida sin ejecutarlo en la computadora; después corrobore su respuesta con la salida real.

```
1  /* Que hace este programa?
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  #define M 10
7
8  int main() {
9      int i;
10
11     for (i = 1; i <= M; i++)
12         printf("%c", (i % 2)? '@': '#');
13     printf("\n");
14
15     return 0;
16 }
```

Ejemplo 3.22: Misterioso 1

20. Considere el programa del Ejemplo 3.23 y determine su salida sin ejecutarlo en la computadora; después corrobore su respuesta con la salida real.

```

1  /* Que hace este programa?
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  #define M 10
7
8  int main(){
9      int i, j;
10
11     for(i = 1; i <= M; i++)
12         for(j = 1; j <= M; j++)
13             printf("%c", (i % 2)? '@': '#');
14     printf("\n");
15
16     return 0;
17 }
```

Ejemplo 3.23: Misterioso 2

21. Considere el programa del Ejemplo 3.24 y determine su salida sin ejecutarlo en la computadora; después corrobore su respuesta con la salida real.

```

1  /* Que hace este programa?
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #define M 50
9
10 int main(){
11     int i;
12
13     srand(time(NULL));
14     for(i = 1; i <= M; i++)
15         printf("%c", (rand() % 2)? '@': '#');
16     printf("\n");
17
18     return 0;
19 }
```

Ejemplo 3.24: Misterioso 3

22. Para el programa del Ejemplo 3.23, agregue una llave izquierda ({) al final de la línea 11, y una llave derecha(}) en la línea 15, de tal

forma que se delimite el bloque del ciclo **for** más externo para que las sentencias de las líneas 12-14 pertenezcan a él. ¿Cambia en algo la salida del programa?. Determine su respuesta sin ejecutarlo en la computadora, después ejecútelo, y analice lo que sucede.

23. Para el programa modificado en base a lo expuesto en el ejercicio anterior, cambie la i por j de la línea 13, y determine su salida sin ejecutarlo en la computadora; después corrobore su respuesta con la salida real.
24. Pruebe con más datos el Ejemplo 3.17, proporcionados tanto del teclado como de un archivo, siguiendo los mecanismos de redireccionamiento descritos en el Apéndice B.
25. Para el Ejemplo 3.17, describa cómo podrían interpretarse en nuestro lenguaje las sentencias de las líneas 18-29.
26. Escriba un programa que, siguiendo la idea del Ejemplo 3.17 y considerando un esquema de calificaciones basado en letras (**A**, **B**, **C**, **D** y **F**), procese un conjunto de dichas calificaciones de la entrada estándar, y determine cuantas calificaciones hay de cada una. Si se procesa una calificación no considerada, se deberá indicar dicha irregularidad (no existe la calificación **E** por ejemplo).
27. Escriba un programa que determine si un número entero positivo es o no un número primo. Un número primo es aquel que sólo es divisible por sí mismo y la unidad.
28. Reescriba el Ejemplo 3.18 con una estructura de repetición **do-while** y asegúrese de que sea lógica y funcionalmente equivalente.
29. Reescriba el Ejemplo 3.18 con una estructura de repetición **for** y asegúrese de que sea lógica y funcionalmente equivalente.
30. Modifique el Ejemplo 3.18 para que lea el número de calificaciones a procesar. Tanto el número de calificaciones (mayor que cero), como las calificaciones (entre 1 y 10) deberán ser validados. Utilice la estructura de repetición **while** para la repetición controlada por contador.
31. Modifique el Ejemplo 3.18 para que lea el número de calificaciones a procesar. Tanto el número de calificaciones (mayor que cero), como las

calificaciones (entre 1 y 10) deberán ser validados. Utilice la estructura de repetición **do-while** para la repetición controlada por contador.

32. Modifique el Ejemplo 3.18 para que lea el número de calificaciones a procesar. Tanto el número de calificaciones (mayor que cero), como las calificaciones (entre 1 y 10) deberán ser validados. Utilice la estructura de repetición **for** para la repetición controlada por contador.
33. Reescriba el Ejemplo 3.19 con una estructura de repetición **do-while**, y asegúrese de que sea lógica y funcionalmente equivalente.
34. Reescriba el Ejemplo 3.19 con una estructura de repetición **for** y asegúrese de que sea lógica y funcionalmente equivalente.
35. Modifique el Ejemplo 3.19 para que las calificaciones sean validadas (que estén entre cero y diez). Utilice la estructura de repetición **while** para la repetición controlada por centinela.
36. Repita el ejercicio anterior pero utilice la estructura de repetición **do-while** para la repetición controlada por centinela.
37. Repita el ejercicio anterior pero utilice la estructura de repetición **for** para la repetición controlada por centinela.
38. Escriba un programa que realice la suma de los números del uno al diez:
 - a) Utilice la estructura de repetición **while**.
 - b) Utilice la estructura de repetición **do-while**.
39. Escriba una programa que realice la suma de números enteros procesados desde la entrada estándar. La suma se detiene cuando el número leído sea igual a cero.
 - a) Utilice la estructura de repetición **while**.
 - b) Utilice la estructura de repetición **do-while**.
40. Escriba un programa que permita determinar el mayor de n números proporcionados desde el teclado, donde $n > 1$.

41. Escriba un programa que permita determinar la suma de los primeros n números enteros positivos proporcionados desde el teclado, donde $n > 1$.
42. Escriba un programa que permita determinar la suma de los primeros n números pares ($n > 1$).
43. Escriba un programa que permita determinar la suma de números pares contenidos entre 1 y n , donde $n > 1$.
44. Escriba un programa que permita identificar los números primos entre 1 y n , donde $n > 1$.
45. Dada una lista de números enteros positivos proporcionados desde la entrada estándar, determine el mayor y el menor de ellos. La lista termina al proporcionar cualquier número negativo o cero.
46. ¿Qué conclusiones puede determinar de la realización de los seis ejercicios anteriores?, ¿qué tipo de estructura de repetición es más natural, desde su perspectiva, para una repetición controlada por contador y por qué?, ¿qué tipo de estructura de repetición es más natural, desde su perspectiva, para una repetición controlada por centinela y por qué?
47. Reescriba el programa del Ejemplo 3.20 utilizando una estructura de repetición **while** y determine lo que pasa. ¿Funciona de la misma manera?, ¿se genera algún tipo de problema?, ¿se mantiene la equivalencia lógica y funcional?.
48. Reescriba el programa del Ejemplo 3.20 utilizando una estructura de repetición **do-while** y determine lo que pasa. ¿Funciona de la misma manera?, ¿se genera algún tipo de problema?, ¿se mantiene la equivalencia lógica y funcional?.
49. Reescriba el programa del Ejemplo 3.21 utilizando una estructura de repetición **while** y determine lo que sucede. ¿Funciona de la misma manera?, ¿se genera algún tipo de problema?, ¿se mantiene la equivalencia lógica y funcional?.
50. Reescriba el programa del Ejemplo 3.21 utilizando una estructura de repetición **do-while** y determine lo que sucede. ¿Funciona de la misma

manera?, ¿se genera algún tipo de problema?, ¿se mantiene la equivalencia lógica y funcional?.

Capítulo 4

Módulos de programa: Funciones

Desde el primer ejemplo de este libro se ha estado haciendo uso de funciones: **main** es una función, **printf**, **scanf**, **getchar**, **rand** y **srand** también son funciones, y al uso de una función se le denomina **invocación** o **llamado de función**.

Este capítulo describe los aspectos relacionados con funciones, tanto las de las bibliotecas del lenguaje C como las funciones definidas por el programador, es decir, se presentan los elementos básicos para construir bibliotecas de funciones personalizadas.

4.1. Funciones de biblioteca

Considere el problema de calcular las raíces de una ecuación de segundo grado utilizando la fórmula general mostrada en la Ecuación 4.1.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (4.1)$$

Una posible solución a dicho problema es el programa que se muestra en el Ejemplo 4.1, el cual es el ejemplo más elaborado hasta ahora en cuanto a los elementos que contiene se refiere, y con excepción de las líneas 6, 26 y 27, debería ser completamente comprendido. La línea 6 le indica al compilador que incluya la biblioteca de funciones matemáticas (**math.h**), ya que en las líneas 26 y 27 se hará uso de la función **sqrt** (*square root*), la cual calcula la raíz cuadrada de su argumento.

```

1  /* Programa para calcular las raices reales de una ecuacion
2     cuadratica usando la formula general.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include <math.h>
7
8  int main(){
9      int a, b, c, d;
10     float x1, x2, r1, r2;
11
12     printf("Programa para determinar las raices reales de una ecuacion\n");
13     printf("de la forma Ax^2 + Bx + C = 0\n\n");
14     do{
15         printf("A(!=0)? : ");
16         scanf("%d", &a);
17     }while(a == 0);
18     printf("B? : ");
19     scanf("%d", &b);
20     printf("C? : ");
21     scanf("%d", &c);
22     d = (b * b) - (4 * a * c);
23     if(d < 0){
24         printf("La ecuacion no tiene solucion en R\n");
25     }else{
26         x1 = (-b + sqrt(d)) / (2*a);
27         x2 = (-b - sqrt(d)) / (2*a);
28         printf("\nX1 = %.1f\text{=> }", x1);
29         r1 = a*(x1*x1) + b*x1 + c;
30         printf("%d(%.1f)^2 + %d(%.1f) + %d = %f\n", a, x1, b, x1, c,
31             r1);
32         printf("X2 = %.1f\text{=> }", x2);
33         r2 = a*(x2*x2) + b*x2 + c;
34         printf("%d(%.1f)^2 + %d(%.1f) + %d = %f\n", a, x2, b, x2, c,
35             r2);
36     }
37     return 0;
38 }

```

Ejemplo 4.1: Cálculo de raíces para una ecuación de segundo grado utilizando la fórmula general

Las líneas 26 y 27 muestran el **uso** de una función de biblioteca (**sqrt**), y a dicho uso se le denomina **invocación** o **llamado de función**.

Sin la función **sqrt**, el Ejemplo 4.1 no podría ser resuelto, a menos que esas líneas fueran substituidas por el algoritmo correspondiente para calcular la raíz cuadrada de un número, y no sólo eso, sino que el código en C de dicho algoritmo tendría que ser repetido dos veces al menos, en base a como está escrita la propuesta de solución del Ejemplo 4.1.

Las funciones de las bibliotecas del lenguaje son cajas negras: se puede saber lo que realizan, los datos que necesitan para trabajar, y la salida que


```
Programa para determinar las raices reales de una ecuacion
de la forma Ax^2 + Bx + C = 0

A(!=0)? : 0
A(!=0)? : 1
B? : 5
C? : 1

X1 = -0.2      ==> 1(-0.2)^2 + 5(-0.2) + 1 = 0.000000
X2 = -4.8      ==> 1(-4.8)^2 + 5(-4.8) + 1 = 0.000000
```

Figura 4.1: Salida del Ejemplo 4.1

proporcionan, pero no se sabe con certeza cómo es que lo hacen.

Al igual que las funciones matemáticas, las funciones en C operan sobre un conjunto de datos, mismos que procesan o transforman, y finalmente regresan un valor; pero tome en cuenta que no todos los problemas se basan en, o modelan funciones matemáticas, y que por otro lado, no todas las funciones que se pudieran necesitar para resolver un problema se encuentran disponibles en las bibliotecas de funciones del lenguaje, por lo que en muchos casos, se necesitarán construir funciones de elaboración propia con servicios y funcionalidades específicas, y para ello, se necesita saber su estructura y cómo es que se definen.

Una posible salida para el programa del Ejemplo 4.1 se muestra en la Figura 4.1.

Tome en consideración que, dependiendo del compilador o de la distribución de GNU/Linux que utilice (si es el caso), es probable que el programa del Ejemplo 4.1 requiera del argumento “-lm” para su compilación. Para más detalles al respecto, consulte la sección B.2.3 del Apéndice B.

4.2. Conceptos y estructura

Las funciones son un *mecanismo de abstracción*, y en el contexto de la programación estructurada, constituyen **módulos** o **segmentos de código**, que le permiten al programador *encapsular* determinada funcionalidad o característica en un *bloque de código*, con la finalidad de que éste pueda ser *reutilizado* y *administrado* de una mejor forma.

Por otro lado, a la *cualidad* de escribir programas gobernados por *módulos*, es a lo que se le conoce como programación modular.

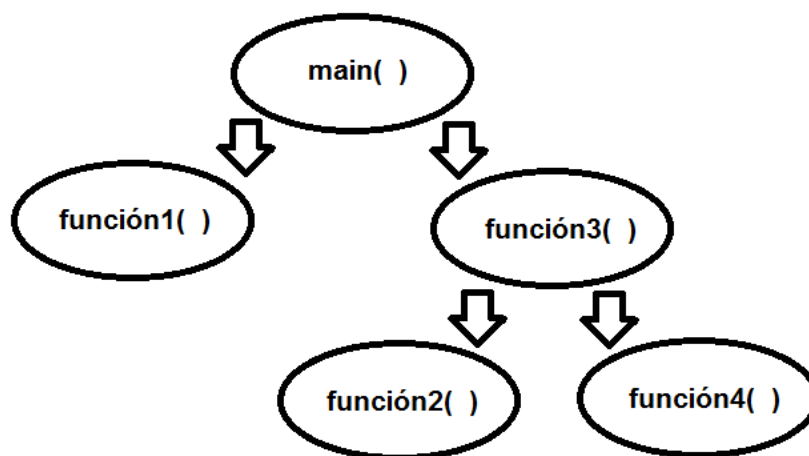


Figura 4.2: Programación modular

La **programación modular** es una *técnica* de programación que consiste en *identificar* partes *funcionales* y *reutilizables* de un programa (abstracción), y *encapsularlas* en entidades que siguen una estructura propia. A dichas entidades se les denomina en el lenguaje de programación C como **funciones**.

En C todos los módulos son funciones, a diferencia de otros lenguajes de programación, en los que los módulos pueden ser funciones o procedimientos.

Un programa modular en C opera de la siguiente manera: la función **main** es la función principal (jefe), y es la encargada de *delegar* responsabilidades y/o solicitar servicios a otras funciones (sus subordinados), las cuales a su vez se pueden valer de otras funciones (colegas) para llevar a cabo su tarea. Lo anterior se ilustra en la Figura 4.2.

Por último, pero no por ello menos importante, se debe recordar que cuando se trabaja con funciones, se requieren de tres aspectos muy importantes a considerar:

1. **Prototipo de función:** es la firma de la función, y le indica al compilador el tipo de valor de retorno de la función, el nombre de la función, y el número y tipo de cada uno de los *parámetros* con los que trabajará.
2. **Llamado (invocación) de función:** es el uso de la función, el llamado de la función se realiza a través de su nombre, y de la lista de *argumentos* que requiere la función para trabajar.

3. **Definición de función:** es en dónde se define el código que describe el funcionamiento de la función.

Los tres aspectos anteriores son igualmente importantes. El primero de ellos se ha usado de manera implícita al incluir las bibliotecas de funciones del lenguaje, el segundo al utilizar (llamar) a las funciones de dichas bibliotecas.

Por otro lado, la definición de funciones hasta ahora no se ha utilizado, debido a que involucra la definición de las funciones escritas por el programador; resta entonces el establecer cómo definir nuestras propias funciones.

4.2.1. Estructura para las funciones en C

La estructura general de la definición de una función en el lenguaje de programación C, es la siguiente:

```
tipo_de_dato_de_retorno nombre_de_función(lista_de_parámetros){
    tipo_de_dato variables_locales;
        sentencial1;
        .
        .
        .
        sentenciaN;
    return variable;
}
```

donde:

- **tipo_de_dato_de_retorno:** es el tipo de dato que la función regresará, y debe ser congruente con el de *variable*.
- **nombre_de_función:** es un identificador válido en C para nombrar a la función. A partir de él se harán los llamados respectivos.
- **lista_de_parámetros:** es una lista de variables (locales) separada por comas “,” , cada una con su propio identificador y tipo de dato.
- **tipo_de_dato:** es alguno de los tipos de datos del lenguaje de programación C.

- **variables locales:** es una lista de variables locales a la función, mismas que serán utilizadas como parte del grupo de *sentencia(s)*.
- **sentencias:** son el grupo de sentencias que representan, definen, describen y detallan la funcionalidad o utilidad de la función.
- **return:** es una palabra reservada, y se utiliza para regresar al invocador de la función el valor determinado por *variable*.

Al trabajar con funciones de biblioteca, la directiva **#include** es sumamente importante ya que incluye, entre otras cosas, los prototipos de función de las funciones de biblioteca que se utilizan en el programa que la contiene. Por otro lado, el llamado de función ocurre cuando se escribe el nombre de la función que se va a utilizar, pero el código fuente (la definición de la función) no es accesible, debido a que se encuentra en formato binario, listo para ser asociado y vinculado al programa que lo necesite, durante el proceso de compilación y vinculación.

4.2.2. Ejemplos de definición de funciones

Mayor y menor

Considere el Ejemplo 4.2. Las líneas 7 y 8 constituyen los prototipos de funciones y constituyen la firma de la función, en cuanto a que le indican al compilador que se definirán dos funciones: *mayor* y *menor*, mismas que recibirán tres parámetros enteros (**int**) y regresarán un valor entero.

Tanto los tipos de datos como el número de parámetros, así como el identificador de la función, están en relación directa de las necesidades del problema específico a resolver. Para el caso del Ejemplo 4.2, se definirán dos funciones que determinan (regresan), el mayor y menor respectivamente, de tres números enteros.

Note que en las líneas 18 y 19 se realiza el llamado de las funciones *mayor* y *menor* respectivamente, y que el valor regresado por cada una de ellas se utiliza en el contexto del especificador de formato “%d” de la función **printf** correspondiente, el cual está asociado al tipo de dato de retorno de la función (**int**).

Note también en las líneas 18 y 19, que a las funciones *mayor* y *menor* se les están pasando los datos *a*, *b* y *c*, los cuales son variables de la función **main** (línea 11), que a su vez contienen valores específicos (línea 15), y que

son transferidos o enviados a las funciones *mayor* y *menor* respectivamente como sus **argumentos**.

```
1  /* Programa que determina, utilizando funciones, el mayor y menor
2     de tres numeros enteros distintos.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int mayor( int , int , int );
8  int menor( int , int , int );
9
10 int main(){
11     int a, b, c;
12
13     do{
14         printf("Proporcione tres numeros enteros distintos: ");
15         scanf("%d%d%d", &a, &b, &c);
16     }while((a == b) || (a == c) || (b == c));
17
18     printf("El mayor es: %d\n", mayor( a, b, c ));
19     printf("El menor es: %d\n", menor( a, b, c ));
20
21     return 0;
22 }
23
24 int mayor(int x, int y, int z){
25     int max = x;
26
27     if(y > max)
28         max = y;
29
30     if (z > max)
31         max = z;
32
33     return max;
34 }
35
36 int menor(int x, int y, int z){
37     int min = x;
38
39     if (y < min)
40         min = y;
41
42     if (z < min)
43         min = z;
44
45     return min;
46 }
```

Ejemplo 4.2: Uso de funciones para determinar el máximo y el mínimo de tres números enteros distintos

Por otro lado, las líneas 24-46 corresponden a la definición de las funciones *mayor* y *menor*. Las líneas 24-34 definen el grupo de sentencias que componen

a la función *mayor*¹. Observe la estructura que sigue ésta función, compárela con la estructura general de una función en C, y asegúrese de comprender la equivalencia.

La línea 24 establece (define) que la función identificada como *mayor*, el cual es el identificador o nombre de la función, regresa un valor de tipo entero (**int**), y recibe tres parámetros de tipo entero (**int**) identificados por *x*, *y* y *z*².

Ahora bien, la línea 25 define una variable local a la función *mayor* identificada como *max*. En C es posible definir variables al inicio de cada bloque ({ ... }), y los bloques de función no son la excepción. Se pueden definir tantas variables locales como se necesiten y del tipo que se necesiten, y se siguen las mismas reglas de declaración de variables que para la función **main**.

Las líneas 27-31 son sentencias de comparación que, a través de estructuras de selección, sirven para determinar el mayor de tres números distintos. La idea subyacente es la siguiente:

1. Se selecciona a una de las variables como la mayor, supongamos que es *x*, y por lo tanto la asignamos a *max* (línea 25).
2. Se compara la segunda variable *y* con el valor de la variable supuestamente mayor (*max*), si $y > max$ (línea 27), entonces *max* no es la mayor, y se almacena en *max* el valor mayor hasta ese momento (línea 28). En caso contrario, *max* tiene el valor mayor entre *x* y *y* y no hay necesidad de intercambiar.
3. Se compara la tercera variable *z* con el valor de la variable hasta el momento mayor (*max*), si $z > max$ (línea 30), entonces *max* no es la mayor, y se almacena en *max* el valor mayor (línea 31). En caso contrario, no hay necesidad de intercambiar, *max* tiene el valor mayor de *x*, *y* y *z*.

La línea 33 contiene la palabra reservada **return** seguida de la variable *max*, y al igual que en **main**, regresa el valor contenido en *max* a quien llamó a la función *mayor*, que para el caso del Ejemplo 4.2, es en la línea 18.

Las líneas 36-46 para la función *menor*, se describen de manera análoga a las de la función *mayor* de las líneas 24-34.

Una posible salida para el Ejemplo 4.2 se muestra en la Figura 4.3.

¹Note las llaves que definen el inicio y fin del bloque de función.

²*x*, *y* y *z* consituyen la **lista de parámetros** de la función.

```
Proporcione tres numeros enteros distintos: 1 1 2
Proporcione tres numeros enteros distintos: 2 1 2
Proporcione tres numeros enteros distintos: 3 2 1
El mayor es: 3
El menor es: 1
```

Figura 4.3: Salida del Ejemplo 4.2

Finalmente, note que es posible definir más de una función en un programa, de hecho no hay un límite en ese sentido, por lo que se pueden definir tantas funciones como sean necesarias, y todas siguen las mismas reglas descritas hasta aquí.

Calculadora básica

El Ejemplo 4.3 implementa una calculadora básica con las cuatro operaciones aritméticas. Las líneas 6-10 muestran los prototipos de las funciones que se definirán en las líneas 37-58. Dichos prototipos deberían ser claros en lo que expresan, con excepción quizá del prototipo de la línea 6.

El prototipo de función de la línea 6, le indica al compilador que se definirá una función que no regresa nada (**void**). En el lenguaje de programación C, las funciones que sólo realizan alguna labor (como la presentación de un mensaje por ejemplo), pero que no necesitan regresar algún valor, usan éste tipo de dato. El tipo **void** es un tipo de dato muy importante, y su utilidad se retomará en el Capítulo 7, por ahora, basta con saber que ésta es una de sus utilidades.

Ahora bien, el bloque (cuerpo) de la función principal **main** se encarga de leer los datos (línea 19), de verificar que sean los apropiados o que cumplan con ciertas características (ciclo **do-while**), y de tomar la decisión de qué servicio (función) solicitar (llamar), en respuesta (estructura **switch**) al operador *op* leído (línea 19), para finalmente presentar el resultado (línea 32).

```
1  /* Programa que simula una calculadora y muestra el uso de funciones.
2     @autor Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  void instrucciones(void);
7  float suma(float , float);
8  float resta(float , float);
9  float multiplica(float , float);
```

```

10 float divide(float, float);
11
12 int main(){
13     float a, b, res;
14     char op;
15
16     do{
17         instrucciones();
18         printf("Expresion: ");
19         scanf("%f %c %f", &a, &op, &b);
20     }while(b == 0 || op != '+' && op != '-' && op != '*' && op != '/');
21
22     switch(op){
23         case '+': res = suma(a, b);
24             break;
25         case '-': res = resta(a, b);
26             break;
27         case '*': res = multiplica(a, b);
28             break;
29         case '/': res = divide(a, b);
30             break;
31     }
32     printf("%.2f %c %.2f = %.2f\n", a, op, b, res);
33
34     return 0;
35 }
36
37 void instrucciones(void){
38     printf("\n*****\n");
39     printf("Escriba su expresion usando el siguiente formato: ");
40     printf("A op B\nDonde A y B son operandos y op puede ser +, -, *, /\n");
41     printf("*****\n");
42 }
43
44 float suma(float a, float b){
45     return a + b;
46 }
47
48 float resta(float a, float b){
49     return a - b;
50 }
51
52 float multiplica(float a, float b){
53     return a * b;
54 }
55
56 float divide(float a, float b){
57     return a / b;
58 }

```

Ejemplo 4.3: Implementación de una calculadora básica utilizando funciones

La definición de funciones está de las líneas 37-58, y se describen de la siguiente manera:

- La función *instrucciones* (líneas 37-42), se utiliza para mostrar un men-


```

*****
Escriba su expresion usando el siguiente formato: A op B
Donde A y B son operandos y op puede ser +, -, *, /
*****
Expresion: 1 / 0

*****
Escriba su expresion usando el siguiente formato: A op B
Donde A y B son operandos y op puede ser +, -, *, /
*****
Expresion: 1 / 3
1.00 / 3.00 = 0.33

```

Figura 4.4: Salida del Ejemplo 4.3

saje en la salida estándar que contiene instrucciones básicas del uso del programa.

- La función *suma* (líneas 44-46) es bastante simple, ya que regresa a quien la llame, el valor de la expresión de suma ($\mathbf{a} + \mathbf{b}$) de los parámetros a y b . Observe que aunque es posible, no es necesario guardar el resultado de dicha expresión en una variable local y después regresarla, de hecho, es muy común en C escribir éste tipo de expresiones como valor de retorno.
- Las funciones *resta* (líneas 48-50), *multiplica* (líneas 52-54), y *divide* (líneas 56-58), se describen de manera análoga a la función *suma*.

Note que la función *divide* no hace ningún tipo de verificación del denominador para realizar la operación de división, debido a que asume que ya se ha realizado dicha verificación en alguna otra parte del programa (**main** línea 20), y que el denominador es distinto de cero. El programa del Ejemplo 4.3 es sólo un posible diseño a dicha situación, el cual no es el único ni necesariamente el mejor, sin embargo funciona y refuerza el concepto de programación modular ilustrado en la Figura 4.2.

Finalmente, tome en cuenta que para el programa del Ejemplo 4.3, ninguna de sus funciones, excepto **main**, hace uso de otra función para completar su tarea debido a su simplicidad, sin embargo, es posible hacerlo y se mostrará en ejemplos posteriores. Una posible salida para el Ejemplo 4.3 se muestra en la Figura 4.4.

4.3. Ámbitos de variables

El Ejemplo 4.4 ilustra el concepto de **ámbito de variables**, el cual se refiere al nivel de *visibilidad* o *alcance* para las variables. Para visualizar mejor el concepto, imagine el ámbito de variables como un conjunto de capas que se sobreponen, todas las capas existen, pero en un momento dado, sólo es accesible una de ellas.

Las líneas 7-9 declaran los prototipos de tres funciones: *función1*, *función2* y *función3*, ninguna de las tres regresa ningún valor ni recibe nada como parámetro.

La línea 11 define una variable global *x* de tipo entero (**int**) que es inicializada con el valor de uno; de hecho, observe que todas las variables se llaman igual y que son del mismo tipo.

Una **variable global** tiene un alcance de archivo, lo cual quiere decir, que dicha variable se reconoce desde el punto de su definición, hasta el final del archivo que la contiene.

Las variables globales no deberían ser utilizadas, debido a que introducen desorden en la estructura de los programas y son susceptibles de ser modificadas de manera accidental o intencional en cualquier parte del archivo, generando con ello resultados o efectos colaterales que invariablemente sumergirán al programador en “entretenidas” sesiones de depuración.

Además de lo anterior, las variables globales se consideran en general una mala práctica de programación. La intención del Ejemplo 4.4 es ilustrar los ámbitos incluso para las variables globales, pero no pretende de ninguna manera promover su uso.

Retomando la explicación del ejemplo, la línea 14 declara la **variable local** *x* a **main** con valor inicial de diez. El ámbito de ésta variable es el bloque asociado con la función **main**, por lo que, ¿qué valor imprimirá la función **printf** de la línea 16?

La pregunta es bastante razonable, debido a que hasta la línea 16 se tienen definidas dos variables con el mismo nombre y del mismo tipo, por lo tanto, ¿cual se imprimirá?. Si bien es cierto que el alcance de una variable global es de todo el archivo, también lo es que los ámbitos se superponen entre sí, pero no se sobre escriben, es decir, que para la línea 16 el ámbito activo es el de la función **main**, por lo que el valor que se imprimirá será diez.

```

1  /* Programa para ilustrar los ambitos de definicion y existencia
2  de varibales automaticas y estaticas (version compacta).
3  @autor Ricardo Ruiz Rodriguez

```

```

4  */
5  #include <stdio.h>
6
7  void funcion1(void);
8  void funcion2(void);
9  void funcion3(void);
10
11 int x = 1;  /* variable global: mala practica */
12
13 int main(){
14     int x = 10;  /* variable local de main */
15
16     printf("Valor de x = %d (main)\n", x);
17     {
18         /* nuevo ambito interno (inicio) */
19         int x = 50;
20         printf("Valor de x = %d (main-> ambito interno)\n", x);
21     }
22     /* nuevo ambito interno (fin) */
23     printf("Valor de x = %d (main)\n", x);
24
25     funcion1(); funcion2(); funcion3();
26     funcion1(); funcion2(); funcion3();
27
28     printf("\nValor de x = %d (main)\n", x);
29
30     return 0;
31 }
32
33 void funcion1(void){
34     int x = 100;  /* la variable se crea e inicializa en cada llamado */
35
36     printf("\nValor de x = %d al entrar (funcion1)\n", x++);
37     printf("Valor de x = %d al salir (funcion1)\n", x);
38 }
39
40 void funcion2(void){
41     static int x = 1000;  /* la variable se crea e inicializa una sola vez */
42
43     printf("\nValor de x = %d al entrar (funcion2)\n", x++);
44     printf("Valor de x = %d al salir (funcion2)\n", x);
45 }
46
47 void funcion3(void){
48     printf("\nValor de x = %d al entrar (funcion3)\n", x++);
49     printf("Valor de x = %d al salir (funcion3)\n", x);
50 }

```

Ejemplo 4.4: Ámbitos de variables, variables automáticas y estáticas

Por otro lado, las líneas 17-20 definen un nuevo **ámbito interno** delimitado por su propio bloque (`{ ... }`) dentro de la función **main**. Observe que dicho bloque no pertenece a ninguna estructura de control, ya que las llaves no son propias de ninguna estructura de control (de hecho son opcionales), sino que delimitan bloques de código, y cada bloque de código tiene su propio ámbito. Éste ámbito interno, declara una nueva *x* con valor de 50, por lo que

```
Valor de x = 10 (main)
Valor de x = 50 (main-> ambito interno)
Valor de x = 10 (main)

Valor de x = 100 al entrar (funcion1)
Valor de x = 101 al salir (funcion1)

Valor de x = 1000 al entrar (funcion2)
Valor de x = 1001 al salir (funcion2)

Valor de x = 1 al entrar (funcion3)
Valor de x = 2 al salir (funcion3)

Valor de x = 100 al entrar (funcion1)
Valor de x = 101 al salir (funcion1)

Valor de x = 1001 al entrar (funcion2)
Valor de x = 1002 al salir (funcion2)

Valor de x = 2 al entrar (funcion3)
Valor de x = 3 al salir (funcion3)

Valor de x = 10 (main)
```

Figura 4.5: Salida del Ejemplo 4.4

el valor a imprimir en la línea 19 es 50. ¿Qué valor se imprime en la línea 21?.

La línea 23 hace un sucesivo llamado a las funciones: *funcion1*, *funcion2* y *funcion3* respectivamente, mientras que la línea 24 repite dicha sucesión. Para entender la salida del programa (Figura 4.5), se debe analizar la definición de las funciones.

Las líneas 31-36 definen a la función *funcion1*, la cual declara una variable local *x* en la línea 32 con un valor inicial de 100. Ésta variable, como todas las que se declaran de esta forma, se denominan **variables automáticas**, debido a que se crean y se destruyen cada vez que la secuencia de ejecución entra y sale, respectivamente, de su respectivo ámbito de función. La función imprimirá en las líneas 34 y 35, los valores de 100 y 101 respectivamente, sin importar si la función se llama una vez, tres veces, 456 o 987 veces.

Por otro lado, las líneas 38-43 definen a la función *funcion2*, la cual declara

una variable local x **estática** (**static**) en la línea 39, con un valor inicial de 1000. El modificador **static** instruye al compilador, para que no destruya a la variable que es afectada por el modificador cuando la secuencia de ejecución abandone el ámbito en que se definió, conservando así su valor. Sin embargo, dicha variable sólo es accesible dentro de su ámbito, esto es, sólo la *funcion2* la puede “ver”. Cuando una variable es estática, su valor de inicialización sólo se lleva a cabo la primera vez que se hace uso de la variable, por lo que sus valores subsecuentes dependerán del último valor asignado o modificado; en éste sentido, los valores que se imprimirán en las líneas 41 y 42 en el primer llamado de la *funcion2* serán 1000 y 1001 respectivamente, pero no lo serán en los llamados subsecuentes.

Finalmente, las líneas 45-48 definen a la función *funcion3*, la cual no declara ninguna variable pero sí hace uso de una variable en las líneas 46 y 47. Quizá se pregunte ¿cómo es esto posible?, pues es posible debido al uso de la variable global (línea 11), por lo que los valores a imprimir son uno y dos respectivamente.

Siguiendo la descripción que se acaba de hacer de las funciones, determine los valores que se imprimirán en la salida estándar para las líneas 24 y 26.

4.4. Bibliotecas personalizadas de funciones

El lenguaje de programación C incorpora muchas y muy útiles, además de variadas, funciones incluidas en sus diferentes bibliotecas, pero además, a través de un mecanismo relativamente sencillo, es posible “crear” bibliotecas personalizadas de funciones, esto es, archivos que contengan funciones definidas por el programador y que puedan ser utilizadas en cualquier programa que las incluya, de manera similar a como se hace con las bibliotecas del lenguaje.

Estrictamente hablando, con el procedimiento que se describirá no se estará creando ninguna biblioteca, sino que más bien haciendo uso del mecanismo de abstracción proporcionado por la directiva de inclusión de archivos **#include**, para dar al programador la sensación de incluir sus propias bibliotecas de funciones.

En esta sección se mostrará brevemente el procedimiento para crear bibliotecas personalizadas de funciones. La descripción se basará en el Ejemplo 4.2 de la sección 4.2.2, por lo que es recomendable darles una revisión antes de continuar.

El primer paso para crear una biblioteca personalizada de funciones, es tener las funciones definidas por el programador en un archivo nuevo e independiente. Dicho archivo debe contener al menos una función, mientras que el límite dependerá del espacio designado para el archivo en el disco que lo almacene, es decir, potencialmente “ilimitado”.

```
1  /* Funciones que determinan el mayor y menor de tres
2     numeros enteros distintos que forman parte de la
3     primera biblioteca personalizada de funciones.
4     @autor Ricardo Ruiz Rodriguez
5  */
6  int mayor(int x, int y, int z){
7      int max = x;
8
9      if(y > max)
10         max = y;
11
12     if (z > max)
13         max = z;
14
15     return max;
16 }
17
18 int menor(int x, int y, int z){
19     int min = x;
20
21     if (y < min)
22         min = y;
23
24     if (z < min)
25         min = z;
26
27     return min;
28 }
```

Ejemplo 4.5: Funciones contenidas en el archivo “miBiblioteca.h”

Resulta sumamente conveniente crear bibliotecas personalizadas de funciones con funciones que tengan algún tipo de relación entre ellas, por lo que en realidad, la limitante en el número de funciones que pueda contener una biblioteca, debería estar supeditada a la abstracción de los servicios que se espera proporcione dicha biblioteca.

Las funciones *mayor* y *menor* del Ejemplo 4.2, se han colocado en un nuevo archivo de nombre *miBiblioteca.h*, tal y como se muestra en el Ejemplo 4.5. El nombre de archivo que contendrá la biblioteca personalizada de funciones no deberá contener espacios preferentemente.

Note que en esencia el Ejemplo 4.2 y el Ejemplo 4.5 son esencialmente iguales, la diferencia radica en la localización de las funciones que, para el caso de bibliotecas personalizadas, se encuentran en un archivo independiente

con extensión “.h”.

```
1  /* Programa que determina, utilizando funciones, el mayor y menor
2     de tres numeros enteros distintos utilizando una biblioteca
3     personalizada de funciones.
4     @autor Ricardo Ruiz Rodriguez
5  */
6  #include <stdio.h>
7  #include "miBiblioteca.h"
8
9  int main(){
10     int a, b, c;
11
12     do{
13         printf("Proporcione tres numeros enteros distintos: ");
14         scanf("%d%d%d", &a, &b, &c);
15     }while((a == b) || (a == c) || (b == c));
16
17     printf("El mayor es: %d\n", mayor( a, b, c ));
18     printf("El menor es: %d\n", menor( a, b, c ));
19
20     return 0;
21 }
```

Ejemplo 4.6: Inclusión y uso de la biblioteca personalizada de funciones

La línea 7 del Ejemplo 4.6 muestra la inclusión de la biblioteca personalizada de funciones del Ejemplo 4.5. Observe que en la directiva **#include** se han cambiado los corchetes angulares (`< >`) por comillas (“ ”), lo cual, además de diferenciar una biblioteca personalizada de funciones de una biblioteca del lenguaje, le indica al compilador que busque el archivo indicado entre comillas, en la misma localidad que el archivo que incluye a dicha biblioteca.

En resumen, tanto el archivo fuente que usará a la biblioteca personalizada de funciones, como el archivo que contiene las funciones de biblioteca personalizada, deberán estar en el mismo directorio. Cabe mencionar que es posible especificar una ruta para la localización de un archivo de biblioteca personalizada en particular, pero en general esto no es recomendable debido a que se sacrifica la portabilidad, ya que la especificación de rutas de archivos y directorios cambia de un sistema operativo a otro.

Así como no hay un límite para el número de funciones que puede contener un archivo de biblioteca personalizada, se debe aclarar que tampoco hay un límite para el número de archivos de bibliotecas personalizadas que se pueden incluir dentro de un programa. Para el caso del Ejemplo 4.6, se han incluido una biblioteca del lenguaje (línea 6) y una biblioteca personalizada (línea 7), pero podrían ser un número arbitrario de cualquiera de ellas; la

única recomendación en este sentido sería, incluir primero las bibliotecas del lenguaje, y después las bibliotecas personalizadas con las funciones definidas por el programador.

Finalmente, note que si se proporcionan los mismos datos que los mostrados en la Figura 4.3, es posible generar la misma salida para el Ejemplo 4.6. Verifique que en verdad así sea.

4.5. Ejercicios

1. Complete el Ejemplo 4.2 con la función *medio*, la cual devuelve el número de en medio en base a su valor ordinal.
2. Escriba un programa que, dado un número de hasta cinco cifras, exprese el valor de dicho número en palabras. Así por ejemplo, para el número 99 999, el programa dará como salida: “99 999: noventa y nueve mil novecientos noventa y nueve”.

El castellano es un idioma complejo, así que si la salida de su programa es “99 999: noventa y nueve mil nueve cientos noventa y nueve”, o genera expresiones por el estilo, considérela correcta.

3. Escriba un programa que, a través de una función, calcule el cuadrado de un número entero.
4. Utilice la función desarrollada en el ejercicio anterior para resolver el problema del Ejemplo 4.1. Observe que en la línea 22, el discriminante lleva el cálculo de \mathbf{b}^2 , utilice su función para calcular éste valor.
5. Escriba un programa que, a través de una función, calcule el producto de dos números enteros positivos **a** y **b**, por medio de sumas sucesivas, esto es:

$$a * b = a + a + \dots + a$$

donde **a** se suma tantas veces como lo indique **b**.

6. Escriba un programa que, a través de una función, calcule el cociente entero de dos números enteros positivos **a** y **b** por medio de restas sucesivas, esto es:

$$\begin{array}{llll}
 a/b \Rightarrow & a - b = c1 & \text{si } c1 > 0 & \text{entonces} \\
 & c1 - b = c2 & \text{si } c2 > 0 & \text{entonces} \\
 & c2 - b = c3 & \text{si } c3 > 0 & \text{entonces} \\
 & & \vdots &
 \end{array}$$

donde el cociente estará dado por el *número de veces* que se haya podido repetir el procedimiento descrito.

7. Escriba un programa que, a través de una función, calcule la potencia de dos números enteros positivos **a** y **b**, por medio de productos sucesivos, esto es:

$$a^b = a * a * \dots * a$$

donde **a** se multiplica tantas veces como lo indique **b**.

8. Repita el ejercicio anterior, pero en lugar de usar el operador de multiplicación (*), utilice la función que desarrolló en el Ejercicio 5.
9. Si para el programa del Ejemplo 4.4 se realizara una nueva ronda de llamados consecutivos a cada una de las tres funciones involucradas, ¿cuáles serían los valores que se visualizarían en la salida estándar?
10. Para el Ejemplo 4.6 elimine la línea 7, guarde el archivo, y compile. ¿Qué sucede?, ¿a qué se debe y por qué?
11. Incorpore a la biblioteca personalizada de funciones del Ejemplo 4.5, la función *medio* realizada en el Ejercicio 1 y pruébela.
12. Pruebe creando otras bibliotecas personalizadas de funciones con los ejemplos y ejercicios realizados hasta ahora. No importa que las funciones de dichas bibliotecas no tengan mucha relación, lo que importa es practicar la creación de bibliotecas personalizadas de funciones.

Capítulo 5

Recursividad

Dos amibas vivían muy contentas en el estómago de Fausto, relativamente cerca del píloro. Pasaban la vida cómodamente, comían muy bien y nunca trabajaban: eran lo que se llama unas parásitas. Se querían mucho, eran buenas amigas, pero de vez en cuando entraban en fuertes discusiones porque tenían temperamentos muy distintos, y cada una aprovechaba su ocio de manera diferente: una era muy pensativa y siempre se preguntaba qué sucedería al día siguiente; la otra, en cambio, era muy glotona, se pasaba el día comiendo y prefería vivir con gusto cada instante de su vida sin pensar en el mañana.

Una vez, a la hora de la comida, la amiba pensativa le platicó a su compañera lo que había estado pensando esa mañana: -A lo mejor -le dijo- el mundo que nos rodea, los ríos, las montañas, los valles, los grandísimos canales, el cielo, no son tan grandes como los vemos; a lo mejor este mundo es muy pequeñito y todos los que vivimos aquí no somos más que unos bichitos diminutos que estamos adentro de otro bicho más grande, y ese otro bicho está en otro más grande y...

La amiba glotona, que estaba comiéndose una lenteja gigantesca, le dijo que eso no era posible y que consideraba una manera de perder el tiempo pensar en esas tonterías. Cuando Fausto terminó el plato de lentejas que estaba comiendo, se tomó una medicina y las dos amibas desaparecieron.

Fausto y Enrique, su gordísimo invitado, se quedaron platicando de sobremesa. Fausto decía que a lo mejor el hombre no

era más que un bichito diminuto que vivía adentro de otro bicho más grande.... Pero Enrique, que no había acabado de comerse su inmenso plato de lentejas, lo interrumpió: -Eso no es posible —le dijo—, y creo que es una manera de perder el tiempo pensar en esas tonterías...

Gonzalo Celorio

La historia de *Dos amibas amigas* está relacionada con la recursividad. La recursividad no es exclusiva de la computación, de hecho su concepción *formal* viene de las matemáticas y las relaciones de recurrencia. Además, existen diferentes organismos y sistemas en la naturaleza que son de naturaleza recursiva, como algunos tipos de corales, plantas, etc., con una impresionante geometría fractal.

5.1. Definición y conceptos

Existen al menos dos películas que, además de ser de mi agrado, ilustran excepcionalmente bien el concepto de recursividad:

1. *Inception*: aquí la recursividad se presenta debido a que dentro de un sueño, se puede estar soñando, incluso, quizá haya sabido de personas que sueñan que sueñan, eso es recursividad.
2. *Being John Malcovich*: uno de los personajes encuentra una especie de túnel que permite estar por unos minutos dentro de la mente de John Malcovich, y así poder ver lo que él ve, sentir lo que él siente, etc. John Malcovich se da cuenta de ello, y se introduce en el túnel que lleva a su propia mente. La recursividad radica en que la mente de John Malcovich es invadida por él mismo.

Recursividad es la forma de especificar un proceso en función de su propia definición, y dicho proceso puede ser finito o infinito.

Note que la definición de recursividad es recursiva.

Para tener la especificación de un proceso *finito*, la especificación debe hacerse en términos de una *simplificación* del problema original, de tal forma que eventualmente se alcance un **caso base** o **criterio de paro**.

Por otro lado, un proceso recursivo infinito se ilustra en la Figura 5.1. Observe que del trazo inicial (nivel 1) se puede generar el nivel 2, repitiendo

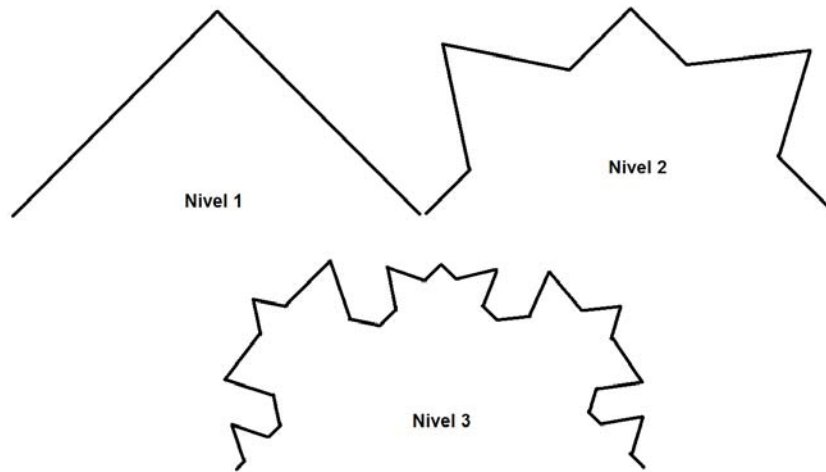


Figura 5.1: Recursividad infinita (aquí sólo hasta el nivel 3)

la misma figura del nivel 1, en cada uno de los segmentos de recta del nivel 1. Si este proceso se vuelve a aplicar a los nuevos segmentos generados en el nivel 2, se genera la figura del nivel 3; y en principio, al margen de las limitaciones físicas, es posible repetir dicho proceso de manera infinita. Para el caso del ejemplo mostrado en la figura, sólo se ha llegado hasta el nivel 3, generando de manera recursiva, una figura geométrica interesante.

La especificación de procesos finitos expresados en un programa en C son el tema del presente capítulo.

5.1.1. Aspectos inherentes: *overhead*

Muchas funciones matemáticas, como el factorial y la serie de Fibonacci por ejemplo, resultan útiles para ilustrar el concepto de recursividad, pero esto no quiere decir que la solución recursiva sea la más apropiada, de hecho es en general más ineficiente, debido a que cada llamado de función lleva implícito un costo de gestión (*overhead*).

¿Se ha puesto a pensar lo que implica por ejemplo, cambiar el flujo de ejecución de un programa para procesar el código de una función?, ¿qué pasa cuando se llama a una función?, ¿cómo sabe la computadora a dónde tiene que regresar después de procesar el código de una función?, ¿cómo es que no se revuelven los datos de la función que llama con la función llamada si todos los datos se procesan en el mismo microprocesador?. En éste sentido,

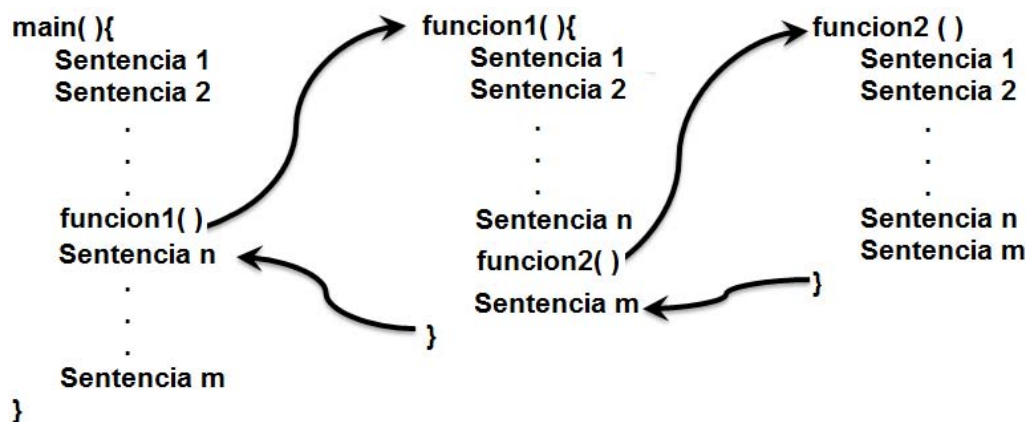


Figura 5.2: Mecanismo de llamado y retorno de función

considere la Figura 5.2.

Cuando ocurre el *llamado* de una función, la *secuencia de ejecución* (*flujo de control*) cambia, y se da un salto hacia el código que representa dicha función¹, pero éste cambio no es tan simple como parece:

- El ámbito de una función es un nuevo contexto en todos los sentidos:
 - Hay nuevas variables.
 - Hay nuevas instrucciones y sentencias a procesar.

Todas ellas se ejecutan en el mismo microprocesador y utilizan el mismo conjunto de registros, por lo que antes de ejecutar la primera línea de código de una función, se debe respaldar el estado de todos los registros, y la dirección de retorno², y otros elementos que se utilizarán para procesar el nuevo contexto (la función).

- Cuando la función termina, antes de regresar el control a donde estaba antes del llamado (ámbito del invocador), se deben restaurar todos los datos respaldados en los registros, para que no exista ninguna afectación en la lógica de ejecución y de procesamiento de datos del ámbito

¹Esto se ilustra con las flechas que van de izquierda a derecha en la Figura 5.2.

²La dirección de la siguiente instrucción en la secuencia de ejecución que se estaba procesando antes del llamado a la función.

del invocador. La ejecución debe continuar como si nunca se hubiera cambiado de flujo de ejecución³.

Aunque la descripción anterior es una versión muy simplista y resumida de lo que pasa con cada llamado de función, lo importante es tener en mente que atrás del llamado de una función suceden muchas muchas cosas, y a eso es a lo que se le llama, desde el punto de vista computacional **costo de gestión** de funciones (*overhead*).

5.2. Soluciones iterativas vs. recursivas

Existen dos funciones clásicas y claves que se resuelven con recursividad:

1. La función que calcula el factorial de un número entero.
2. La función que calcula los términos de la serie de Fibonacci.

Específicamente para éstas dos funciones, el enfoque de la recursividad no es la manera más eficiente de resolverlas, pero ayudan a ilustrar el concepto de recursividad.

Por otro lado, dichas funciones son tan clásicas que, para no incurrir en algún tipo de maldición del estilo del programa “Hola mundo!”, se revisarán necesaria, y rápidamente, primero en su versión iterativa (ciclos) y posteriormente en su versión recursiva.

5.2.1. Factorial

Versión iterativa

El factorial de n denotado por $n!$ ($n \geq 0$), se obtiene en base a lo expresado en la Ecuación 5.1.

$$n! = n * (n - 1) * (n - 2) * \dots 1 \quad (5.1)$$

de tal forma que, si $n = 5$, se tiene:

$$5! = 5 * 4 * 3 * 2 * 1 = 120 \quad (5.2)$$

³Lo cual se ilustra con las flechas que van de derecha a izquierda en la Figura 5.2.

De lo anterior puede observarse que la solución está dada por un ciclo. El Ejemplo 5.1 muestra la función que calcula de manera iterativa el factorial de un número. Note que la función asume que $n \geq 0$.

```

1  /* Programa para calcular el factorial de un numero de manera
2     iterativa (ciclos) con funcion.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  long factorial(int);
8
9  int main(){
10     int n;
11
12     do{
13         printf("n (>=0)? = ");
14         scanf("%d", &n);
15         if(n < 0)
16             printf("Error: el factorial esta definido para n >= 0\n\a");
17     }while(n < 0);
18
19     printf("%d! = %ld\n", n, factorial(n));
20
21     return 0;
22 }
23
24 long factorial(int n){
25     long fact = 1;
26     int i;
27
28     for(i = 2; i <= n; i++)
29         fact *= i;
30
31     return fact;
32 }

```

Ejemplo 5.1: Función para el factorial de un número (versión iterativa)

Todas las sentencias del Ejemplo 5.1 deberían ser entendidas. Asegúrese de que así sea antes de continuar.

Versión recursiva

Matemáticamente, la función factorial se define según lo expresado en la Ecuación 5.3, para $n \geq 0$.

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n - 1)! & \text{en otro caso} \end{cases} \quad (5.3)$$

La función del Ejemplo 5.2 es esencialmente una traducción al lenguaje C de la ecuación 5.3, donde a las líneas 25 y 26 se les conoce como *criterio*

de *paro* o *caso base* de la recursividad, mientras que la línea 28 contiene el *paso recursivo*.

```

1  /* Programa para calcular el factorial de un numero
2     de manera recursiva.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  long factorial(int);
8
9  int main(){
10     int n;
11
12     do{
13         printf("n (>=0)? = ");
14         scanf("%d", &n);
15         if(n < 0)
16             printf("Error: el factorial esta definido para n >= 0\n\a");
17     }while(n < 0);
18
19     printf(" %d! = %ld\n", n, factorial(n));
20
21     return 0;
22 }
23
24 long factorial(int n){
25     if(n == 0)
26         return 1;
27     else
28         return n * factorial(n - 1);
29 }

```

Ejemplo 5.2: Función para el factorial de un número (versión recursiva)

Es importante hacer notar que en cada llamado recursivo se tiene una simplificación del problema original, lo que asegura que eventualmente se llegue al caso base de la recursividad, siempre que $n \geq 0$.

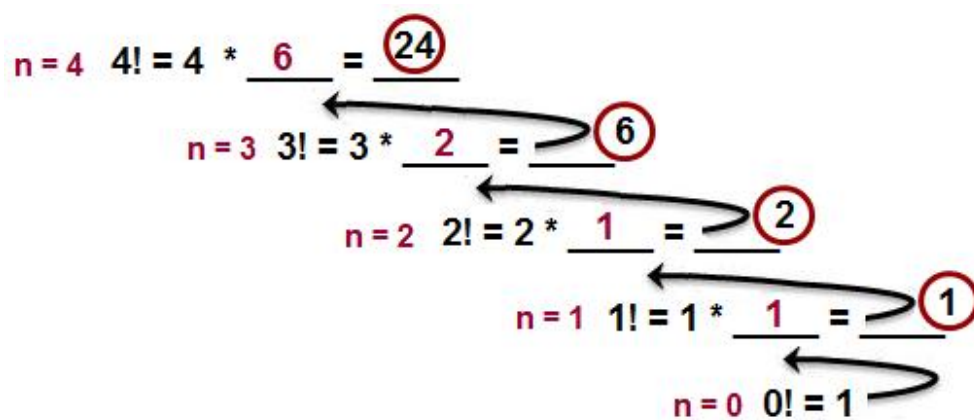
La Figura 5.3 muestra el funcionamiento de la función *factorial* (líneas 24-29) del Ejemplo 5.2 para $n = 4$.

En la Figura 5.3 (a) se muestran los llamados recursivos, note cómo en cada nivel de n existen valores pendientes de ser calculados, debido precisamente al funcionamiento de la recursividad. Cuando finalmente se alcanza el caso base ($n = 0$), es posible determinar los valores que fueron quedando pendientes⁴, y calcular los resultados de n para cada nivel, los cuales son mostrados encerrados en círculos en la Figura 5.3 (b). Analice y estudie la Figura 5.3 hasta comprenderla, ya que en buena medida, de ella depende la comprensión del Ejemplo 5.2.

⁴Las flechas de la Figura 5.3 (b) muestran éste procedimiento.

$$\begin{aligned}
 n=4 \quad 4! &= 4 * \underline{\hspace{1cm}} = \underline{\hspace{1cm}} \\
 n=3 \quad 3! &= 3 * \underline{\hspace{1cm}} = \underline{\hspace{1cm}} \\
 n=2 \quad 2! &= 2 * \underline{\hspace{1cm}} = \underline{\hspace{1cm}} \\
 n=1 \quad 1! &= 1 * \underline{\hspace{1cm}} = \underline{\hspace{1cm}} \\
 n=0 \quad 0! &= 1
 \end{aligned}$$

(a) Llamados recursivos



(b) Retornos recursivos

Figura 5.3: Llamados y retornos recursivos para el factorial de cuatro

```
n (>=0)? = -1  
Error: el factorial esta definido para n >= 0  
n (>=0)? = 4  
4! = 24
```

Figura 5.4: Salida de los Ejemplos 5.1 y 5.2 para $n = 4$

Es preciso mencionar que una vez que se tiene **la relación de recurrencia**⁵ que establece tanto el criterio de paro, como el (los) paso(s) recursivo(s), el proceso de traducción al lenguaje C es muy sencillo, por lo que la deducción de dicha relación de recurrencia es en realidad la parte difícil en el diseño de funciones recursivas. Para el caso de la función factorial, la relación de recurrencia está dada por la definición matemática de Ecuación 5.3.

Por último, note que se ha utilizado el tipo de dato **long** debido a que el factorial de un número crece de manera exponencial. La salida de los Ejemplos 5.1 y 5.2 para $n = 4$ se muestra en la Figura 5.4.

5.2.2. Fibonacci

Versión iterativa

Otra función matemática clásica y clave, es la función que genera la serie o sucesión de Fibonacci. La serie de Fibonacci, es la siguiente sucesión infinita de números:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Observe que, con excepción de los dos primeros términos, cada término de la serie se obtiene como la suma de los dos términos anteriores, como se muestra en la Tabla 5.1.

De los elementos de la serie mostrados en la Tabla 5.1, es posible deducir que la solución puede obtenerse también mediante un ciclo.

El Ejemplo 5.3 muestra la función que calcula de manera iterativa el número de Fibonacci para un término de la serie. Observe que la función asume que $n \geq 0$.

Asegúrese de entender el programa del Ejemplo 5.3 antes de continuar, particularmente la función *fibonacci*, todas las sentencias deberían ser claramente comprendidas. Note que, al igual que antes, se ha utilizado el tipo de dato **long**.

⁵Una relación de recurrencia es una ecuación que relaciona a los términos de una sucesión con alguno de sus predecesores.

fib(0) = 0	fib(6) = 8
fib(1) = 1	fib(7) = 13
fib(2) = 1	fib(8) = 21
fib(3) = 2	fib(9) = 34
fib(4) = 3	fib(10) = 55
fib(5) = 5	fib(11) = 89

Tabla 5.1: Primeros 12 términos de la serie de Fibonacci

```

1  /* Programa para calcular un numero de la serie Fibonacci
2  de manera iterativa (ciclos) con funcion.
3  @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  long fibonacci(int);
8
9  int main(){
10     int n;
11
12     do{
13         printf("n (>=0)? = ");
14         scanf("%d", &n);
15         if(n < 0)
16             printf("Error: fibonacci esta definido para n >= 0\n\a");
17     }while(n < 0);
18
19     printf("fibonacci(%d) = %ld\n", n, fibonacci(n));
20
21     return 0;
22 }
23
24 long fibonacci(int n){
25     long fib = 0, fibn_1 = 1, fibn_2 = 0;
26     int i;
27
28     if(n == 0 || n == 1)
29         fib = n;
30     else
31         for(i = 2; i <= n; i++){
32             fib = fibn_1 + fibn_2;
33             fibn_2 = fibn_1 ;
34             fibn_1 = fib;
35         }
36
37     return fib;
38 }

```

Ejemplo 5.3: Función de Fibonacci (versión iterativa)

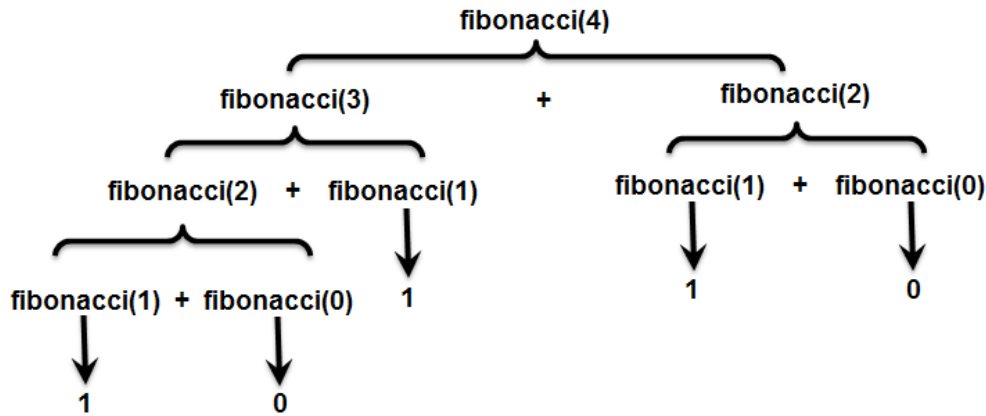


Figura 5.5: Árbol de recursividad para $n = 4$ de la serie de Fibonacci

Versión recursiva

Ahora bien, matemáticamente, la función de Fibonacci que genera los términos de la sucesión de Fibonacci, se define en base a lo expresado en la Ecuación 5.5 para $n \geq 0$.

$$fib(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ó } n = 1 \\ fib(n-1) + fib(n-2) & \text{si } n > 1 \end{cases} \quad (5.4)$$

Note que la función *fibonacci* del Ejemplo 5.4 es una traducción al lenguaje C de la Ecuación 5.4. Observe también que las líneas 25 y 26 implementan el criterio de paro o caso base de la recursividad, mientras que la línea 28 contiene los dos pasos recursivos involucrados en el cálculo de los términos de la serie.

Para $n \geq 2$, cada término de la serie se calcula como la suma de dos llamados recursivos, por lo que la función *fibonacci* del Ejemplo 5.4 genera un *árbol de recursividad* como el mostrado en la Figura 5.5 para $n = 4$.

Observe de la Figura 5.5 que el término *fibonacci(2)* se repite en la rama derecha y en la rama izquierda. Ésta repetición de cálculos, aunada al *overhead* implicado en cada llamado de función, hace que las implementaciones recursivas sean en general ineficientes respecto del tiempo.

Lo anterior debe reafirmar en la mente del lector que las implementaciones recursivas de las funciones *factorial* y *fibonacci* son ineficientes, y por consiguiente, un mal enfoque de solución, sin embargo, ambas funciones ilustran de manera extraordinaria el concepto de recursividad. Por otro lado,

```

n (>=0)? = -5
Error: fibonacci esta definido para n >= 0
n (>=0)? = 4
fibonacci(4) = 3

```

Figura 5.6: Salida de los Ejemplos 5.3 y 5.4 para $n = 4$

basta con ver las definiciones matemáticas y compararlas con sus correspondientes versiones escritas en lenguaje C, para darse cuenta de la elegancia, la correspondencia directa, y la simplicidad inherente a las implementaciones recursivas.

```

1  /* Programa para calcular un numero de la serie Fibonacci
2     de manera recursiva.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  long fibonacci(int);
8
9  int main(){
10     int n;
11
12     do{
13         printf("n (>=0)? = ");
14         scanf("%d", &n);
15         if(n < 0)
16             printf("Error: fibonacci esta definido para n >= 0\n\a");
17     }while(n < 0);
18
19     printf("fibonacci(%d) = %ld\n", n, fibonacci(n));
20
21     return 0;
22 }
23
24 long fibonacci(int n){
25     if(n == 0 || n == 1)
26         return n;
27     else
28         return fibonacci(n - 1) + fibonacci(n - 2);
29 }

```

Ejemplo 5.4: Función de Fibonacci (versión recursiva)

La salida de los Ejemplos 5.3 y 5.4 para $n = 4$ se muestra en la Figura 5.6.

5.2.3. Experimento empírico

El Ejemplo 5.5 muestra un esqueleto de programa para medir el tiempo en segundos de un proceso largo.

La línea 6 incluye la biblioteca de funciones **time.h**, necesaria para usar las funciones **time** (líneas 11 y 13) y **difftime** (línea 16).

La función **time** y el tipo **time_t** se han descrito con anterioridad brevemente en el Ejemplo 3.16, basta ahora con recordar que sirven para leer la fecha y hora del sistema en donde se ejecuta el programa. Por otro lado, la función **difftime** calcula la diferencia en segundos que existe entre las variables *comienzo* y *final* (líneas 9, 11 y 13).

```

1  /* Programa para medir el tiempo en segundos de un proceso largo
2     (version corta).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include <time.h>
7
8  int main(){
9     time_t comienzo, final;
10
11     comienzo = time(NULL);
12     /* Proceso largo: fibonacciRecursivo(46) */
13     final = time(NULL);
14
15     printf("Numero de segundos transcurridos: %.2f s\n", difftime(final,
16     comienzo));
17     return 0;
18 }
```

Ejemplo 5.5: Esqueleto para medir el tiempo de ejecución en segundos de un proceso largo

La línea 12 representa el proceso al que se desea tomar el tiempo de ejecución en segundos. El Ejemplo 5.5 también sugiere que se incluya ahí el llamado a la función recursiva de la serie de Fibonacci con un argumento de 46 por proponer un determinado valor. Resultaría sumamente interesante probar éste y otros valores.

El Ejemplo 5.5 como tal y por sí mismo, no ejecuta ni determina nada. De hecho si lo ejecuta tal cual visualizará siempre una salida de cero segundos. Sin embargo, dicho ejemplo resultará de mucha utilidad para poder medir, de manera empírica y poco precisa hay que decirlo⁶, el tiempo en segundos

⁶Aunque para los seres humanos es poco, para una computadora un segundo es muchísimo tiempo, ya que en un segundo se procesan millones de micro instrucciones en el pro-

que le toma a una función recursiva por ejemplo, terminar sus cálculos. En la sección de ejercicios se le pedirá que trabaje con el Ejemplo 5.5.

5.3. Torres de Hanoi

Existen varias versiones y variantes del problema de las torres de Hanoi, pero la versión clásica es la siguiente:

Se tienen tres postes, normalmente denotados como **A**, **B** y **C**, y en uno de ellos (poste de origen, usualmente **A**) se tienen apilados n discos de diferentes diámetros.

El problema consiste en pasar los n discos del poste origen al poste destino (usualmente **C**) manteniendo en todo momento las siguientes reglas:

1. Sólo se puede mover un disco cada vez.
2. Un disco de mayor diámetro no puede estar sobre uno de menor diámetro.
3. Sólo se puede mover el disco que se encuentre en la parte superior de cada poste.

Ahora bien, para el caso de $n = 1$, si **A** es el poste origen, **C** el poste destino, y **B** el poste auxiliar, la solución es directa:

Mover el disco 1 del poste A al poste C.

Para el caso de $n = 2$, si **A** es el poste origen, **C** el poste destino, y **B** el poste auxiliar, la solución está dada por los siguientes movimientos de discos, donde el disco 1 es el más pequeño y el disco 2 es el más grande:

Mover disco 1 del poste A al poste B
Mover disco 2 del poste A al poste C
Mover disco 1 del poste B al poste C

Para el caso de $n = 3$, si **A** es el poste origen, **C** el poste destino, y **B** el poste auxiliar, la solución está dada por los siguientes movimientos de discos, donde el disco 1 es el más pequeño y el disco 3 es el más grande:

cesador.

Mover disco 1 del poste A al poste C
Mover disco 2 del poste A al poste B
Mover disco 1 del poste C al poste B
Mover disco 3 del poste A al poste C
Mover disco 1 del poste B al poste A
Mover disco 2 del poste B al poste C
Mover disco 1 del poste A al poste C

Para visualizar mejor la solución, se sugiere como ejercicio realizar los dibujos que reflejen la naturaleza de estos movimientos. Adicionalmente, se sugiere realizar la misma labor para cuatro y cinco discos. Hágalo antes de continuar.

Si realizó lo sugerido en el párrafo anterior, quizá haya identificado algunos patrones:

- Para el caso de cuatro discos, se tiene la situación inicial sugerida por la Figura 5.7.
- Para el caso de cuatro discos (en general para el caso de n discos), después de algunos movimientos se tiene la situación presentada en la Figura 5.8, en la que se observan los $n - 1$ discos colocados en la posición auxiliar **B**, y el disco de mayor diámetro listo para ser movido a su posición final.
- El siguiente paso es el movimiento del disco de mayor diámetro de la posición origen **A** hacia su posición final de destino (**C**). Observe que los discos en la posición auxiliar no se mueven, como lo muestra la Figura 5.9.
- Ahora bien, de la misma Figura 5.9 puede notarse lo siguiente: en la posición **B** están los $n - 1$ discos restantes, es decir, el problema original de n discos pero simplificado (al menos por un disco), por lo que, si se toma como nuevo origen la posición **B**, se mantiene el destino en la posición **C** y se hace que la posición auxiliar sea ahora **A**, se tiene la misma situación del problema original pero con menos discos y con nuevos roles respecto a la posición de origen y la auxiliar.

En resumen el problema se reduce ahora a mover los $n - 1$ discos de la posición **B** a la posición **C** utilizando a la posición **A** como auxiliar, y si se

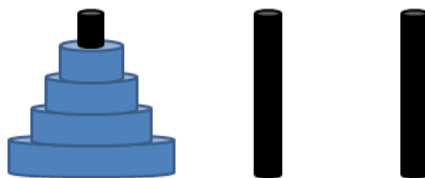


Figura 5.7: Estado inicial para cuatro discos en el problema de las torres de Hanoi

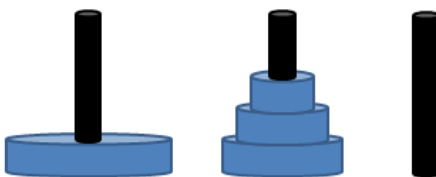


Figura 5.8: Después de algunos movimientos, los $n - 1$ discos están en la posición auxiliar **B**

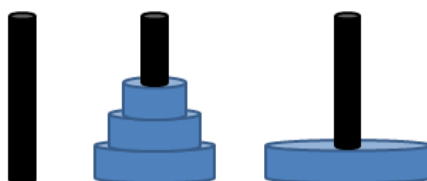


Figura 5.9: Movimiento del disco de mayor diámetro a su posición final

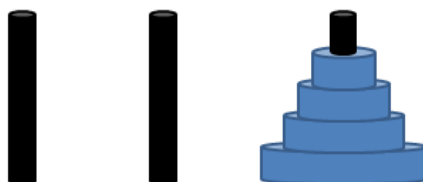


Figura 5.10: Situación final para el problema de las torres de Hanoi con cuatro discos

```
if ( $n = 1$ )  
    mover el disco  $n$  de la posición A a la posición C  
else  
    mover los  $n - 1$  discos de A a B utilizando a C  
    mover el disco  $n$  de la posición A a la posición C  
    mover los  $n - 1$  discos de B a C utilizando a A  
end if
```

Tabla 5.2: Algoritmo de las Torres de Hanoi

repite el procedimiento, se obtendrá eventualmente la situación presentada por la Figura 5.10.

De lo anterior se pueden deducir las relaciones de recurrencia y el caso base para dar solución al problema de las torres de Hanoi. La Tabla 5.2 describe el algoritmo de las torres de Hanoi.

Observe que el algoritmo anterior es una descripción en pseudo código de los patrones identificados en el texto. Por otro lado, si se realizó el ejercicio sugerido, un análisis cuidadoso deberá llevar al lector a la identificación de dichos patrones. Se deja como ejercicio la traducción del algoritmo de la Tabla 5.2 en una función recursiva.

5.4. Consideraciones finales

En resumen, las implementaciones recursivas son en general ineficientes en tiempo, pero muy elegantes y simples una vez que se tiene identificada la relación de recurrencia. En éste sentido, siempre que pueda optar por un enfoque iterativo que sea sencillo de implementar, quédese con él.

Por otro lado, el problema de las torres de Hanoi es un ejemplo ineludible de que, por medio de la recursividad, es posible solucionar problemas complejos de manera bastante sencilla, aunque el precio a pagar sea la eficiencia en tiempo. Intente resolver el problema de las torres de Hanoi de manera iterativa para tener una mejor comprensión de lo anterior, y experimente en carne propia el eterno conflicto (*tradeoff*) computacional entre espacio vs. tiempo, y eficiencia vs. simplicidad.

5.5. Ejercicios

1. Agregue al menos otro nivel de recursividad a la Figura 5.1, siguiendo el proceso descrito en el texto.
2. Haga un análisis de los llamados y retornos recursivos como los realizados en la Figura 5.3 para los llamados de función *factorial(5)* y *factorial(6)*.
3. Realice en una hoja de papel el árbol de recursividad (vea la Figura 5.5) que se genera para el cálculo de *fibonacci(5)* y *fibonacci(6)*.
4. Compare las versiones recursivas con las versiones iterativas descritas en el texto y analícelas. Simplemente por la lectura del código, ¿qué versión representa una mejor abstracción de las definiciones matemáticas?. Realice diferentes ejecuciones y determine ¿qué versión tiene un mejor rendimiento?.
5. Tanto el factorial como la serie de Fibonacci crecen de manera exponencial y no hay tipo de dato que las pueda contener. En el lenguaje de programación C, los tipos de datos numéricos dividen su rango tanto en números positivos como en números negativos, por lo que su capacidad de almacenamiento se ve, por decirlo de alguna manera, dividida. Sin embargo, existe el modificador **unsigned** (sin signo), que instruye al compilador para usar todo el rango de bits del tipo de dato para representar números positivos incluyendo el cero. Pruebe con este modificador de tipo de datos, documéntese en su funcionamiento y amplíe la capacidad de representación para los resultados de los ejemplos estudiados en este capítulo.
6. En base a lo indicado en el ejercicio anterior, calcule con la versión iterativa y recursiva respectivamente, *fibonacci(49)*, ¿qué observa? ¿qué puede deducir?
7. En base a lo expuesto en el texto, y basándose en el Ejemplo 5.5, determine el tiempo en segundos que tarda la función de Fibonacci en determinar por ejemplo *fibonacci(49)* en sus respectivas versiones: iterativa y recursiva.

8. Escriba un programa que, en base al algoritmo descrito en la Tabla 5.2, resuelva el problema de las torres de Hanoi por medio de una función recursiva. Note que es un proceso de simple traducción, ya que tanto las relaciones de recurrencia, como el criterio de paro están dados en el algoritmo.
9. Basándose en el ejercicio anterior y en el Ejemplo 5.5, determine el tiempo en segundos que tarda la función que da solución al problema de las torres de Hanoi para 20, 35 y 50 discos por ejemplo.
10. Escriba un programa que, a través de una función recursiva, calcule el producto de dos números enteros positivos a y b , tomando en cuenta que:

$$a * b = a + a + \dots + a$$

significa que a se suma tantas veces como lo indique b , donde $a, b > 0$.

11. Escriba un programa que, a través de una función recursiva, calcule el cociente entero de dos números enteros positivos a y b , esto es:

$$\begin{array}{llll} a/b \Rightarrow & a - b = c1 & \text{si } c1 > 0 & \text{entonces} \\ & c1 - b = c2 & \text{si } c2 > 0 & \text{entonces} \\ & c2 - b = c3 & \text{si } c3 > 0 & \text{entonces} \\ & & \vdots & \end{array}$$

El cociente estará dado por el número de veces que se haya podido repetir recursivamente el procedimiento descrito, donde $a, b > 0$.

12. Escriba un programa que, a través de una función recursiva, calcule la potencia de dos números enteros positivos a y b , tomando en cuenta que:

$$a^b = a * a * \dots * a$$

significa que a se multiplica tantas veces como lo indique b , donde $a, b > 0$.

Capítulo 6

Arreglos

Antes de iniciar con los conceptos y la descripción de los arreglos, considere el siguiente problema:

Se desea almacenar un grupo de cuatro calificaciones, y determinar:

- Cuál es la menor.
- Cuál es la mayor.
- Cuáles y cuántas son aprobatorias y reprobatorias según un determinado criterio.
- Su media aritmética (promedio).
- Si existe o no una calificación específica.

Finalmente, las calificaciones deben ser ordenadas de manera decreciente.

Con lo que sabe hasta ahora, ¿puede dar solución a este problema?, si en lugar de cuatro calificaciones se tuvieran que procesar diez, ¿qué cambios tendría que hacer?, ¿y si en lugar de diez fueran 100?, ¿y si fueran 1000?.

Su respuesta debería ser sí se puede resolver el problema sin importar el número de calificaciones, pero por cada nuevo incremento en el número de calificaciones, tendrían que agregarse nuevas variables con sus respectivos identificadores, además de modificar el código que realiza las tareas solicitadas para considerar a estas nuevas variables. Dicha labor, además de tediosa es ardua, pero sobre todo, ineficiente.

En base a lo anterior, debería ser claro que se requiere algo más, se necesita otra cosa, algo que permita manejar de mejor manera este tipo de problemas. Pues bien, eso que hace falta es lo que se conoce como arreglos.

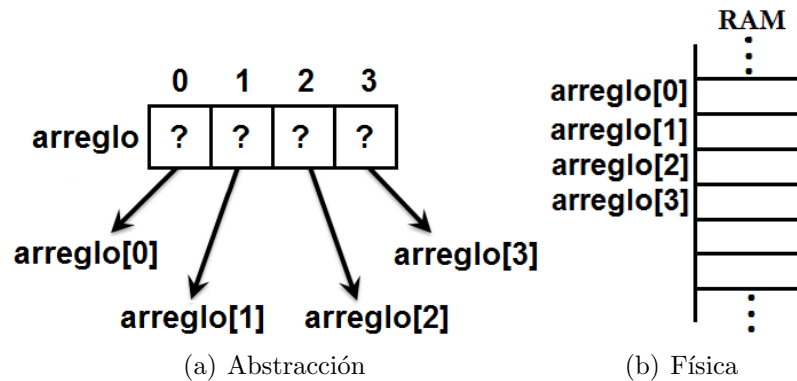


Figura 6.1: Representación de un arreglo

Un arreglo es una **estructura de datos** con un tamaño determinado, que agrupa elementos del **mismo** tipo de datos, los cuales están relacionados por un nombre (identificador) y se distinguen por uno o más índices.

En este capítulo se analizarán los arreglos de una dimensión, también conocidos como vectores, así como los arreglos de dos dimensiones o matrices, los arreglos de tres dimensiones, y una generalización para los arreglos de n dimensiones.

6.1. Arreglos de una dimensión (vectores)

Los arreglos de una dimensión se denominan comúnmente arreglos o vectores, en este texto se utilizará la primera denominación.

6.1.1. Conceptos, representación y estructura

Los **arreglos** son una estructura de datos lineal de **elementos contiguos**, relacionados por un **mismo tipo de datos**, un **mismo nombre** (identificador), y **distinguidos por un índice**.

La Figura 6.1 muestra la representación de un arreglo de tamaño cuatro de nombre *arreglo*. En (a) se observa la representación abstracta de dicho arreglo, mientras que en (b) se presenta la forma en que dicho arreglo es almacenado en la memoria principal de la computadora. Note que los arreglos se representan de forma directa en la memoria de la computadora.

Los arreglos en C, siempre empiezan en el índice 0 y terminan en *tamaño* - 1 , dónde *tamaño* se refiere al tamaño, dimensión o longitud del arreglo.

Un aspecto importante a considerar respecto a la conveniencia en el uso de arreglos es la siguiente: no es lo mismo declarar 100 variables de tipo entero, que un arreglo de 100 enteros. ¿Puede imaginar lo terriblemente tedioso que sería declarar 100 variables de tipo entero?.

Otra ventaja de utilizar arreglos en lugar de una tediosa declaración masiva de variables, es que el grupo de datos relacionados por el nombre del arreglo están contiguos en la memoria, mientras que el grupo de variables declaradas de manera independiente, no necesariamente.

En general, la declaración de un arreglo en el lenguaje de programación C tiene la siguiente estructura general:

```
tipo_de_dato nombre_del_arreglo[TAMAÑO];
```

en donde:

- **tipo_de_dato** es cualquier tipo de dato válido en C.
- **nombre_del_arreglo** es un identificador válido en C.
- **TAMAÑO** es la longitud que tendrá el arreglo.

6.1.2. Declaración, inicialización y recorrido

El Ejemplo 6.1 muestra la declaración, inicialización y recorrido de un arreglo de enteros.

En la línea 10 se está declarando un arreglo de nombre *arreglo* de tamaño N . Observe que el arreglo está siendo inicializado, en la declaración o en **tiempo de compilación**, y lo que hace el compilador es asignarle 0 al elemento *arreglo*[0] y 1 al elemento *arreglo*[1], todos los demás elementos del arreglo se inicializan **por omisión** en 0 , es decir:

$$\text{arreglo}[0] = 0, \text{arreglo}[1] = 1, \text{arreglo}[2] = 0, \dots, \text{arreglo}[N-1] = 0$$

Si un arreglo, independientemente de su tipo de dato, se declara pero no se inicializa, el contenido de cada uno de sus elementos no está determinado a un valor concreto o específico, lo cual implica que contendrán lo que en el *argot* computacional se conoce como basura.

```
1  /* Programa que muestra la declaracion, el uso y
2     recorrido de arreglos de tipo entero.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  #define N 15
8
9  int main() {
10     int arreglo[N] = {0, 1};
11     int i;
12
13     for(i = 2; i < N; i++)
14         arreglo[i] = arreglo[i-1] + arreglo[i-2];
15
16     printf("Valores almacenados en el arreglo:\n");
17
18     for(i = 0; i < N; i++)
19         printf("arreglo[%2d] = %3d\n", i, arreglo[i]);
20
21     return 0;
22 }
```

Ejemplo 6.1: Uso de un arreglo de enteros

La línea 13 está realizando un **recorrido** del arreglo; note que para dicho recorrido, el índice se inicializa en 2, mostrando así que es posible iniciar el recorrido de un arreglo en donde se necesite, y no siempre desde 0.

Por otro lado, la línea 14 está asignándole valores al arreglo a partir de la posición 2. Observe que el valor *i-ésimo* del arreglo se define como la suma de los dos valores anteriores, por lo que en realidad lo que hace el ciclo for es inicializar el arreglo con los valores de la serie de Fibonacci que se estudió en la Sección 5.2.2.

Finalmente, las líneas 18 y 19 son lo que se conoce como un **recorrido para impresión** del arreglo, en el cual se imprime por renglón en la salida estándar, cada uno de los elementos del arreglo *arreglo*. La salida del Ejemplo 6.1 se muestra en la Figura 6.2.

Ahora bien, el Ejemplo 6.2 muestra otra variante de inicialización de arreglos, y un doble recorrido para un arreglo de tipo *float*.

Observe que el arreglo *a* de la línea 10 no tiene un tamaño explícito. Cuando se declara un arreglo en C, éste debe tener una dimensión o tamaño definidos explícita o implícitamente; para el caso del Ejemplo 6.2, el tamaño está implícito, y es determinado por el número de inicializadores (5), los cuales son calculados por el compilador, y no están relacionados con el **#define** de la línea 7, al menos en cuanto a la definición del tamaño del arreglo se

```
Valores almacenados en el arreglo:
arreglo[ 0] =  0
arreglo[ 1] =  1
arreglo[ 2] =  1
arreglo[ 3] =  2
arreglo[ 4] =  3
arreglo[ 5] =  5
arreglo[ 6] =  8
arreglo[ 7] = 13
arreglo[ 8] = 21
arreglo[ 9] = 34
arreglo[10] = 55
arreglo[11] = 89
arreglo[12] = 144
arreglo[13] = 233
arreglo[14] = 377
```

Figura 6.2: Salida del Ejemplo 6.1

refiere.

```
1  /* Programa que muestra la declaracion, el uso y
2     recorrido de arreglos de tipo float.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  #define N 5
8
9  int main(){
10     float a[ ] = {1.6, 1.5, 1.4, 1.3, 1.2};
11     int i, j;
12
13     printf("Valores almacenados en el arreglo:\n");
14
15     for(i = 0, j = N - 1; i < N; i++, j--)
16         printf("a[%d] = %.2f\ta[%d] = %.2f\n", i, a[i], j, a[j]);
17
18     return 0;
19 }
```

Ejemplo 6.2: Uso de un arreglo de tipo **float**

La estructura de repetición **for** de la línea 15, utiliza dos variables de control para hacer un recorrido ascendente (*i*) y descendente (*j*) del arreglo *a*, con la finalidad de mostrar que no siempre ni necesariamente, los arreglos se recorren de manera ascendente. La salida del Ejemplo 6.2 se muestra en la Figura 6.3.

Valores almacenados en el arreglo:	
a[0] = 1.60	a[4] = 1.20
a[1] = 1.50	a[3] = 1.30
a[2] = 1.40	a[2] = 1.40
a[3] = 1.30	a[1] = 1.50
a[4] = 1.20	a[0] = 1.60

Figura 6.3: Salida del Ejemplo 6.2

6.1.3. Arreglos y funciones

Los datos almacenados en los arreglos no siempre provienen de inicializaciones, es muy usual por ejemplo, el procesar un conjunto de datos de la entrada estándar y almacenarlos en un arreglo, y dado que es una tarea muy común, convendría tener una forma de encapsular ésta labor y solicitarla en dónde se necesite. El mecanismo de abstracción proporcionado por las funciones cumplen con dicho cometido.

En base a lo descrito con anterioridad, lo que se necesita entonces es saber cómo enviarle un arreglo a una función y cómo decirle a la función que lo que va a recibir como parámetro es un arreglo. El Ejemplo 6.3 muestra cómo hacer esto para un arreglo de enteros.

Observe primero las líneas 9 y 10, y note que cada una de las funciones tienen en su lista de parámetros *int []* y *n*. Estas declaraciones le indican al compilador que las funciones recibirán un arreglo de enteros como primer argumento, y un número entero como segundo argumento. Lo mismo sucede en las líneas 29 y 38 respectivamente.

Continuando con los prototipos (líneas 9 y 10) y los encabezados de las definiciones de función (líneas 29 y 38), note que en ambos casos los corchetes están vacíos, esto es así debido a que no se está declarando un arreglo ni designado un tamaño, sino que se está indicando que la función recibirá un arreglo que ha sido declarado y dimensionado en alguna otra parte (comúnmente en **main**).

Observe también que el nombre (identificador) del arreglo dentro de la función *leeArregloInt* es *a*, el cual coincide con el nombre del arreglo declarado en la función **main** en la línea 13, mientras que para la función *imprimeArregloInt* el nombre del arreglo es *b*; esto se ha hecho con la intención de mostrar que el nombre del (los) arreglo(s) dentro de las funciones puede(n) coincidir o no con el nombre del (los) arreglo(s) declarado(s) en **main** sin

que exista algún tipo de conflicto.

```

1  /* Programa que muestra el uso de arreglos como argumentos
2     y parametros de funciones.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  #define N 100
8
9  void leeArregloInt(int [ ], int);
10 void imprimeArregloInt(int [ ], int);
11
12 int main(){
13     int n, a[N];
14
15     do{
16         printf("Numero de datos a procesar?: ");
17         scanf("%d", &n);
18     }while(n < 1 || n > (N - 1));
19
20     printf("Introduza los datos a almacenar en el arreglo:\n");
21     leeArregloInt(a, n);
22
23     printf("\nLos datos almacenados en el arreglo son:\n");
24     imprimeArregloInt(a, n);
25
26     return 0;
27 }
28
29 void leeArregloInt(int a[ ], int n){
30     int i;
31
32     for(i = 0; i < n; i++){
33         printf("arreglo[%d] = ? ", i);
34         scanf("%d", &a[i]);
35     }
36 }
37
38 void imprimeArregloInt(int b[ ], int n){
39     int i;
40
41     for(i = 0; i < n; i++)
42         printf("arreglo[%d] = %d\n", i, b[i]);
43 }

```

Ejemplo 6.3: Arreglos y funciones

Tanto la función *leeArregloInt* como *imprimeArregloInt* pueden ser integradas dentro de una biblioteca personalizada de funciones tal y como se describió en la Sección 4.1, con la finalidad de poder reutilizarlas cuando se necesite leer de la entrada estándar, un arreglo de números enteros, o imprimir un arreglo de enteros. Una posible salida del Ejemplo 6.3 se muestra en la Figura 6.4.

```
Numero de datos a procesar?: -8
Numero de datos a procesar?: 666
Numero de datos a procesar?: 5
Introduza los datos a almacenar en el arreglo:
arreglo[0] = ? 2122
arreglo[1] = ? 1974
arreglo[2] = ? 2018
arreglo[3] = ? 2007
arreglo[4] = ? 2008

Los datos almacenados en el arreglo son:
arreglo[0] = 2122
arreglo[1] = 1974
arreglo[2] = 2018
arreglo[3] = 2007
arreglo[4] = 2008
```

Figura 6.4: Salida del Ejemplo 6.2

6.1.4. Ordenamiento por burbuja

El ordenamiento de elementos es uno de los problemas clásicos de la computación. En esta sección se presentará un algoritmo clave y sencillo, aunque no eficiente, para el ordenamiento de elementos: el ordenamiento por burbuja.

Considere la siguiente secuencia de elementos:

15, -4, 0, -9, 7, 10

Con toda seguridad, si consigue una hoja de papel y un lápiz, podrá ordenarlos ascendentemente, obteniendo como resultado:

-9, -4, 0, 7, 10, 15

Ahora bien, ¿cuál fue el procedimiento que siguió para ordenarlos?, ¿lo podría especificar y detallar para convertirlo en un algoritmo?.

No hay una única forma de ordenar una secuencia de elementos, aunque quizá la más común como ser humano, sea la búsqueda del elemento menor de la lista, colocarlo en su lugar, y buscar el siguiente elemento menor de la lista, colocarlo en su lugar, y continuar así sucesivamente hasta terminar con toda la secuencia de elementos a ordenar.

El algoritmo de la burbuja se basa en ir comparando elementos contiguos, es decir, tomar el elemento i -ésimo y compararlo con el elemento i -ésimo + 1, si ese par de elementos no está ordenado, entonces se intercambian, si lo están, se dejan tal cual se encontraron.

Para la secuencia anterior, el algoritmo de la burbuja generaría los siguientes intercambios para el primer recorrido:

15, -4, 0, -9, 7, 10	
-4, 15, 0, -9, 7, 10	
-4, 0, 15, -9, 7, 10	
-4, 0, -9, 15, 7, 10	Recorrido 1
-4, 0, -9, 7, 15, 10	
-4, 0, -9, 7, 10, 15	

Observe que los elementos aún no están ordenados, sin embargo, el mayor de ellos (15) fue intercambiándose y ascendiendo a su posición final, razón por la cual el algoritmo recibe el nombre de burbuja, en analogía al efecto ascendente de las burbujas de aire en el agua.

La secuencia anterior de intercambios, constituye de hecho el *primer recorrido* para la lista de elementos, y en principio, se necesitan $n - 1$ recorridos para una lista de n elementos y $n - 2$ comparaciones dentro de cada recorrido para saber si realiza o no el intercambio.

Siguiendo la idea planteada hasta ahora, termine de hacer los recorridos restantes, es decir: los cuatro recorridos que faltan con sus cinco iteraciones de comparación en cada uno.

El Ejemplo 6.4 muestra un archivo de biblioteca personalizada (.h) que contiene dos versiones para el ordenamiento por burbuja. La versión que nos compete por el momento es la función definida en las líneas 23-34, la cual implementa la versión tradicional del ordenamiento por burbuja, también conocido como *burbuja ingenuo*.

La estructura de repetición **for** de la línea 26, controla el número de veces que se recorre el arreglo, que como ya se mencionó es de $n - 1$ veces, donde n es el número de elementos a ordenar. Ahora bien, para una n muy grande se tiene:

$$\lim_{n \rightarrow \infty} (n - 1) \rightarrow \infty \quad (6.1)$$

Por lo que se puede decir que un arreglo de n elementos se recorre n veces.

```

1  /* Biblioteca de funciones que implementa el ordenamiento
2     por burbuja "ingenuo" y burbuja mejorado.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #define VERDADERO 1
6  #define FALSO     0
7
8  void burbujaMejorado(int a[], int n){
9      int veces, i, aux, bandera = VERDADERO;
10
11      for(veces = 1; veces < n && bandera; veces++){
12          bandera = FALSO;
13          for(i = 0; i < n - veces; i++){
14              if(a[i] > a[i + 1]){
15                  aux = a[i];
16                  a[i] = a[i + 1];
17                  a[i + 1] = aux;
18                  bandera = VERDADERO;
19              }
20          }
21      }
22
23  void burbuja(int a[], int n){
24      int veces, i, aux;
25
26      for(veces = 1; veces < n; veces++){
27          for(i = 0; i < n - 1; i++){
28              if(a[i] > a[i + 1]){
29                  aux = a[i];
30                  a[i] = a[i + 1];
31                  a[i + 1] = aux;
32              }
33          }
34      }

```

Ejemplo 6.4: Funciones para el ordenamiento por burbuja y burbuja mejorado

Por otro lado, el ciclo **for** de la línea 27 realiza en general $n - 2$ iteraciones para comparar parejas de números candidatas a intercambio para los n elementos del arreglo, pero por la misma razón que antes (Ecuación 6.1), es posible decir que se repite n veces. Esto quiere decir que, por cada recorrido se realizan n comparaciones.

En base a lo anterior, el ordenamiento por burbuja tiene el siguiente comportamiento general:

$$O(n^2) \tag{6.2}$$

donde n es el número de elementos a ordenar.

La notación de la Ecuación (6.2) se lee como "o grande de n cuadrada", y su explicación queda fuera de los alcances de este libro, pero por ahora,

basta con saber que el ordenamiento por burbuja tiene un comportamiento cuadrático.

Burbuja mejorado

Si terminó de realizar las iteraciones propuestas en la sección anterior para ordenar la secuencia de números que se presentó, probablemente se haya dado cuenta de que la secuencia está ordenada después de la primera iteración de comparación del tercer recorrido.

¿Tiene sentido entonces realizar los recorridos e iteraciones de comparación restantes? Sinceramente espero que su respuesta haya sido negativa.

El ordenamiento por burbuja clásico no hace ningún tipo de verificación, por lo que realizaría siempre todos los recorridos e iteraciones de comparación basándose única y exclusivamente en el número de elementos a ordenar, aún cuando los datos estuvieran ordenados desde el principio, de ahí que al ordenamiento por burbuja clásico también se le denomine ingenuo.

Para mejorar esta deficiencia, se pueden hacer esencialmente dos cosas basadas en las siguientes consideraciones:

1. Después de cada recorrido, el elemento mayor de serie de elementos se encontrará ya en su posición final; en el segundo recorrido el segundo elemento mayor y así sucesivamente, ¿tiene sentido seguir comparando estos elementos que sabemos ya ordenados? La respuesta es no, por lo que si después de cada recorrido reducimos también el número de iteraciones de comparación, se mejorará la eficiencia. Ésta es precisamente la mejora que se implementa en el ciclo **for** de la línea 13 del Ejemplo 6.4, en donde la expresión condicional se ha modificado para que dependa también del número de *veces* que se ha realizado el recorrido.
2. Si la serie de elementos se encuentra ya ordenada en alguna parte del proceso de ordenamiento antes de las n^2 iteraciones, tampoco tendría sentido continuar. Para éste caso, se ha utilizado una *bandera* (línea 9), la cual asumirá que después de cada recorrido, la serie de elementos estará probablemente ordenada, por lo que la bandera se apaga (línea 12) para que la expresión condicional del ciclo **for** de la línea 11 sea falsa y los recorridos sobre la serie de elementos terminen; sin embargo, si la expresión condicional del **if** de la línea 14 se cumple para al menos una de las iteraciones de comparación, entonces la serie de elementos

no está ordenada y se enciende nuevamente la bandera (línea 18) para procesar al menos un nuevo recorrido.

En el *argot* computacional, una *bandera* es un indicador (centinela), y es una variable que sirve para indicar una determinada situación.

Las banderas más comunes son las banderas binarias o *booleanas*, las cuales toman únicamente dos valores: *1* (verdadero) o *0* (falso).

El lenguaje de programación C no tiene tipos de datos *booleanos*, por lo que tienen que ser simulados o implementados con variables de tipo entero. Cuando a una bandera *booleana* se le asigna *verdadero*, se dice que se *enciende* la bandera, y cuando se le asigna *falso*, se dice que se *apaga* la bandera, en analogía a un interruptor (*switch*) de encendido/apagado.

6.1.5. Búsqueda lineal y búsqueda binaria

Búsqueda lineal

La búsqueda lineal es muy sencilla y natural, dado que es la forma en que los seres humanos realizamos búsquedas.

La idea es bastante simple, se va recorriendo uno a uno la secuencia de elementos sobre los que se desea realizar la búsqueda (*conjunto de datos*), y se va comparando cada uno de ellos con el elemento a buscar (*clave* o *llave*); si existe coincidencia, entonces la búsqueda ha tenido éxito y se reporta el índice (*posición*) del elemento dentro del conjunto de datos.

También puede ocurrir que se haya recorrido todo el conjunto de datos y no se haya encontrado coincidencia con la clave, en éste caso, se reporta dicha situación con una posición (índice) no válida (-1) respecto al conjunto de datos.

La función *busquedaLineal* de las líneas 8-15 del Ejemplo 6.5, recibe un arreglo de enteros constantes¹ *a* (conjunto de datos), un entero *x* (*clave* o *llave*), y un número *n* que representa la cantidad de datos del conjunto sobre los que se realizará la búsqueda².

El ciclo **for** de la línea 11 realiza el recorrido sobre el conjunto de datos, mientras que el **if** de la línea 12 verifica la coincidencia con el elemento clave;

¹El modificador **const** le indica al compilador que el arreglo es de elementos *constantes*, y previene al programador de alguna modificación accidental o incidental sobre los elementos del arreglo afectado por el modificador. En resumen, vuelve al arreglo afectado de sólo lectura: no es posible modificar sus elementos dentro de la función.

²Podría ser todo el conjunto de datos o sólo una parte de él.

en caso de existir, se reporta en la línea 13, pero si el elemento clave no existe, se reporta dicha situación en la línea 14.

Búsqueda binaria

La búsqueda binaria se fundamenta en un pre requisito sumamente importante: que el conjunto de datos esté **ordenado**.

```
1  /* Biblioteca de funciones que implementa la búsqueda
2     lineal y la búsqueda binaria (iterativa).
3     @autor Ricardo Ruiz Rodriguez
4  */
5
6  #define NO_ENCONTRADO -1
7
8  int busquedaLineal(const int a[ ], int x, int n){
9      int i;
10
11     for(i = 0; i < n; i++){
12         if(x == a[i])
13             return i;
14     }
15     return NO_ENCONTRADO;
16 }
17
18 int busquedaBinaria(const int a[ ], int x, int lim_inf, int lim_sup){
19     int medio;
20
21     while(lim_inf <= lim_sup){
22         medio = (lim_inf + lim_sup) / 2;
23         if (x == a[medio])
24             return medio;
25         else if(x < a[medio])
26             lim_sup = medio - 1;
27         else
28             lim_inf = medio + 1;
29     }
30     return NO_ENCONTRADO;
31 }
```

Ejemplo 6.5: Funciones para la búsqueda lineal y la búsqueda binaria

La importancia de tal pre requisito es tal que, si no se cumple, no se garantiza el buen funcionamiento de la búsqueda binaria.

La búsqueda binaria capitaliza el hecho de que los datos estén ordenados para ir eliminando en cada iteración, a la mitad de los datos, lo cual la hace muy eficiente, pero el costo a pagar es el ordenamiento previo de los datos.

Una analogía con la forma de realizar la búsqueda de una persona en un directorio telefónico, puede ayudar a visualizar mejor el funcionamiento de la búsqueda binaria.

Suponga que se desea buscar en un directorio telefónico el siguiente nombre: *Ricardo Ruiz Rodríguez*. En México los nombres aparecen registrados en el directorio en base al primer apellido, segundo apellido y nombre(s) de las personas registradas. Supongamos también que se decide abrir el directorio por la mitad (aproximadamente), y que la lista de apellidos que vemos es *Murrieta*.

¿Hacia qué parte del directorio deberíamos dirigir la búsqueda?. Debería ser obvio que hacia la segunda mitad, ya que *Ruiz* se encuentra alfabéticamente después que *Murrieta*. Pues bien, ésta es en esencia la idea de la búsqueda binaria.

El Ejemplo 6.5 muestra en las líneas 17-31 la función que implementa el mecanismo de búsqueda binaria.

La función recibe (línea 17) un arreglo de elementos constantes a (conjunto de búsqueda), el elemento a buscar x (clave o llave), y dos números que representan las posiciones de los límites inferior (lim_inf) y superior (lim_sup) del conjunto de búsqueda respectivamente. Estos límites en general coinciden con el índice inferior y superior del arreglo, pero no necesariamente tiene que ser así, el conjunto de búsqueda podría ser un subconjunto de todos los elementos.

La expresión condicional de la estructura de repetición **while** de la línea 20, controla la continuidad o no de la búsqueda del elemento x sobre el arreglo a . En cuanto ya no se cumpla dicha expresión, se sabrá que el elemento clave no existe en el conjunto de búsqueda (¿por qué?), y dicha situación se reporta en la línea 30.

La línea 20 es la fórmula del punto medio, y calcula el elemento central del conjunto de búsqueda. En analogía al ejemplo del directorio telefónico, ésta sería la simulación de abrirlo aproximadamente a la mitad.

La estructura **if/else** anidada de las líneas 22-27 determina lo siguiente:

- Si la clave se encontró (línea 22), se regresa su posición dentro del conjunto (línea 23).
- Si no se encontró y hay que buscarla en la parte izquierda del conjunto (línea 24), se ajusta el límite superior (línea 25) para re definir un nuevo subconjunto de búsqueda.
- Si la clave no es igual ni menor respecto al elemento actual ($a[medio]$), por tricotomía de números, no le queda otra (línea 26) más que estar (si existe) en la parte derecha del conjunto de búsqueda, por lo que se

ajusta el límite inferior (línea 27), para re definir el nuevo subconjunto de búsqueda.

Finalmente, si no se ha encontrado la clave, y los límites del subconjunto de búsqueda son válidos, se repite nuevamente todo el proceso descrito, pero sobre un conjunto de búsqueda con menos datos, de hecho la mitad de los datos en cada iteración.

Probablemente el lector haya tenido una sensación parecida a un *deja vu* y haya venido a su mente el concepto de recursividad. La búsqueda binaria puede ser abordada utilizando un enfoque recursivo. En la sección de ejercicios tendrá la oportunidad de poner en práctica dicho enfoque.

6.1.6. Arreglos de caracteres (cadenas)

Todo lo que hasta este momento se ha comentado para los arreglos respecto a su declaración, inicialización, recorrido, y uso con funciones, es aplicable a arreglos de cualquiera de los tipos de datos de C incluyendo **char**, sin embargo, éste tipo de arreglos poseen características particulares, empezando porque es común denominar a un arreglo de caracteres (**char**) como **cadena**, de ahí que su tratamiento y discusión se hayan postergado hasta ahora y separado en una sección especial.

En este momento el lector cuenta con algo de experiencia en la declaración, inicialización, recorrido, y uso de arreglos con funciones, por lo que en el Ejemplo 6.6 se presentan estos conceptos en el contexto de cadenas, es decir: arreglos de caracteres.

Las líneas 12, 13 y 14 muestran la declaración de tres cadenas: *cadena1*, *cadena2* y *cadena3*. Note que las dos primeras no tienen un tamaño explícito y que han sido inicializadas de forma distinta.

La *cadena1* se ha inicializado con una frase contenida entre comillas dobles. En el lenguaje de programación C, cualquier sucesión de elementos contenida entre comillas dobles es una cadena, y dicha cadena está compuesta por todos los caracteres entre comillas (incluyendo los espacios en blanco), más un carácter no visible, pero que delimita el fin de la cadena. Éste carácter recibe el nombre de *fin de cadena* y es ‘\0’, que aunque es un carácter compuesto por dos símbolos (\ y 0), se considera como uno solo y se denota entre comillas simples: ‘\0’. Por lo tanto la *cadena1* tiene un tamaño implícito o longitud, no de 18 caracteres como podría pensarse en un principio, sino de 19.

```

1  /* Programa que muestra la declaracion, el uso y
2     recorrido de cadenas.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define N 50
7
8  void imprimeAlReves(char [ ]);
9  int longitud(char [ ]);
10
11 int main(){
12     char cadena1[] = "Anita lava la tina";
13     char cadena2[] = {'M', 'a', 'd', 'a', 'm', '\0'};
14     char cadena3[N];
15     int i;
16
17     printf("Cual es tu nombre? ");
18     gets(cadena3);
19     printf("Hola %s!!!\n", cadena3);
20
21     for(i = 0; cadena1[i] != '\0'; i++)
22         printf("%c ", cadena1[i]);
23     putchar('\n');
24
25     imprimeAlReves(cadena2);
26
27     return 0;
28 }
29
30 void imprimeAlReves(char c[ ]){
31     int i;
32
33     for(i = longitud(c) - 1; i >=0; i--)
34         putchar(c[i]);
35     putchar('\n');
36 }
37
38 int longitud(char c[ ]){
39     int i;
40
41     for(i = 0; c[i] != '\0'; i++)
42         ;
43     return i;
44 }

```

Ejemplo 6.6: Uso de cadenas

Por otro lado, la *cadena2* (línea 13) tiene una inicialización más parecida a lo que se vio en su momento para arreglos de tipo **int** y **float**. Note que cada uno de los elementos de la cadena, incluyendo el fin de cadena, se encuentran entre comillas simples. Para este tipo de inicialización el terminador de cadena ‘\0’ debe proporcionarse de manera explícita, ya que de otra forma, la cadena estaría mal formada, es decir, sin terminar, y esto podría llevar

a efectos o resultados desde inesperados³ hasta catastróficos⁴. Esta cadena tiene una longitud implícita de seis caracteres.

La *cadena3* no ha sido inicializada y se utilizará para leer una cadena desde la entrada estándar a través de la función *gets*⁵ (línea 18).

El ciclo **for** de la línea 21 muestra el *recorrido* clásico de una cadena. Observe que la expresión condicional depende del carácter de fin de cadena (`'\0'`), y que el cuerpo del ciclo (línea 22) imprime la cadena *cadena1* carácter por carácter con un espacio entre ellos.

La línea 23 muestra el uso de la función **putchar** para imprimir el carácter de avance de línea y un retorno de carro. Ésta función es equivalente a la sentencia *printf*(`"\n"`);. Note que la secuencia de escape `'\n'` ha sido escrita entre comillas simples, debido a que la función *putchar* escribe en la salida estándar el carácter que recibe como argumento.

El Ejemplo 6.6 tiene también dos funciones:

- *longitud* (línea 38): recibe una cadena, la cual se llamará *c* dentro de la función, y realiza un recorrido sobre *c* para llegar al final, de tal forma que la variable de control *i* contiene la longitud de *c* cuando la expresión condicional del **for** se evalúa como 0; el cuerpo del ciclo está vacío porque no hay nada más que hacer, y esto se enfatiza con el `;;` de la línea 42. Finalmente la función regresa la longitud de la cadena: *i* (línea 43).
- *imprimeAlReves* (línea 30): recibe una cadena, la cual se llamará *c* dentro de la función, y realiza un recorrido inverso sobre *c*, es decir, del último carácter de la cadena (*longitud* - 1) al primero (0). Dentro del recorrido, la cadena se imprime en la salida estándar carácter por carácter.

Note que la cadena original⁶ no se invierte, únicamente se imprime al revés, por lo tanto la cadena original permanece inalterada. Asegúrese de comprender esto antes de continuar. Una posible salida del Ejemplo 6.6 se muestra en la Figura 6.4.

³Impresión de caracteres extraños más allá de lo que en principio conforma la cadena.

⁴Intentos de acceso a memoria no reservada y la consecuente anulación del programa.

⁵La función *gets* recibe una cadena, y almacena en ella lo que se procesa de la entrada estándar, hasta que se introduzca un ENTER, (`\n`).

⁶Como argumento la cadena original es *cadena2* (línea 25), como parámetro es *c* (línea 30).

```
Cual es tu nombre? Ricardo
Hola Ricardo!!!
A n i t a   l a v a   l a   t i n a
madaM
```

Figura 6.5: Salida del Ejemplo 6.6

En el Capítulo 7 se retomará el tema de cadenas, por ahora es suficiente con el panorama general descrito hasta el momento, y con los ejercicios correspondientes al final del capítulo.

6.2. Arreglos de dos dimensiones (matrices)

Los arreglos de dos dimensiones también reciben el nombre de arreglos bidimensionales o simplemente matrices.

6.2.1. Conceptos, representación y estructura

Los **arreglos bidimensionales** son una estructura de datos de dos dimensiones de *elementos contiguos*, relacionados por *un mismo tipo de datos* y *un mismo nombre*, donde los elementos son *distinguidos por dos índices*.

La Figura 6.6 (a) muestra una representación de una matriz m de cuatro renglones y cuatro columnas (cuadrada). Los renglones y columnas de una matriz en C siempre empiezan en el índice 0, y terminan en el índice *tamaño* - 1 (3), donde *tamaño* se refiere al tamaño, dimensión o longitud del renglón o columna de la matriz respectivamente.

Por otro lado, la Figura 6.6 (b) muestra la representación física de la matriz m en la memoria principal de la computadora. Observe que la memoria de la computadora es una estructura lineal y no una estructura cuadrada (bidimensional), por lo que los elementos de la matriz son almacenados linealmente, de tal forma que el elemento $m[1][0]$ es el inmediato posterior del elemento $m[0][3]$, y que el elemento $m[2][0]$ es el inmediato posterior del elemento $m[1][3]$, y así sucesivamente.

En general, la declaración de un arreglo bidimensional en el lenguaje de programación C tiene la siguiente estructura:

```
tipo_de_dato nombre_de_la_matriz[TAMAÑO01][TAMAÑO02];
```

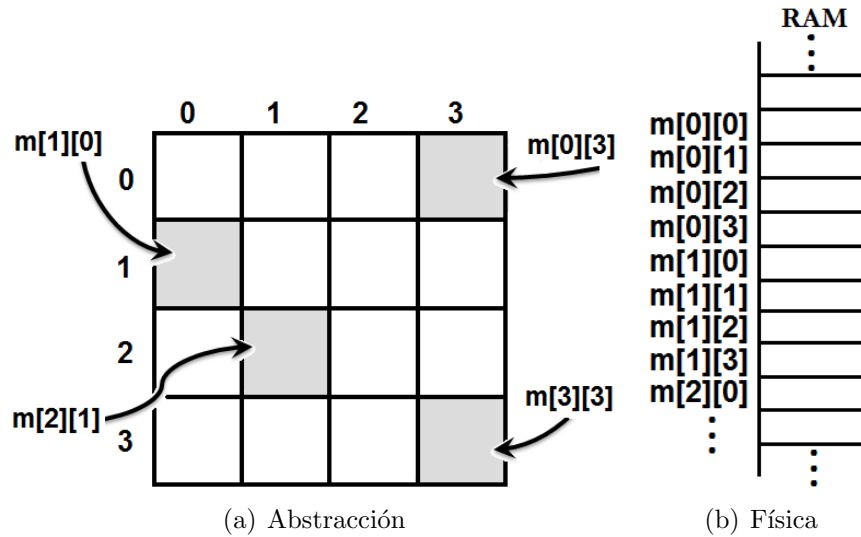



Figura 6.6: Representación de una matriz

donde:

- **tipo_de_dato** es cualquier tipo de dato válido en C.
- **nombre_de_la_matriz** es un identificador válido en C.
- **TAMAÑO1** es el número de renglones que tendrá la matriz.
- **TAMAÑO2** es el número de columnas que tendrá la matriz.

6.2.2. Declaración, inicialización, recorrido y uso con funciones

El Ejemplo 6.7 muestra la declaración, la inicialización, el recorrido, y el uso de matrices con funciones. Las líneas 6 y 7 definen dos constantes simbólicas M y N para el número de renglones y columnas de la matriz respectivamente. Éstas constantes determinarán las dimensiones *reales* o *físicas* de *matriz1*, *matriz2* y *matriz3*, declaradas en las líneas 14 y 15. Note que éstas tres matrices tiene la forma de la Figura 6.6.

Observe también que *matriz2* y *matriz3* han sido inicializadas en su declaración, esto quiere decir en tiempo de compilación le son asignados los

valores especificados. Los valores asignados están representados en la Figura 6.7.

Ahora bien, la línea 14 muestra un tipo de inicialización en secuencia, la cual es posible debido a la forma en que se representa la matriz en la memoria de la computadora (vea la Figura 6.6 (b)).

```

1  /* Programa que muestra la declaracion, el uso y
2     recorrido de arreglos de dos dimensiones(matrices).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define M 4
7  #define N 4
8
9  void leeMatriz(int [ ][N], int, int);
10 void imprimeMatriz(int [ ][N], int, int);
11
12 int main() {
13     int m, n;
14     int matriz1[M][N], matriz2[M][N] = {1, 2, 3, 4, 5},
15         matriz3[M][N] = {{1, 2, 3}, {3, 4}};
16
17     do{
18         printf("Dimensiones? ");
19         scanf("%d %d", &m, &n);
20     }while(m < 2 || n < 2 || m > M || n > N);
21     leeMatriz(matriz1, m, n);
22
23     printf("\tMatriz1:\n");
24     imprimeMatriz(matriz1, m, n);
25     printf("\tMatriz2:\n");
26     imprimeMatriz(matriz2, 2, 4);
27     printf("\tMatriz3:\n");
28     imprimeMatriz(matriz3, 2, 4);
29
30     return 0;
31 }
32
33 void leeMatriz(int matriz[ ][N], int m, int n){
34     int i, j;
35
36     for(i = 0; i < m; i++){
37         printf("Renglon %d/%d:\n", i, m - 1);
38         for(j = 0; j < n; j++){
39             printf("matriz[%d][%d] = ? ", i, j);
40             scanf("%d", &matriz[i][j]);
41         }
42         putchar('\n');
43     }
44 }
45
46 void imprimeMatriz(int matriz[ ][N], int m, int n){
47     int i, j;
48
49     for(i = 0; i < m; i++){

```

```

50     for(j = 0; j < n; j++)
51         printf("matriz[%d][%d] = %d ", i, j, matriz[i][j]);
52         putchar('\n');
53     }
54 }

```

Ejemplo 6.7: Uso de arreglos de dos dimensiones (matrices)

En base a lo anterior, los primeros cinco elementos físicos de *matriz2* tendrán los valores que se muestran en la Figura 6.7 (a). Los elementos que no son inicializados explícitamente, son puestos en 0 por omisión.

Por otro lado, la línea 15 muestra una inicialización por renglones para *matriz3*; éste tipo de inicialización se explica mejor con la Figura 6.7 (b), en donde se observa que los tres primeros elementos del renglón 0 están inicializados con los valores especificados (línea 15), mientras que el último de ellos ha sido inicializado por omisión con 0. Antes de continuar, asegúrese de entender lo que sucede con el segundo renglón de inicialización de la *matriz3*, y su correspondiente representación en la Figura 6.7 (b).

El ciclo **do-while** de las líneas 17-20 valida⁷ las dimensiones para la *matriz1* específicamente, debido a que el Ejemplo 6.7 define unas dimensiones físicas (líneas 14 y 15), pero controla unas dimensiones *lógicas* (líneas 21 y 24), y éstas dimensiones lógicas estarán determinadas por las variables *m* y *n* (línea 19), para el número de renglones y columnas respectivamente.

La Figura 6.7 indica también las dimensiones lógicas (indicadas con una elipse) para la *matriz2* y la *matriz3* respectivamente, mientras que las dimensiones físicas están representadas por la malla (matriz) de 4×4 .

La línea 21 hace el llamado a la función *leeMatriz* definida en las líneas 33-44. El llamado envía como argumentos la *matriz1*, y las dimensiones *m* y *n* obtenidas en la línea 19.

Ahora bien, la función *leeMatriz* define tres parámetros: *matriz* [][*N*], *m* y *n*. Observe que *matriz* será el nombre con el que la función hará referencia al arreglo bidimensional que reciba, y que la notación [][*N*] le indica al compilador que lo que recibirá será precisamente un arreglo bidimensional. Note también que los primeros corchetes están vacíos, mientras que los segundos tienen la dimensión física *N* declarada para el arreglo (líneas 14 y 15).

⁷La validación sólo considera matrices cuyas dimensiones sean de al menos 2×2 y de a lo más $M \times N$. Si bien es cierto que existen matrices de 3×1 o de 1×4 por ejemplo, también es cierto que técnicamente se les puede denominar *vectores*, por lo que por simplicidad, y para motivos de ilustración, se ha hecho éste tipo de validación.

	0	1	2	3
0	1	2	3	4
1	5			
2				
3				

(a) *matriz2*

	0	1	2	3
0	1	2	3	0
1	3	4	0	0
2				
3				

(b) *matriz3*

Figura 6.7: Inicialización de matrices (Ejemplo 6.7)

En C, los arreglos de más de una dimensión deben llevar siempre, con posibilidad de omitir únicamente el tamaño de la primera dimensión, el tamaño físico de cada una de las dimensiones restantes, y esto está estrechamente relacionado con la representación física en la memoria de la computadora de los arreglos de n dimensiones (ver Figura 6.6).

La especificación del tamaño de la segunda dimensión para las matrices, le indica al compilador cuántos elementos hay por renglón, y así poder determinar en dónde empiezan cada unos de los renglones de la matriz.

Continuando con el Ejemplo 6.7, la función *leeMatriz* se encarga de leer de la entrada estándar (línea 40) los elementos que conformarán la matriz lógica determinada por los parámetros m y n . Note cómo los ciclos **for** de las líneas 36 y 38 están en función de estos parámetros.

Por otro lado, las líneas 24, 26 y 28 hacen el llamado a la función *imprimeMatriz* definida en las líneas 33-44. El primer llamado envía como argumentos a la *matriz1* y a las dimensiones obtenidas en la línea 19; mientras que el segundo y tercer llamado envían a la *matriz2* y a la *matriz3* respectivamente, ambas con dimensiones lógicas 2 y 4.

Finalmente, la función *imprimeMatriz* definida en las líneas 46-54, define una lista de parámetros análoga a la de la función *leeMatriz*. La función *imprimeMatriz* realiza un recorrido por renglones de la matriz *matriz* para su impresión en la salida estándar. Una posible salida del Ejemplo 6.7 se muestra en la Figura 6.8.

```
Dimensiones? -1 4
Dimensiones? 2 56
Dimensiones? 3 2
Renglon 0/2:
matriz[0][0] = ? 1974
matriz[0][1] = ? 1978

Renglon 1/2:
matriz[1][0] = ? 2007
matriz[1][1] = ? 2008

Renglon 2/2:
matriz[2][0] = ? 4965
matriz[2][1] = ? 4460

    Matriz1:
matriz[0][0] = 1974 matriz[0][1] = 1978
matriz[1][0] = 2007 matriz[1][1] = 2008
matriz[2][0] = 4965 matriz[2][1] = 4460
    Matriz2:
matriz[0][0] = 1 matriz[0][1] = 2 matriz[0][2] = 3 matriz[0][3] = 4
matriz[1][0] = 5 matriz[1][1] = 0 matriz[1][2] = 0 matriz[1][3] = 0
    Matriz3:
matriz[0][0] = 1 matriz[0][1] = 2 matriz[0][2] = 3 matriz[0][3] = 0
matriz[1][0] = 3 matriz[1][1] = 4 matriz[1][2] = 0 matriz[1][3] = 0
```

Figura 6.8: Salida del Ejemplo 6.7

6.2.3. Suma de matrices

La suma de matrices es una operación clave del álgebra de matrices, y es muy sencilla debido a que la suma se realiza elemento a elemento entre las matrices que conforman los operandos.

Sean A y B dos matrices definidas de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad B = \begin{pmatrix} b_{0,0} & \dots & b_{0,n-1} \\ \vdots & \ddots & \vdots \\ b_{m-1,0} & \dots & b_{m-1,n-1} \end{pmatrix}_{m \times n} \quad (6.3)$$

La suma de las matrices 6.3, denotada por: $C_{m \times n}$, se define como:

$$C_{ij} = \sum_{j=0}^{n-1} a_{ij} b_{ij} \forall i, i = 0, \dots, m-1 \quad (6.4)$$

es decir:

$$\begin{pmatrix} a_{0,0} + b_{0,0} & \dots & a_{0,n-1} + b_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} + b_{m-1,0} & \dots & a_{m-1,n-1} + b_{m-1,n-1} \end{pmatrix}_{m \times n} \quad (6.5)$$

El Ejemplo 6.8 muestra la función *suma* que realiza la suma de las matrices a y b , y almacena el resultado en la matriz c . Observe que las matrices a y b tienen la forma descrita en la expresión 6.3, y que la matriz c que almacena la suma, se calcula siguiendo los lineamientos expuestos en las Ecuaciones 6.4 y 6.5.

```

1  /* Biblioteca de funciones que implementa las operaciones con matrices.
2  Las funciones asumen que el programa que las incluya definen la
3  constante simbolica N.
4  @autor Ricardo Ruiz Rodriguez
5  */
6
7  void suma(float a[ ][N], float b[ ][N], float c[ ][N], int m, int n){
8      int i, j;
9
10     for(i = 0; i < m; i++)
11         for(j = 0; j < n; j++)
12             c[i][j] = a[i][j] + b[i][j];
13 }

```

Ejemplo 6.8: Biblioteca de funciones para operaciones con matrices

Quizá el Ejemplo 6.8 no luzca todavía como una biblioteca de funciones, la intención es que progresivamente se vaya conformando y robusteciendo dicha biblioteca con más funciones, mismas que se proponen como ejercicios para el lector en la sección correspondiente. El Ejemplo 6.8 sólo ha dado el paso inicial.

Por otro lado, se propone como ejercicio para el lector el escribir un programa que pruebe la función *suma* del Ejemplo 6.8. Asegúrese de declarar tres matrices, colocar datos en dos de ellas (los operandos), y enviárselos a la función *suma*.

6.3. Arreglos de n dimensiones

En base a lo descrito en las dos secciones anteriores, es posible realizar una generalización para declarar y manipular arreglos n -dimensionales o de n dimensiones, ya que:

- Un **arreglo** es una sucesión contigua de variables de un mismo tipo de datos.
- Un **arreglo de dos dimensiones** o **matriz**, es un arreglo de arreglos.
- Un **arreglo de tres dimensiones** o **cubo**⁸, es un arreglo de arreglos de dos dimensiones, o un arreglo de matrices.
- Un **arreglo de cuatro dimensiones** o **hipercubo**⁹, es un arreglo de arreglos de tres dimensiones, o un arreglo de cubos para el caso de los hipercubos.
- Un arreglo de **n dimensiones**, es un arreglo de $n-1$ dimensiones.

En general, la declaración de un arreglo de n dimensiones, tiene la siguiente estructura en el lenguaje de programación C:

```
tipo_de_dato arreglo_n_dimensional [TAMAÑO1]
                                     [TAMAÑO2]
                                     ...
                                     [TAMAÑO_N] ;
```

⁸Si las tres dimensiones son iguales.

⁹Si las cuatro dimensiones son iguales.

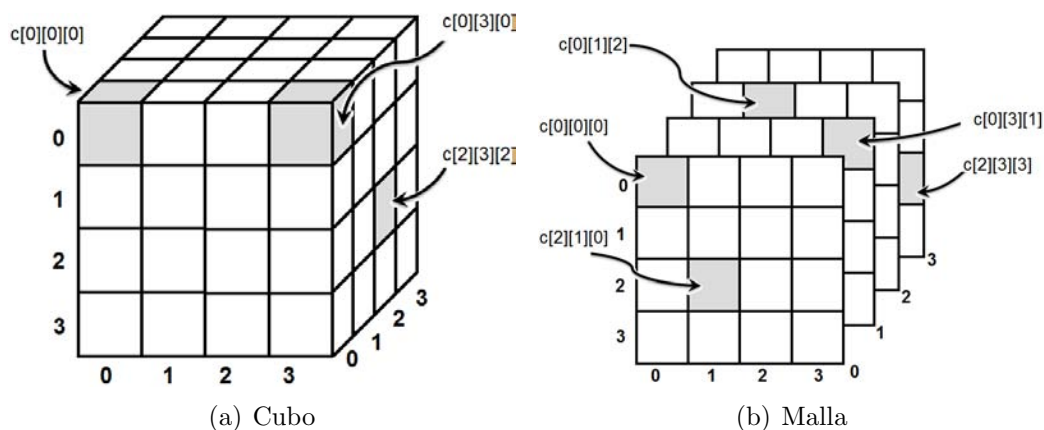


Figura 6.9: Representación de arreglos de tres dimensiones

en donde:

- **tipo_de_dato** es cualquier tipo de dato válido en C.
- **arreglo_n_dimensional** es un identificador válido en C.
- **TAMAÑO1** es la longitud para la primera dimensión.
- **TAMAÑO2** es la longitud para la segunda dimensión.
- **TAMAÑO_N** es la longitud para la *n-ésima* dimensión.

La Figura 6.9 muestra dos posibles representaciones para un arreglo *c* de tres dimensiones de $4 \times 4 \times 4$. La Figura 6.9 (a) muestra dicho arreglo representado como un cubo; mientras que la Figura 6.9 (b) muestra el mismo arreglo como una representación de malla, o arreglo de matrices. Asegúrese de entender la ubicación de los elementos del arreglo tridimensional en ambas representaciones.

El Ejemplo 6.9 muestra cómo representar en C (línea 11) la abstracción ilustrada en la Figura 6.9.

Observe cómo para la función *imprimeCubo*, en la línea 24 se especifica el tamaño de la segunda y tercera dimensión del arreglo *c*, ya que para arreglos de *n* dimensiones en general, se debe especificar el tamaño de todas las *n-1* dimensiones a partir de la segunda, la única opcional es la primera.


```

1  /* Programa que muestra la declaracion, el uso y
2     recorrido de arreglos de tres dimensiones (cubos).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define SIZE 4
7
8  void imprimeCubo(int [][SIZE][SIZE]);
9
10 main(){
11     int cubo[SIZE][SIZE][SIZE];
12     int i, j, k;
13
14     for(i = 0; i < SIZE; i++)
15         for(j = 0; j < SIZE; j++)
16             for(k = 0; k < SIZE; k++)
17                 cubo[j][k][i] = i + j + k;
18
19     imprimeCubo(cubo);
20
21     return 0;
22 }
23
24 void imprimeCubo(int c[][SIZE][SIZE]){
25     int i, j, k;
26
27     for(i = 0; i < SIZE; i++){
28         printf("Matriz %d/%d\n", i, SIZE - 1);
29         for(j = 0; j < SIZE; j++){
30             for(k = 0; k < SIZE; k++)
31                 printf("Cubo[%d][%d][%d] = %d ", j, k, i, c[j][k][i]);
32             putchar('\n');
33         }
34         putchar('\n');
35     }
36 }

```

Ejemplo 6.9: Uso de arreglos de tres dimensiones (cubos)

Note cómo por cada dimensión se tiene un ciclo que controla el recorrido de la misma, y por consiguiente, cada ciclo tiene su propia variable de control. En este orden de ideas, un arreglo de cuatro dimensiones requerirá de cuatro ciclos y de cuatro variables de control, un arreglo de cinco dimensiones cinco ciclos y cinco variables de control, y así sucesivamente.

Finalmente, es importante hacer notar que aunque el Ejemplo 6.9 presenta las tres dimensiones iguales para el arreglo *cubo*, esto no tiene por qué ser siempre así, de hecho, cada una de las tres dimensiones que definen un arreglo tridimensional pueden ser distintas.

La salida del Ejemplo 6.9 se muestra en la Figura 6.10. Note cómo la impresión de *cubo* se ha hecho en la forma de un arreglo de matrices. Apóyese

```

Matriz 0/3
Cubo[0][0][0] = 0 Cubo[0][1][0] = 1 Cubo[0][2][0] = 2 Cubo[0][3][0] = 3
Cubo[1][0][0] = 1 Cubo[1][1][0] = 2 Cubo[1][2][0] = 3 Cubo[1][3][0] = 4
Cubo[2][0][0] = 2 Cubo[2][1][0] = 3 Cubo[2][2][0] = 4 Cubo[2][3][0] = 5
Cubo[3][0][0] = 3 Cubo[3][1][0] = 4 Cubo[3][2][0] = 5 Cubo[3][3][0] = 6

Matriz 1/3
Cubo[0][0][1] = 1 Cubo[0][1][1] = 2 Cubo[0][2][1] = 3 Cubo[0][3][1] = 4
Cubo[1][0][1] = 2 Cubo[1][1][1] = 3 Cubo[1][2][1] = 4 Cubo[1][3][1] = 5
Cubo[2][0][1] = 3 Cubo[2][1][1] = 4 Cubo[2][2][1] = 5 Cubo[2][3][1] = 6
Cubo[3][0][1] = 4 Cubo[3][1][1] = 5 Cubo[3][2][1] = 6 Cubo[3][3][1] = 7

Matriz 2/3
Cubo[0][0][2] = 2 Cubo[0][1][2] = 3 Cubo[0][2][2] = 4 Cubo[0][3][2] = 5
Cubo[1][0][2] = 3 Cubo[1][1][2] = 4 Cubo[1][2][2] = 5 Cubo[1][3][2] = 6
Cubo[2][0][2] = 4 Cubo[2][1][2] = 5 Cubo[2][2][2] = 6 Cubo[2][3][2] = 7
Cubo[3][0][2] = 5 Cubo[3][1][2] = 6 Cubo[3][2][2] = 7 Cubo[3][3][2] = 8

Matriz 3/3
Cubo[0][0][3] = 3 Cubo[0][1][3] = 4 Cubo[0][2][3] = 5 Cubo[0][3][3] = 6
Cubo[1][0][3] = 4 Cubo[1][1][3] = 5 Cubo[1][2][3] = 6 Cubo[1][3][3] = 7
Cubo[2][0][3] = 5 Cubo[2][1][3] = 6 Cubo[2][2][3] = 7 Cubo[2][3][3] = 8
Cubo[3][0][3] = 6 Cubo[3][1][3] = 7 Cubo[3][2][3] = 8 Cubo[3][3][3] = 9

```

Figura 6.10: Salida del Ejemplo 6.9

de la Figura 6.9 (b) para visualizarlo mejor.

6.4. Consideraciones finales con arreglos

Probablemente en la Sección 6.2.3 se haya preguntado, por qué la función *suma* del Ejemplo 6.8 recibe también como parámetro a la matriz resultado *c*, ¿acaso no sería más conveniente y de mejor diseño regresar dicha matriz como valor de retorno de la función?.

Considere el Ejemplo 6.10, el cual propone, en base a los conceptos hasta ahora discutidos, una posible solución al planteamiento de la pregunta anterior.

La línea 8 presenta una forma intuitiva en la que se expresaría el deseo de que la función regrese una matriz; note que la lista de parámetros se ha reducido en uno respecto de la función *suma* del Ejemplo 6.8. Tómese el tiempo necesario para comparar ambas funciones antes de continuar.

Con la excepción del cambio expresado en el párrafo anterior, las funciones de los Ejemplos 6.8 y 6.10 son esencialmente la misma. El Ejemplo 6.10 no

compila, ya que no es posible indicarle al compilador de C que regrese una matriz como el tipo de dato de retorno para una función.

```
1  /* Ejemplo de funcion que intenta regresar una matriz como valor
2     de retorno para la funcion.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #define M 10
6  #define N 10
7
8  float [][][N] suma(float a[ ][N], float b[ ][N], int m, int n){
9      float c[M][N];
10     int i, j;
11
12     for(i = 0; i < m; i++)
13         for(j = 0; j < n; j++)
14             c[i][j] = a[i][j] + b[i][j];
15
16     return c;
17 }
```

Ejemplo 6.10: Matriz como valor de retorno

¿Por qué no se puede indicarle al compilador de C que regrese un arreglo?, además de que la gramática del lenguaje C no considera válida dicha expresión, se debe también a que las variables declaradas dentro de una función son variables automáticas y locales a ella, por lo tanto, sólo existen en el contexto de la propia función, por lo que cuando ésta termina, los datos son desechados antes de regresar el control de ejecución a la siguiente sentencia después de la invocación de la función, por lo que no tendría sentido regresar como valor de retorno una variable que ya fue desechada. Por otro lado, aún cuando dicha variable fuera afectada por el modificador **static** (véase la Sección 4.3), la variable sólo sería accesible dentro del contexto de la función.

Finalmente, la respuesta a la pregunta expresada al inicio de la sección es, que en general sería más conveniente y también un mejor diseño el enfoque del Ejemplo 6.10, pero en el lenguaje de programación C, los arreglos no se pueden regresar como valor de retorno de función, al menos no con los arreglos declarados en tiempo de compilación y con las herramientas hasta ahora estudiadas, hace falta algo más, y ese algo más son los *apuntadores*, los cuales se estudiarán en el Capítulo 7.

6.5. Ejercicios

1. Modifique el Ejemplo 6.1 de tal forma que imprima los 15 elementos del arreglo *arreglo* antes del ciclo **for** de la línea 13, con la finalidad de verificar que los elementos no inicializados explícitamente, son implícitamente inicializados a 0, tal y como se describe en el texto.

2. Para el Ejemplo 6.1 suponga que N es 5 (línea 7) y que los inicializadores (línea 10) son:

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

¿Qué sucede si existen más inicializadores que número de elementos para el arreglo?. Haga un programa que compruebe su respuesta.

3. Escriba un programa que defina y pruebe una función con la siguiente firma:

void rellenaArreglo(int a[], int n, int rango)

La función deberá inicializar los n elementos del arreglo a , con números **aleatorios** definidos entre 0 y $rango - 1$.

4. Escriba un programa que defina y pruebe una función con la siguiente firma:

void inicializaArreglo(int a[], int n, int valor)

La función deberá inicializar los n elementos del arreglo a , con el valor *valor*.

5. Escriba un programa que defina y pruebe una función con la siguiente firma:

void imprimeArreglo(const int a[], int n, int c)

La función deberá imprimir en la salida estándar, los n elementos del arreglo a , con c elementos por renglón.

6. Escriba un programa que defina y pruebe una función con la siguiente firma:

void copiaArreglo(const int original[], int copia[], int n)

La función deberá copiar los n elementos del arreglo *original*, en el arreglo *copia*.

7. Escriba un programa que mida el tiempo en segundos para cada una de las funciones del Ejemplo 6.4. Su programa deberá:

- Declarar dos arreglos de 32,767 elementos cada uno.
- Utilizar una función para inicializar uno de ellos con números aleatorios.
- Usar una función para copiar el arreglo inicializado en el segundo arreglo declarado.
- Enviar uno de los arreglos a la función *burbuja* y medir el tiempo que le toma ordenar los elementos.
- Enviar el otro arreglo a la función *burbujaMejorado* y medir el tiempo que le toma ordenar los elementos.

La intención de este ejercicio es determinar si existe o no una mejor eficiencia con las modificaciones descritas en el texto, ya que el algoritmo de ordenamiento no cambia, sólo se ha hecho, en principio, más eficiente.

8. Escriba un programa que pruebe las funciones del Ejemplo 6.5. Pruebe con datos leídos desde el teclado, con datos aleatorios, y verifique que las funciones de búsqueda trabajan de acuerdo a lo esperado.
Nota: No olvide que la búsqueda binaria tiene como pre requisito que el conjunto de datos esté **ordenado**.

9. Realice una prueba de escritorio para la función *busquedaBinaria* del Ejemplo 6.5 para que busque las siguientes claves: -4, 22, 1 y 13. Utilice el siguiente conjunto de búsqueda:

-20, -4, 0, -9, 7, 10, 15, 22

10. Realice una implementación recursiva de la búsqueda binaria en base a la descripción hecha en el texto respecto a la búsqueda binaria, y una vez que haya realizado las pruebas de escritorio propuestas en el ejercicio anterior.
11. Siguiendo las ideas expuestas en la Sección 6.1.6, escriba un programa que defina y pruebe una función con la siguiente firma:

void copiaCadena(const char original[], char copia[])

La función deberá copiar los elementos de la cadena *original*, en la cadena *copia*. Puede apoyarse de las funciones desarrolladas hasta el momento.

12. Siguiendo las ideas expuestas en la Sección 6.1.6, escriba un programa que defina y pruebe una función con la siguiente firma:

void invierteCadena(const char original[], char invertida[])

La función deberá copiar **invertidos** los elementos de la cadena *original*, en la cadena *invertida*. Puede apoyarse de las funciones desarrolladas hasta el momento.

13. Un palíndromo es una expresión que representa o expresa lo mismo si se lee de izquierda a derecha que de derecha a izquierda, como por ejemplo:

- Madam.
- 2002.
- Anita lava la tina.
- Dabale arroz a la zorra el abad.

entre cientos de palíndromos más.

Siguiendo las ideas expuestas en la sección 6.1.6, escriba un programa que defina y pruebe una función con la siguiente firma:

int esPalindromo(const char c[])

La función deberá determinar si los elementos de la cadena *c*, constituyen (regresa 1) o no (regresa 0) un palíndromo. Puede apoyarse de las funciones desarrolladas hasta el momento, y por ahora, asuma que los elementos de la cadena no contendrán espacios y que todos los caracteres serán introducidos en minúsculas.

14. Escriba un programa que obtenga la CURP (Clave Única de Registro de Población). La CURP se obtiene de la siguiente forma:

- Primera letra y primera vocal del primer apellido.
- Primera letra del segundo apellido.
- Primera letra del nombre de la persona, en casos como José o María, de ser posible, se tomará la primera letra del segundo nombre.
- Últimas dos cifras del año de nacimiento.
- Dos cifras del mes de nacimiento.

- Dos cifras del día de nacimiento .
- Sexo de la persona: H (Hombre) o M (Mujer).
- Dos Letras, correspondientes a la entidad de nacimiento. La entidad de nacimiento se obtiene como la primera letra de la entidad y la última consonante. En caso de extranjeros es NE (Nacido en el Extranjero).
- Consonante del primer apellido (la primera si el primer apellido empieza con vocal, la segunda si empieza en consonante).
- Consonante del segundo apellido (la primera si el primer apellido empieza con vocal, la segunda si empieza en consonante).
- Consonante del nombre de la persona (la primera si el primer apellido empieza con vocal, la segunda si empieza en consonante).
- Dos dígitos para evitar duplicidad (genérelos aleatoriamente).
- En casos en los que las personas no tienen segundo apellido, o que la letra que corresponde insertar sea la letra Ñ, se utilizará en su lugar la X.

Verifique estas reglas para su propia CURP.

15. Considere el Ejemplo 6.7. ¿Qué pasa si hay más inicializadores que elementos físicos? Pruebe ésta situación en los dos esquemas de inicialización planteados en el texto.
16. Escriba un programa que pruebe la función *suma* del Ejemplo 6.8.
17. Modifique la función *leeMatriz* del Ejemplo 6.7 para que almacene valores de tipo **float** en la matriz *matriz*. Una vez que haya probado su función, agréguela a la biblioteca de funciones del Ejemplo 6.8.
18. Modifique la función *imprimeMatriz* del Ejemplo 6.7 para que imprima los valores de tipo **float** de la matriz *matriz*. Una vez que haya probado su función, agréguela a la biblioteca de funciones del Ejemplo 6.8.
19. Escriba un programa que defina y pruebe una función con la siguiente firma:

```
void resta(float a[ ][N], float b[ ][N], float c[ ][N],  
           int m, int n)
```

La función deberá calcular en la matriz c la resta de la matriz a menos la matriz b , de manera análoga a como lo hace la función *suma* del Ejemplo 6.8.

20. Agregue la función *resta* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 6.8.

21. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad (6.6)$$

La matriz transpuesta de A , denotada por A^T , está definida como:

$$(A^T)_{ij} = A_{ji}, 0 \leq i \leq n-1, 0 \leq j \leq m-1 \quad (6.7)$$

es decir:

$$\begin{pmatrix} a_{0,0} & \dots & a_{m-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,n-1} & \dots & a_{m-1,n-1} \end{pmatrix}_{n \times m} \quad (6.8)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

void imprimeTranspuesta(float a[][N], int m, int n)

La función deberá **imprimir** en la salida estándar, la matriz transpuesta de a , siguiendo los lineamientos de las Expresiones 6.7 y 6.8.

22. Escriba un programa que defina y pruebe una función con la siguiente firma:

void transpuesta(float a[][N], float aT[][N], int m, int n)

La función deberá **generar** en la matriz aT , la matriz transpuesta de a , siguiendo los lineamientos de las Expresiones 6.7 y 6.8.

23. Agregue la función *transpuesta* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 6.8.

24. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{n \times n} \quad (6.9)$$

Se dice que la matriz A es simétrica respecto a la diagonal principal si:

$$a_{ij} = a_{ji}, \forall i \neq j \quad (6.10)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

int esSimetrica(float a[][N], int n)

La función deberá determinar si la matriz a , es o no simétrica en base a lo expuesto en la Expresión 6.10.

25. Agregue la función *esSimetrica* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 6.8.
26. Sean A y B dos matrices definidas de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad B = \begin{pmatrix} b_{0,0} & \dots & b_{0,l-1} \\ \vdots & \ddots & \vdots \\ b_{n-1,0} & \dots & b_{n-1,l-1} \end{pmatrix}_{n \times l} \quad (6.11)$$

El producto de $A_{m \times n} \times B_{n \times l}$ denotado por $C_{m \times l}$, se define como:

$$C_{ij} = \sum_{r=0}^{n-1} a_{ir} b_{rj} \quad (6.12)$$

es decir:

$$\begin{pmatrix} a_{0,0}b_{0,0} + \dots + a_{0,n-1}b_{n-1,0} & \dots & a_{0,0}b_{0,l-1} + \dots + a_{0,n-1}b_{n-1,l-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}b_{0,0} + \dots + a_{m-1,n-1}b_{n-1,0} & \dots & a_{m-1,0}b_{0,l-1} + \dots + a_{m-1,n-1}b_{n-1,l-1} \end{pmatrix}_{m \times l} \quad (6.13)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

**void producto(float a[][N], float b[][N], float c[][N],
int m, int n, int l)**

de tal forma que la función calcule el producto de las matrices a y b almacenando el resultado en c . Las matrices tienen la forma descrita en la Expresión 6.11; y el producto se calcula siguiendo los lineamientos expuestos en las Expresiones 6.12 y 6.13.

27. Agregue la función *producto* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 6.8.
28. La Figura 6.9 muestra en gris algunos de los elementos del arreglo de tres dimensiones. Denote todos los elementos visibles (para ambas representaciones (a) y (b)) de dicho arreglo siguiendo la nomenclatura del lenguaje C.
29. De manera análoga a lo descrito con anterioridad para los arreglos y las matrices, dibuje y describa la representación física en la memoria de La Figura 6.9.

Sugerencia: la representación en memoria se realiza también de manera lineal: los renglones de la primera matriz, después los de la segunda matriz, etcétera, la representación de malla puede ayudarle a visualizar mejor la representación.

30. Basándose en el Ejemplo 6.9, escriba un programa que generalice la idea planteada para definir, inicializar, e imprimir en la salida estándar un hipercubo.
31. Basándose en el Ejemplo 6.9, escriba un programa que generalice la idea planteada para definir, inicializar, e imprimir en la salida estándar un arreglo de cinco dimensiones, cada una con un tamaño distinto, por ejemplo: dos para la primera dimensión, tres para la segunda, cuatro para la tercera y así sucesivamente.

Capítulo 7

Apuntadores

Los apuntadores son uno de los tabúes del lenguaje de programación C¹.

Este capítulo introduce a los conceptos y manipulación de apuntadores, la principal intención es que, al final de su lectura, los apuntadores dejen de ser un tabú y pasen a ser parte del repertorio de herramientas del programador, ya que los apuntadores son, en más de un sentido, la piedra angular del lenguaje de programación C.

7.1. Definición, estructura y representación

La sencillez de los apuntadores se basa en su definición, la cual es bastante fácil de recordar y es, en esencia, casi todo lo que se tiene que saber respecto a los apuntadores, de ahí la importancia de su comprensión.

Un **apuntador** es una variable cuyo contenido es la dirección en memoria de otra variable.

La Figura 7.1 muestra la representación de un apuntador, y su explicación se verá reforzada cuando se analice el Ejemplo 7.1.

Mientras tanto, tome en cuenta que un apuntador sólo puede *hacer referencia* o *apuntar*, a objetos de su mismo tipo de datos, esto es: un apuntador a **int**, sólo puede hacer referencia a variables de tipo **int**, un apuntador a **char**, sólo puede hacer referencia a variables de tipo **char**, etc.

En este sentido, la declaración de una variable de tipo apuntador tiene la siguiente estructura general en el lenguaje de programación C:

¹Otro tabú es la recursividad, aunque la recursividad no está asociada a ningún lenguaje, sino con la programación.

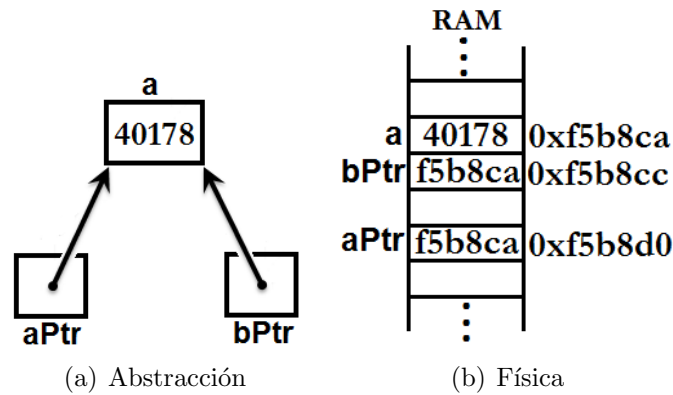


Figura 7.1: Representación de un puntero

```
tipo_de_dato * nombre_del_apuntador;
```

en donde:

- **tipo_de_dato** es cualquier tipo de dato válido en C.
- **nombre_del_apuntador** es un identificador válido en C.
- ***** es el operador que denota que la variable **nombre_del_apuntador** es una variable de tipo puntero a **tipo_de_dato**.

7.1.1. Piedra angular de la piedra angular

El Ejemplo 7.1 es *fundamental* para entender a los punteros, es indispensable su total y absoluta comprensión para poder continuar y construir, en base a él, los conceptos subsiguientes.

La línea 9 muestra la declaración de dos variables de tipo puntero a entero: *aPtr* y *bPtr*². Observe que el operador “*” no se distribuye en las variables como lo hace el tipo de dato, por lo que por cada variable de tipo puntero que se necesite, se deberá especificar dicho operador, para especificar que la variable asociada es de tipo puntero.

²Note que como parte del identificador para cada una de las variables, se ha utilizado la terminación *Ptr* (*Pointer*) a forma de sufijo, esto no es obligatorio, ni por sí mismo vuelve a la variable de tipo puntero, pero es una buena práctica de programación el hacerlo así, debido a que refuerza, por auto documentación, que la variable es de tipo puntero, ya que después de la declaración, no hay nada particular en la variable que en sí misma denote, que es o no, de tipo puntero.

```

1  /* Programa que muestra la declaracion de un apuntador a entero,
2     el uso del operador de direccion (&) y de desreferencia (*).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int main() {
8      int a;
9      int *aPtr, *bPtr;
10
11     a = 40178;
12     aPtr = &a; /* aPtr apunta a la variable a */
13     bPtr = &a; /* bPtr guarda la direccion de la variable a */
14
15     printf("Direccion de a: %p\n", &a);
16     printf("Valor de aPtr: %p\n", aPtr);
17
18     printf("\nValor de a: %d\n", a);
19     printf("Valor de *aPtr (desreferencia): %d\n", *aPtr);
20
21     *bPtr = 2018;
22     printf("\nValor de a: %d\n", a);
23
24     return 0;
25 }

```

Ejemplo 7.1: Uso de apuntadores

Las líneas 11, 12 y 13 inicializan a la variable *a* con *40178*, y a *aPtr* y *bPtr*, con la *dirección en memoria* de *a*. El operador “&”, obtiene la dirección en memoria del operando (variable) asociado(a), y se conoce como el **operador de dirección**.

Una variable de tipo apuntador, al ser una variable cuyo contenido es la dirección en memoria de otra variable, almacena direcciones de memoria obtenidas por el operador de dirección³, y aunque en principio es posible asignarle a un apuntador un valor específico, esto generará, con toda seguridad, una violación de memoria, y el programa será terminado por cualquier sistema operativo decente⁴. La Figura 7.1 ilustra lo que ocurre en las líneas 11-13.

Las funciones **printf** de las líneas 15 y 16 contienen el especificador de formato “%p”(*pointer*), el cual permite visualizar en la salida estándar direcciones de memoria. Note cómo la dirección de memoria que se imprime en

³El único valor que puede asignarse explícitamente a una variable de tipo apuntador es el 0, denotando así que el apuntador no hace referencia a nada, es decir que no apunta a ninguna dirección válida de memoria, y se dice que es un apuntador *nulo*.

⁴Ningún sistema operativo debería permitir la intromisión en áreas de memoria que no sean las propias del programa, ya que esto incurriría en graves errores de seguridad.

```

Direccion de a: 0x7fffdb32630c
Valor de aPtr: 0x7fffdb32630c

Valor de a: 40178
Valor de *aPtr (desreferencia): 40178

Valor de a: 2018

```

Figura 7.2: Salida del Ejemplo 7.1

estas líneas es la misma, tal y como lo muestra la Figura 7.2.

La línea 19 muestra el uso del **operador de des referencia** “*”, el cual se utiliza para acceder al valor de la variable a la que se apunta, o se hace referencia a través del apuntador.

Para variables de tipo apuntador, el operador “*” tiene una dualidad:

1. **En la declaración de variables:** se utiliza o sirve para denotar que la variable asociada es de tipo apuntador.
2. **En la des referencia de variables:** se utiliza o sirve para acceder al valor al que hace referencia o apunta el apuntador.

Finalmente, la línea 21 modifica el valor de la variable *a* a través del apuntador *bptr*⁵, y del operador de des referencia, por lo que el valor a imprimir en la salida estándar para la variable *a*, será *2018*, tal y como lo muestra la Figura 7.2.

Antes de continuar, asegúrese de entender en su totalidad el Ejemplo 7.1 y lo que se ha descrito de él, ya que es *fundamental* para la comprensión de los conceptos descritos en las secciones siguientes.

7.2. Parámetros por valor y por referencia

En el lenguaje de programación C, cuando se hace el llamado a una función y se le envían argumentos, dichos argumentos son *copiados* en los parámetros de la función; es decir, por cada argumento existe una copia almacenada en el parámetro correspondiente. A éste esquema de **copia** de valores se le conoce en el *argot* computacional como **parámetros por valor**.

⁵Recuerde que la variable *a* está siendo referida (apuntada) por las variables *aPtr* y *bPtr* (líneas 12 y 13 respectivamente).

Los parámetros por valor son el esquema estándar de C, pero existe otro enfoque para el envío de argumentos y su respectiva recepción como parámetros. En dicho enfoque, no hay una copia de valores, sino una **referencia** a los valores originales, y la forma de implementarlo es a través de apuntadores. Cuando se utiliza éste esquema se habla de **parámetros por referencia**.

El Ejemplo 7.2 muestra el uso, funcionamiento, y diferencia de los parámetros por valor y los parámetros por referencia.

```
1  /* Programa que muestra la diferencia entre un parametro
2     por valor y uno por referencia (apuntador).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  void llamadaValor(int);
8  void llamadaReferencia(int *);
9
10 int main(){
11     int argumento = 40178;
12
13     printf("Valor de argumento (main()): %d\n", argumento);
14     llamadaValor(argumento); /* parametro = argumento */
15     printf("\nValor de argumento (main()): %d\n", argumento);
16     llamadaReferencia(&argumento); /* parametro = &argumento */
17     printf("\nValor de argumento (main()): %d\n", argumento);
18
19     return 0;
20 }
21
22 void llamadaValor(int parametro){
23     printf("\nValor de argumento (llamadaValor()): %d\n", parametro);
24     parametro *= 2;
25     printf("Valor de argumento (llamadaValor()): %d\n", parametro);
26 }
27
28 void llamadaReferencia(int *parametro){
29     printf("\nValor de argumento (llamadaReferencia()): %d\n", *parametro);
30     *parametro *= 2; /* *parametro = *parametro * 2; */
31     printf("Valor de argumento (llamadaReferencia()): %d\n", *parametro);
32 }
```

Ejemplo 7.2: Parámetros por valor y por referencia

La principal diferencia se observa en los prototipos de las funciones.

La línea 7 indica que se definirá una función de nombre *llamadaValor* que no regresará nada, y que recibirá un **entero**, es decir, un **parámetro por valor**.

Por otro lado, la línea 8 indica que se definirá una función de nombre *llamadaReferencia* que no regresará nada y que recibirá un **apuntador a entero**, es decir, un **parámetro por referencia**.

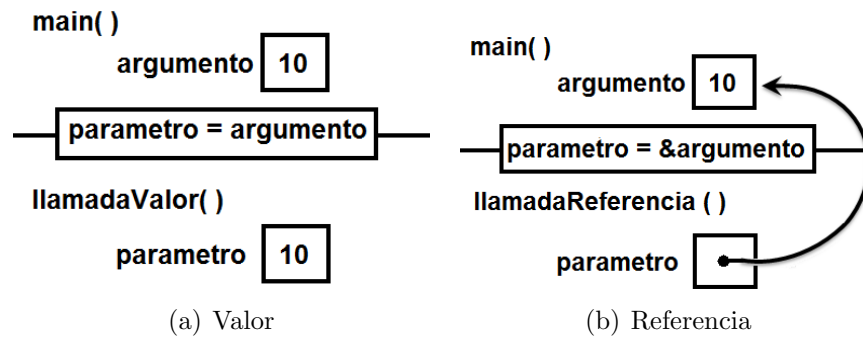


Figura 7.3: Representación de parámetros por valor y por referencia

La función **main** define e inicializa en la línea 11, a la variable entera *argumento*, misma que será enviada a las funciones *llamadaValor* y *llamadaReferencia* en las líneas 14 y 16 respectivamente. Así mismo, se imprime en la salida estándar el valor de *argumento* en las líneas 13, 15 y 17, para determinar si éste ha sido afectado por los llamados de función correspondientes.

Sólo resta explicar las funciones:

- *llamadaValor*: recibe en *parametro* una **copia** de *argumento* justo en el llamado de la función (línea 14). La Figura 7.3 (a) muestra los contextos de las funciones **main** y *llamadaValor* separados por una línea y un recuadro, el cual contiene la asignación implícita que ocurre en el llamado de la función.
- *llamadaReferencia*: recibe en *parametro* la **referencia** a *argumento* justo en el llamado de la función (línea 16)⁶. La Figura 7.3 (b) muestra los contextos de las funciones **main** y *llamadaReferencia* separados por una línea y un recuadro, el cual contiene la asignación implícita que ocurre en el llamado de la función.

La salida del Ejemplo 7.2 se muestra en la Figura 7.4.

7.3. Aritmética de apuntadores

Las variables de tipo apuntador son variables que almacenan direcciones en memoria de otras variables, pero son al fin y al cabo variables, por lo

⁶Note que en la línea 16 se está enviando la **referencia** o **dirección en memoria** de *argumento* a *parametro*.


```
Valor de argumento (main()): 40178
Valor de argumento (llamadaValor()): 40178
Valor de argumento (llamadaValor()): 80356

Valor de argumento (main()): 40178
Valor de argumento (llamadaReferencia()): 40178
Valor de argumento (llamadaReferencia()): 80356

Valor de argumento (main()): 80356
```

Figura 7.4: Salida del Ejemplo 7.2

que es posible realizar algunas operaciones aritméticas sobre los valores que almacenan.

Las operaciones válidas de apuntadores son [Kernighan2]:

1. Asignación de apuntadores del mismo tipo.
2. Adición de un apuntador y un entero.
3. Sustracción de un apuntador y un entero.
4. Resta de dos apuntadores a miembros del mismo arreglo.
5. Comparación de dos apuntadores a miembros del mismo arreglo.
6. Asignación del valor cero.
7. Comparación con cero.

Cualquier otra aritmética de apuntadores es ilegal. No es legal sumar dos apuntadores, multiplicarlos o dividirlos, sumarles un valor **float**, o realizar asignaciones entre diferentes tipos de apuntadores sin una conversión de tipo forzada (*cast*).

Cuando se suma o resta un entero a un apuntador, dicho entero representa una *escala*, que está en función del tamaño en *bytes* del tipo de datos a los que apunta el apuntador.

Suponga que *ptr* es un apuntador a entero que hace referencia al primer elemento de un arreglo de enteros, y que *n* es también un entero, donde $0 < n < TAM^7$, entonces la expresión:

⁷TAM representa el tamaño del arreglo.

$$\begin{array}{c} \text{ptr} += \text{n}; \\ \text{ó} \\ \text{ptr} = \text{ptr} + \text{n}; \end{array}$$

indica al compilador que *ptr* se mueva o avance al *n*-ésimo elemento respecto de su posición actual.

En resumen, no se está realizando una suma convencional de una dirección de memoria con un entero, sino una *suma escalada* en función del tipo de datos. Así, si cada entero se almacena en cuatro *bytes* por ejemplo, a la dirección que almacena el apuntador se le asigna $\text{ptr} + (4 * n)$, y en general.

$$\text{ptr} = \text{ptr} + (\text{sizeof}(\text{tipo_de_dato}) * \text{n})$$

donde *tipo_de_dato* es el tipo de dato del elemento al que hace referencia el apuntador.

En la Sección 7.4 se hará uso de éste tipo de notación; por ahora, el Ejemplo 7.3 muestra el uso del modificador **const** con apuntadores, y el uso de aritmética de apuntadores para realizar recorridos de cadenas.

La línea 8 establece que se definirá una función de nombre *imprimeSinEspacios* que no regresará nada, y que recibirá un apuntador constante a **char**, lo cual significa que, dentro de la función, no se podrá modificar, a través del apuntador *cPtr* (línea 22), el elemento al que se haga referencia. La línea 9 se describe de manera análoga a la línea 8, pero para la función *imprimeAlReves*.

Por otro lado, la función **main** se encarga de definir un palíndromo y almacenarlo en una cadena (línea 12), imprimir dicha cadena (líneas 14 y 17), y enviársela a las funciones *imprimeAlReves* e *imprimeSinEspacios* (líneas 15 y 16 respectivamente).

La función *imprimeSinEspacios* (línea 22), realiza un recorrido de la cadena a través del apuntador *cPtr* y el ciclo **while**. Observe cómo el apuntador se va moviendo por los distintos elementos de la cadena (línea 26), haciendo referencia a cada uno de ellos uno a la vez (líneas 24 y 25). La función hace lo que su nombre indica: imprime sin espacios (línea 24) la cadena a la que hace referencia *cPtr* sin modificar la cadena.

Finalmente, la función *imprimeAlReves* (línea 31), realiza un recorrido de la cadena (línea 35) para llegar al final de ella, a través del apuntador *cPtr* y el ciclo **while**. Una vez que se llega al final de la cadena, se utiliza otro ciclo **while** (línea 36) para regresar, a través de *cPtr*, a la posición de inicio

que se respaldó en *inicioPtr* (línea 32). La función hace lo que su nombre indica: imprime al revés la cadena a la que hace referencia *cPtr* sin modificar la cadena.

```

1  /* Programa que muestra el uso del modificador const con apuntadores,
2     y el recorrido de cadenas por medio de apuntadores.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define ESPACIO_EN_BLANCO ' '
7
8  void imprimeSinEspacios(const char *);
9  void imprimeAlReves(const char *);
10
11 int main(){
12     char cadena[] = "Dabale arroz a la zorra el abad";
13
14     printf("Cadena original: %s\n", cadena);
15     imprimeAlReves(cadena); /* cPtr = &cadena[0] */
16     imprimeSinEspacios(cadena); /* cPtr = cadena */
17     printf("Cadena original: %s\n", cadena);
18
19     return 0;
20 }
21
22 void imprimeSinEspacios(const char *cPtr){
23     while(*cPtr != '\0'){ /* Mientras no hagas referencia a fin de cadena */
24         if(*cPtr != ESPACIO_EN_BLANCO)
25             putchar(*cPtr);
26         cPtr++; /* Avanza a la siguiente posicion de la cadena */
27     }
28     putchar('\n');
29 }
30
31 void imprimeAlReves(const char *cPtr){
32     const char *inicioPtr = cPtr; /* Respalda la direccion de inicio */
33
34     while(*cPtr != '\0') /* Mientras no hagas referencia a fin de cadena */
35         cPtr++; /* Avanza a la siguiente posicion de la cadena */
36     while(cPtr-- != inicioPtr) /* Mientras no llegues al inicio (línea 32) */
37         putchar(*cPtr);
38     putchar('\n');
39 }

```

Ejemplo 7.3: Recorrido de cadenas con apuntadores

El Ejemplo 7.3 hace uso de la aritmética de apuntadores más usual, note que se está incrementando (líneas 26 y 35) y decrementando (línea 36) el apuntador *cPtr*, las cuales son operaciones clásicas de recorridos de arreglos con apuntadores. La salida de dicho ejemplo se muestra en la Figura 7.5.

La biblioteca estándar de funciones *ctype.h* incluye un conjunto de funciones para clasificar y transformar caracteres individuales. Se recomienda la revisión de esta biblioteca en algún manual de referencia, aquí sólo se

```

Cadena original: Dabale arroz a la zorra el abad
daba le arroz al a zorra elabaD
Dabalearrozalazorraelabad
Cadena original: Dabale arroz a la zorra el abad

```

Figura 7.5: Salida del Ejemplo 7.3

presenta un breve resumen de algunas de las funciones que incorpora:

- **isalnum**: regresa *1* si su argumento es alfanumérico y *0* si no lo es.
- **isalpha**: regresa *1* si su argumento pertenece al alfabeto (inglés) y *0* si no pertenece.
- **isdigit**: regresa *1* si su argumento es un dígito decimal y *0* si no lo es.
- **isxdigit**: regresa *1* si su argumento es un dígito hexadecimal y *0* si no lo es.
- **islower**: regresa *1* si su argumento es una letra minúscula y *0* si no lo es.
- **isupper**: regresa *1* si su argumento es una letra mayúscula y *0* si no lo es.
- **ispunct**: regresa *1* si su argumento es un carácter de puntuación y *0* si no lo es.
- **tolower**: regresa su argumento convertido a una letra minúscula.
- **toupper**: regresa su argumento convertido a una letra mayúscula.

El Ejemplo 7.4 muestra el uso de las funciones *islower* y *toupper*, las otras funciones trabajan de manera análoga.

Respecto al Ejemplo 7.4, note que el símbolo de fin de cadena se ha colocado ahora como una constante simbólica (línea 8), misma que es utilizada por el ciclo **for** de la línea 24 como parte de su expresión condicional. En éste sentido, note también que ahora el recorrido de la cadena referida por *cPtr* se realiza con un ciclo **for** en lugar de un ciclo **while**, y que como el recorrido se realiza a través del apuntador *cPtr*, no es necesaria la expresión

```
Introduzca una cadena: Ricardo Ruiz, Cubo #40 Inst. Computacion
Su cadena en mayúsculas es: RICARDO RUIZ, CUBO #40 INST. COMPUTACION
```

Figura 7.6: Salida del Ejemplo 7.4

de inicialización de variables de control para el **for**, por lo que ésta aparece vacía.

La función *convierteAMayusculas* (líneas 23-27), verifica si el carácter al que hace referencia *cPtr* es una letra minúscula (línea 25), y si lo es, lo convierte a mayúscula y lo almacena en la misma posición en donde estaba dicho carácter (línea 26).

```
1  /* Ejemplo de uso de las funciones islower y toupper de
2     la biblioteca estandar de funciones ctype.h.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include <ctype.h>
7  #define TAM 100
8  #define FIN_DE_CADENA '\0'
9
10 void convierteAMayusculas(char *);
11
12 int main(){
13     char cadena[TAM];
14
15     printf("Introduzca una cadena: ");
16     gets(cadena);
17     convierteAMayusculas(cadena);
18     printf("Su cadena en mayusculas es: %s\n", cadena);
19
20     return 0;
21 }
22
23 void convierteAMayusculas(char *cPtr){
24     for( ; *cPtr != FIN_DE_CADENA; cPtr++)
25         if(islower(*cPtr))
26             *cPtr = toupper(*cPtr);
27 }
```

Ejemplo 7.4: Conversión de una cadena a mayúsculas

Una posible salida del Ejemplo 7.4 se muestra en la Figura 7.6. Note cómo solamente son afectados los caracteres en minúscula, todos los demás caracteres, incluyendo los espacios, son ignorados, es decir, se dejan sin modificación en la cadena que los contiene.

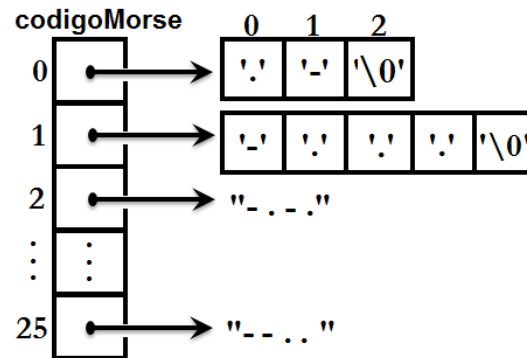


Figura 7.7: Representación del arreglo de punteros *codigoMorse* del Ejemplo 7.5

7.4. Apuntadores y arreglos

Los apuntadores son variables especiales pero variables finalmente, ¿será posible entonces crear un arreglo de apuntadores?. La respuesta es sí, de hecho, los arreglos de apuntadores son de lo más común en los programas en C.

El Ejemplo 7.5 muestra en la línea 9, un arreglo de apuntadores a **char** de nombre *codigoMorse*. Observe que dicho arreglo es inicializado en tiempo de compilación, y que cada elemento del arreglo hace referencia a una cadena o arreglo de caracteres.

La Figura 7.7 muestra la representación del arreglo de apuntadores definido en la línea 9 del Ejemplo 7.5. Note cómo cada elemento de *codigoMorse* hace referencia a cadenas de diferente longitud, correspondientes al código Morse de cada una de las letras del alfabeto inglés⁸. Observe también que en las primeras dos representaciones de la Figura 7.7⁹, se ha hecho explícito el almacenamiento de arreglo de caracteres para los apuntadores correspondientes, mientras que para las siguientes, la representación de cadenas se ha puesto entre comillas dobles.

El ciclo **for** de la línea 15 resulta peculiar, pero es un ejemplo clave del poder del lenguaje de programación C. Note cómo la variable de control *letra* es de tipo **char** y cómo *letra* es inicializada con el símbolo 'a' en la expresión

⁸Se ha hecho así para no tener que lidiar con aspectos irrelevantes a los arreglos de apuntadores, como el código de la letra “ñ” por ejemplo.

⁹Correspondientes a los índices 0 y 1 respectivamente del arreglo de apuntadores *codigoMorse*.

de inicialización del ciclo; *letra* se utilizará para recorrer el alfabeto (*letra++*), mientras no se llegue al símbolo 'z' (*letra <= 'z'*) como lo indica la expresión condicional.

```

1  /* Ejemplo de un arreglo de apuntadores a caracteres para
2     convertir alfabeto ingles a codigo Morse.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  int main() {
8      char letra;
9      const char *codigoMorse[] =
10         { ".-", "-...", "-.-.", "-..", ".", "-.-.", "--.", "...",
11           ". .", "-.-.", "-.-.", "-.-.", "--", "--", "--", "--",
12           "--.", "-.", "...", "-", "-.-", "-.-", "-.-", "-.-",
13           "-.-", "-.-." };
14
15      for (letra = 'a'; letra <= 'z'; letra++)
16          printf("'%c' = \"%s\"\\n\\t ", letra, codigoMorse[letra - 'a']);
17      putchar('\\n');
18
19      return 0;
20  }
```

Ejemplo 7.5: Arreglo de apuntadores a **char**

Lo último que resta por explicar es la expresión de la línea 16:

codigoMorse[letra - 'a']

letra es inicialmente 'a', así que:

$$\begin{aligned}
 &'a' - 'a' = 0 \\
 &'b' - 'a' = 1 \\
 &\quad \cdot \\
 &\quad \cdot \\
 &\quad \cdot \\
 &'z' - 'a' = 25
 \end{aligned}$$

Si comprendió la explicación quizá esté sorprendido y la respuesta a su posible interrogante es sí: C puede restar letras. En realidad está restando los códigos de los símbolos respectivos, pero es mejor dejarle los detalles de los códigos al compilador que tener que lidiar con los códigos de cada símbolo¹⁰.

La salida del Ejemplo 7.5 se muestra en la Figura 7.8.

¹⁰Esto, además de ser un mejor enfoque de abstracción que utilizar *números mágicos*, hace que el código sea más portable al no depender de códigos de tipo ASCII (*American Standard Code for Information Interchange*), UTF (*Unicode Transformation Format*), etc.

```

'a' = ".-"      'b' = "-..."  'c' = "-.-"    'd' = "-..."  'e' = ".-"
'f' = "...-"    'g' = "-.-"    'h' = "...-"   'i' = "...-"   'j' = "...-"
'k' = "...-"    'l' = "...-"   'm' = "...-"   'n' = "...-"   'o' = "...-"
'p' = "...-"    'q' = "...-"   'r' = "...-"   's' = "...-"   't' = "...-"
'u' = "...-"    'v' = "...-"   'w' = "...-"   'x' = "...-"   'y' = "...-"
'z' = "...-"

```

Figura 7.8: Salida del Ejemplo 7.5

7.4.1. Asignación dinámica de memoria

En el Ejercicio 10 de la Sección 2.4, se mencionó y presentó un ejemplo (Ejemplo 2.5) del operador `sizeof`. En esta sección se utilizará dicho operador para desarrollar, en conjunto con la función `malloc`, un par de ejemplos para mostrar la asignación dinámica de memoria.

Arreglos

El primer ejemplo se muestra en el Ejemplo 7.6. La línea 12 define un arreglo de enteros de tamaño N de nombre *estatico*, y la línea 13 un apuntador a entero de nombre *dinamico*.

Las líneas 16-18 llenan el arreglo *estatico* con números aleatorios entre 0 y 100.

La parte medular del Ejemplo 7.6 ocurre en la línea 20. El argumento de `malloc`¹¹ representa el tamaño del bloque, o la cantidad en *bytes* de memoria solicitados. Si la memoria solicitada es concedida, `malloc` regresa un apuntador al inicio del bloque, en caso contrario, se regresa un apuntador nulo (`NULL`).

Note que la línea 20 está solicitando la cantidad de memoria necesaria para almacenar N enteros ($N * \text{sizeof}(int)$).

Si la función `malloc` tiene éxito, asignará a la variable *dinamico*, la dirección de inicio del bloque solicitado, y dado que la función `malloc` regresa dicha dirección como un apuntador a `void` (`void *`)¹², la línea 20 hace una conversión forzada (*cast*) al tipo de dato de la variable *dinamico* para su adecuada interpretación de ahí en adelante: un apuntador a enteros que hará referencia

¹¹La función `malloc` se encuentra definida en la biblioteca estándar de funciones `stdlib.h` (línea 6).

¹²Un apuntador a `void` es un tipo genérico, puede interpretarlo como un tipo de dato comodín, y esto se debe a que la función `malloc` no sabe (y de hecho no le interesa) para qué será utilizada, ni cómo será interpretada la memoria solicitada.

a un arreglo de enteros.

```

1  /* Programa que muestra el uso de la funcion malloc para obtener
2     memoria dinamicamente (durante la ejecucion del programa).
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  #define N 5
10
11 int main() {
12     int estatico[N];
13     int *dinamico;
14     int i, j;
15
16     srand(time(NULL));
17     for(i = 0; i < N; i++)
18         estatico[i] = rand();
19
20     dinamico = (int *) malloc(N * sizeof(int));
21
22     if(dinamico != NULL){
23         for(i = 0, j = N-1; i < N; i++, j--){
24             dinamico[j] = estatico[i]; /* *(dinamico + j) = estatico[i]; */
25         }
26         for(i = 0; i < N; i++){
27             printf("estatico[%d] = %d\t", i, estatico[i]);
28             /* printf("*(estatico + %d) = %d\t", i, *(estatico + i)); */
29             printf("*(dinamico + %d) = %d\n", i, *(dinamico + i));
30             /* printf("dinamico[%d] = %d\n", i, dinamico[i]); */
31         }
32         free(dinamico);
33     } else
34         printf("No hubo memoria\n");
35     putchar('\n');
36     return 0;
37 }

```

Ejemplo 7.6: Asignación dinámica de memoria

La verificación de éxito por parte de la función *malloc* ocurre en la línea 22, en caso contrario¹³, la situación se reporta y el programa termina, lo cual es habitual para los programas que trabajan con asignación dinámica de memoria.

La representación de la memoria asignada por la función *malloc* para el

¹³Casi siempre la memoria solicitada por medio de *malloc* es concedida, pero debe tomar en cuenta que no tiene por qué ser siempre así, debido por ejemplo, a un exceso en la cantidad de memoria solicitada, o a que existen otros procesos de mayor prioridad solicitando también memoria, o a que no hay suficiente memoria para atender por el momento las solicitudes de *malloc*, entre otras muchas situaciones.

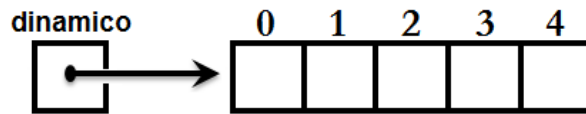


Figura 7.9: Representación del arreglo generado en el Ejemplo 7.6

Ejemplo 7.6 se muestra en la Figura 7.9. Observe que la representación es un arreglo referido por la variable *dinamico*.

El ciclo **for** de la línea 23 copia el arreglo *estatico* (línea 12) en el arreglo *dinamico* generado en la línea 20 pero con los elementos invertidos. Observe que se usa la notación de arreglos para ambos arreglos, y que entre comentarios aparece la notación con apuntadores y arreglos, de hecho:

$$dinamico[j] \equiv *(dinamico + j) \quad (7.1)$$

La Notación 7.1 hace uso de la aritmética de apuntadores. Note que no se está modificando la dirección de referencia, sino que a dicha dirección se le está sumando un *desplazamiento* determinado por j , para acceder al elemento correspondiente y hacer la respectiva desreferencia.

Observe también que:

$$dinamico + j \neq dinamico = dinamico + j \quad (7.2)$$

Asegúrese de comprender la expresión 7.2 antes de continuar.

Ahora bien, en base a lo expuesto en 7.1, la notación de arreglo entonces suma un desplazamiento (determinado por el índice), a una dirección base (el apuntador).

El ciclo **for** de la línea 25 recorre ambos arreglos para visualizar su contenido en la salida estándar. La línea 26 hace uso de la notación de arreglo para *estatico* y de la notación de apuntador para *dinamico* en la línea 28, en directa correspondencia a sus respectivas naturalezas.

Por otro lado, las líneas 27 y 29 aparecen en comentario, pero muestran que es posible utilizar notación de apuntadores para un arreglo estático (declarado en la línea 12), y notación de arreglos para un arreglo que ha sido generado con asignación dinámica de memoria (línea 20), lo cual es ejemplo de la versatilidad y expresividad del lenguaje de programación C.

La salida del Ejemplo 7.6 se presenta en la Figura 7.10.

estatico[0] = 72	*(dinamico + 0) = 83
estatico[1] = 52	*(dinamico + 1) = 96
estatico[2] = 61	*(dinamico + 2) = 61
estatico[3] = 96	*(dinamico + 3) = 52
estatico[4] = 83	*(dinamico + 4) = 72

Figura 7.10: Salida del Ejemplo 7.6

Por último, vale la pena mencionar que el tamaño de un arreglo que se genere utilizando asignación dinámica de memoria a través de la función *malloc*, estará limitado exclusivamente por la cantidad de memoria disponible de la computadora, y por las restricciones que establezca el sistema operativo anfitrión¹⁴.

Matrices

El segundo ejemplo de esta sección está compuesto por dos partes, la primera de ellas aparece en el Ejemplo 7.8, y la segunda parte la conforma su correspondiente biblioteca de funciones del Ejemplo 7.7.

La parte medular de la sección yace en la función *creaMatriz* (líneas 17-31) de la biblioteca de funciones, razón por la cual, los detalles de explicación inician con el Ejemplo 7.7.

Primero que nada, observe que la función *creaMatriz* regresa un doble apuntador a entero (línea 17), que corresponde al tipo de dato de la variable *matriz* de la línea 18, la cual se utilizará como valor de retorno en la línea 30.

```

1  /* Biblioteca de funciones que implementa funciones para trabajar con
2     matrices creadas con asignacion dinamica de memoria.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdlib.h>
6
7  void liberaMatriz(int **matriz, int m){
8      int i;
9
10     for(i = 0; i < m; i++)
11         free(matriz[i]); /* Libera los arreglos de enteros */
12
13     free(matriz); /* Libera el arreglo de apuntadores a entero */
14     matriz = NULL;
15 }
16

```

¹⁴Sistema operativo en donde se ejecuten los programas.

```

17 int **creaMatriz(int m, int n){
18     int **matriz;
19     int i;
20
21     matriz = (int **) malloc(m * sizeof(int *)); /* Arreglo de apuntadores */
22     if(matriz != NULL)
23         for(i = 0; i < m; i++){
24             /* Arreglo de enteros asignado a cada apuntador del arreglo matriz */
25             matriz[i] = (int *) malloc(n * sizeof(int));
26             /* *(matriz + i) = (int *) malloc(n * sizeof(int)); */
27             if (matriz[i] == NULL) /* Si falla la creacion de algun renglon: */
28                 liberaMatriz(matriz, m); /* hacer limpieza */
29         }
30     return matriz;
31 }
32
33 void leeMatriz(int **matriz, int m, int n, char *c){
34     int i, j;
35
36     for(i = 0; i < m; i++){
37         printf("Renglon %d/%d:\n", i, m - 1);
38         for(j = 0; j < n; j++){
39             printf("%s[%d][%d] = ? ", c, i, j);
40             scanf("%d", &matriz[i][j]);
41         }
42         putchar('\n');
43     }
44 }
45
46 void imprimeMatriz(int **matriz, int m, int n, char *c){
47     int i, j;
48
49     putchar('\n');
50     for(i = 0; i < m; i++){
51         for(j = 0; j < n; j++){
52             printf("%s[%d][%d] = %d ", c, i, j, matriz[i][j]);
53
54             putchar('\n');
55         }
56     }
57
58 int **suma(int **a, int **b, int m, int n){
59     int **c;
60     int i, j;
61
62     if((c = creaMatriz(m, n)) != NULL)
63         for(i = 0; i < m; i++){
64             for(j = 0; j < n; j++){
65                 c[i][j] = a[i][j] + b[i][j];
66             }
67         }
68     return c;
69 }

```

Ejemplo 7.7: Biblioteca de funciones para crear matrices con asignación dinámica de memoria

La línea 21 genera un arreglo parecido al del Ejemplo 7.6 en la línea 20,

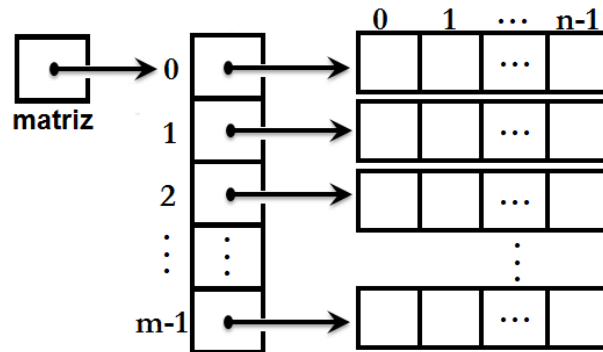


Figura 7.11: Representación del arreglo generado en el Ejemplo 7.7

con la diferencia de que el arreglo generado en la línea 21 es un arreglo de m apuntadores a entero, y que el *cast* se realiza en función del tipo de dato de la variable *matriz*. La Figura 7.11 muestra, de forma vertical, el arreglo de apuntadores generado.

La Figura 7.11, debería ser suficiente para entender por qué *matriz* (línea 18) es un doble apuntador: el arreglo vertical es de apuntadores y por lo tanto, sólo puede ser referido por un apuntador que haga referencia a variables de tipo apuntador, es decir, un doble apuntador.

Ahora bien, si el arreglo de apuntadores es concedido (línea 22), para cada uno de los m apuntadores (línea 23) se genera un arreglo de n enteros (línea 25)¹⁵; los arreglos generados en la línea 25 están representados de manera horizontal en la Figura 7.11. Note el uso de la notación de arreglos para la variable *matriz* en la línea 25, y la notación de apuntadores entre comentarios en la línea 26, ambas líneas son equivalentes.

Para cada arreglo de enteros solicitado (línea 25) se debe verificar si éste ha sido otorgado o no (línea 27). En caso de que no, se procede a eliminar, en caso de que hayan sido creados, los elementos parciales de la matriz generados hasta ese momento por medio de la función *liberaMatriz*.

La función *liberaMatriz* de las líneas 7-15, recibe un doble apuntador a entero (la matriz a liberar), y el número de renglones de la matriz. La función libera cada uno de los elementos del arreglo de apuntadores (representados horizontalmente de la Figura 7.11). Por último, la función libera el arreglo de apuntadores (línea 13), y establece que *matriz* no hace referencia a nada: apunta a *NULL* (línea 14).

¹⁵Este sí es un arreglo como el de la línea 20 del Ejemplo 7.6.

Continuando, la función *leeMatriz* de las líneas 33-44, es análoga a la versión realizada en el Ejemplo 6.7 del Capítulo 6, la diferencia es que ahora *matriz* se declara con notación de apuntador y se recibe además un apuntador a **char** *c*, el cuál es utilizado para imprimir el nombre de la matriz (%s de la línea 39) que se le envíe (líneas 21 y 22 del Ejemplo 7.8), lo cual hace más versatil e informativa la lectura de datos. Asegúrese de comparar las funciones y de comprender su equivalencia.

```

1  /* Uso de malloc para la creacion de una matriz de m x n
2     usando memoria dinamica.
3     @autor: Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include "matricesPtr.h"
7
8  int main(){
9      int **matA, **matB, **matC;
10     int m, n;
11
12     do{
13         printf("Escribe m y n (ambos mayores a 1): ");
14         scanf("%d %d", &m, &n);
15     }while(m < 2 || n < 2);
16
17     matA = creaMatriz(m, n);
18     matB = creaMatriz(m, n);
19
20     if(matA != NULL && matB != NULL){
21         leeMatriz(matA, m, n, "A");
22         leeMatriz(matB, m, n, "B");
23         matC = suma(matA, matB, m, n);
24         imprimeMatriz(matA, m, n, "A");
25         imprimeMatriz(matB, m, n, "B");
26         imprimeMatriz(matC, m, n, "C");
27         liberaMatriz(matA, m);
28         liberaMatriz(matB, m);
29         liberaMatriz(matC, m);
30         /* imprimeMatriz(matA, m, n); <— ??? */
31     }
32
33     return 0;
34 }

```

Ejemplo 7.8: Asignación dinámica de memoria para matrices

Por otro lado, la función *imprimeMatriz* de las líneas 46-56, es también análoga a la versión realizada en el Ejemplo 6.7 del Capítulo 6. Ahora *matriz* se declara con notación de apuntador y se ha agregado un cuarto parámetro: el apuntador a **char** *c*, el cuál es utilizado para imprimir el nombre de la matriz (%s de la línea 52) que se le envíe (líneas 24, 25 y 26 del Ejemplo 7.8), lo cual la dota de mayor versatilidad. Al igual que antes, asegúrese

también de comparar las funciones y de comprender su equivalencia.

Por último, en lo que compete al Ejemplo 7.7, la función *suma* de las líneas 58-67, es la versión que se deseaba en el Ejemplo 6.10 de la Sección 6.4. Note que la función regresa la matriz resultado *c* como un doble apuntador (líneas 58, 59, 62¹⁶ y 66), y que recibe únicamente los operandos (matrices *a* y *b*) también como doble apuntador.

Pasando ahora al Ejemplo 7.8, lo primero que debe observar es la inclusión de la biblioteca personalizada de funciones del Ejemplo 7.7 en la línea 6.

La línea 9 muestra tres dobles apuntadores a entero de nombres *matA*, *matB* y *matC*, mismos que representarán las matrices *A*, *B* y *C* respectivamente, mientras que las líneas 12-15 realizan una validación como la del Ejemplo 6.7.

Por otro lado, en las líneas 17 y 18 se genera la memoria, a través de la función *creaMatriz*, para las matrices *matA* y *matB* de *m* renglones y *n* columnas cada una. Si las matrices pudieron ser creadas (línea 20), se lee de la entrada estándar, por medio de la función *leeMatriz*, *matA* (línea 21) y *matB* (línea 22).

La línea 23 realiza la suma de las matrices *matA* y *matB* de *m* \times *n*, y la matriz que regresa como resultado se asigna a *matC*.

Por ultimo, las líneas 24, 25 y 26 imprimen en la salida estándar las matrices *matA*, *matB* y *matC* respectivamente, mientras que las líneas 27, 28 y 29 las liberan.

La salida del Ejemplo 7.8 puede ser bastante extensa dependiendo de las dimensiones de las matrices, una posible salida se muestra en la Figura 7.12.

Experimente con el programa de ejemplo, genere varias matrices y vea lo que sucede, hasta tener un nivel de comprensión satisfactorio de todas las sentencias que lo componen.

7.4.2. Argumentos en la invocación de programas

Muchos programas y comandos en línea reciben y procesan argumentos que modifican o alteran su funcionamiento. El compilador GNU de C (gcc) por ejemplo, cuando se ejecuta en la línea de comandos, recibe argumentos

¹⁶En esta línea es de hecho donde se crea la matriz *c* valiéndose de la función *creaMatriz*. Note que la memoria regresada por la función *malloc* no es local a ninguna función, y que ésta está disponible y accesible mientras no se haga una liberación explícita, o termine el programa en donde se generó.

```

Escribe m y n (ambos mayores a 2): -2 5
Escribe m y n (ambos mayores a 2): 2 -1
Escribe m y n (ambos mayores a 2): 2 3
Renglon 0/1:
A[0][0] = ? 1
A[0][1] = ? 2
A[0][2] = ? 3

Renglon 1/1:
A[1][0] = ? 4
A[1][1] = ? 5
A[1][2] = ? 6

Renglon 0/1:
B[0][0] = ? 4
B[0][1] = ? 5
B[0][2] = ? 6

Renglon 1/1:
B[1][0] = ? 1
B[1][1] = ? 2
B[1][2] = ? 3

A[0][0] = 1 A[0][1] = 2 A[0][2] = 3
A[1][0] = 4 A[1][1] = 5 A[1][2] = 6

B[0][0] = 4 B[0][1] = 5 B[0][2] = 6
B[1][0] = 1 B[1][1] = 2 B[1][2] = 3

C[0][0] = 5 C[0][1] = 7 C[0][2] = 9
C[1][0] = 5 C[1][1] = 7 C[1][2] = 9

```

Figura 7.12: Salida del Ejemplo 7.8


```
Argumento 0: ./cambiaBase
Argumento 1: 101011101
Argumento 2: 2
Argumento 3: 9
```

Figura 7.13: Salida del Ejemplo 7.9

como el nombre del programa a compilar, si se va a generar un archivo de salida ejecutable específico, etc. (vea el Apéndice B para mayor información).

La función principal de cada programa escrito en C (**main**), es capaz de recibir argumentos que pueden ser manipulados y procesados. El Ejemplo 7.9 muestra un sencillo programa de prueba de éstas capacidades.

```
1  /* Programa que muestra los argumentos proporcionados
2   a la funcion main en la invocacion del programa.
3   @autor Ricardo Ruiz Rodriguez
4   */
5  #include <stdio.h>
6
7  int main(int argc, char *argv[]) {
8      int i;
9
10     for(i = 0; i < argc; i++)
11         printf("Argumento %d: %s\n", i, argv[i]);
12
13     return 0;
14 }
```

Ejemplo 7.9: Argumentos de la función **main**

Observe primero la línea 7, la cual presenta una lista de dos argumentos para **main**:

1. **argc** (*argument counter*) es el número de argumentos proporcionados en la invocación del programa, incluyendo el nombre del programa.
2. **argv** (*arguments vector*) es un arreglo de apuntadores a **char** de longitud *argc*. Cada elemento del arreglo, hará referencia a una cadena correspondiente al argumento en cuestión.

El ciclo **for** de la línea 10 se repite *argc* veces, e imprime en la salida estándar (línea 11) los *argc* argumentos recibidos y referidos por cada uno de los apuntadores de *argv*.

Suponga que el nombre del archivo del Ejemplo 7.9 es *mainArgs.c* y que se compila de la siguiente manera:

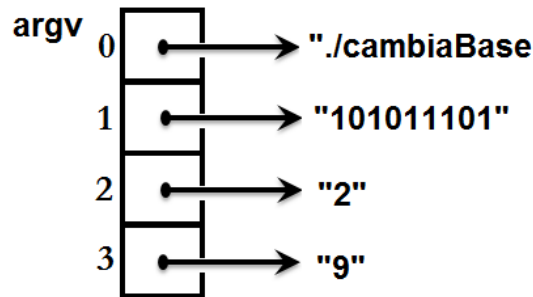


Figura 7.14: Representación de *argv* para la ejecución mostrada en 7.13

```
$gcc -o cambiaBase mainArgs.c
```

El ejecutable de *mainArgs.c* se genera entonces en el archivo *cambiaBase*, por lo que si ahora se ejecuta el programa de la forma siguiente:

```
$./cambiaBase 101011101 2 9
```

Se obtendrá la salida de la Figura 7.13 y su correspondiente representación mostrada en la Figura 7.14.

Observe que el valor de *argc* es de cuatro, y cómo cada argumento es referido por los cuatro apuntadores de *argv* para el caso específico de la ejecución mostrada en la Figura 7.13. Note también que el primer argumento es el nombre del programa.

La idea aquí expuesta, será retomada en la sección de ejercicios, en donde tendrá que usarla para procesar dichos argumentos para determinar los datos de entrada para un programa convertidor de bases.

7.5. Apuntadores a funciones

En la Sección 6.1.4 se presentó, describió e implementó el ordenamiento por burbuja, en esta sección se retomará dicho algoritmo para abordar la siguiente problemática: se desea ahora poder ordenar los datos de manera ascendente o descendente, ¿cómo lo solucionaría con lo que sabe hasta ahora?.

Una posible solución sería hacer dos funciones: una que ordene los datos de manera ascendente, y otra que los ordene de manera descendente, tal y como se muestra en el Ejemplo 7.10.

```

1  /* Biblioteca de funciones que implementa el ordenamiento
2     por burbuja para ordenar ascendente o descendente.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #define VERDADERO 1
6  #define FALSO     0
7
8  void burbujaAscendente(int *a, const int n){
9      int veces, i, aux, bandera = VERDADERO;
10
11     for(veces = 1; veces < n && bandera; veces++){
12         bandera = FALSO;
13         for(i = 0; i < n - veces; i++){
14             if(a[i] > a[i + 1]){
15                 aux = a[i];
16                 a[i] = a[i + 1];
17                 a[i + 1] = aux;
18                 bandera = VERDADERO;
19             }
20         }
21     }
22
23     void burbujaDescendente(int *a, const int n){
24         int veces, i, aux, bandera = VERDADERO;
25
26         for(veces = 1; veces < n && bandera; veces++){
27             bandera = FALSO;
28             for(i = 0; i < n - veces; i++){
29                 if(a[i] < a[i + 1]){
30                     aux = a[i];
31                     a[i] = a[i + 1];
32                     a[i + 1] = aux;
33                     bandera = VERDADERO;
34                 }
35             }
36         }

```

Ejemplo 7.10: Ordenamiento ascendente y descendente por burbuja

La función *burbujaAscendente* ordena los datos de forma ascendente, y la función *burbujaDescendente* los ordena de forma descendente, tal y como sus identificadores lo sugieren.

Compare las funciones y notará que son exactamente iguales, excepto por las líneas 14 y 29 respectivamente, que es en donde se realiza la comparación de los elementos para determinar el orden en que están e intercambiarlos, si fuera necesario, en base al tipo de ordenamiento.

La solución presentada, aunque es útil y válida, no es una solución eficiente en espacio, ni elegante. El lenguaje de programación C proporciona un mecanismo para abordar de mejor forma este tipo de situaciones: los apuntadores a funciones.

Un **apuntador a función** almacena la dirección en memoria del inicio de las instrucciones de una función, la cuál está representada por su identificador.

De manera informal puede decirse entonces, que un apuntador a función es un mecanismo que permite hacer referencia a una función de manera implícita.

```

1  /* Biblioteca de funciones que implementa el ordenamiento
2     por burbuja mejorado utilizando un apuntador a funcion
3     para ordenar ascendente o descendentemente.
4     @autor Ricardo Ruiz Rodriguez
5  */
6  #define VERDADERO 1
7  #define FALSO     0
8
9  int ascendente(int a, int b){
10     return a > b;
11 }
12
13 int descendente(int a, int b){
14     return b > a;
15 }
16
17 void burbuja(int *a, const int n, int (*orden)(int, int)){
18     int veces, i, aux, bandera = VERDADERO;
19
20     for(veces = 1; veces < n && bandera; veces++){
21         bandera = FALSO;
22         for(i = 0; i < n - veces; i++){
23             if((*orden)(a[i], a[i + 1])){
24                 aux = a[i];
25                 a[i] = a[i + 1];
26                 a[i + 1] = aux;
27                 bandera = VERDADERO;
28             }
29         }
30     }

```

Ejemplo 7.11: Burbuja con apuntador a funciones

Los apuntadores a funciones pueden asignarse, ser almacenados en arreglos, pasados a funciones, regresados por funciones, etcétera. El Ejemplo 7.11 muestra la forma de definir un apuntador a funciones en la función *burbuja* de las líneas 17-30.

La explicación de los ejemplos siguientes se centrará exclusivamente en el tema de la sección: apuntadores a funciones, no en el ordenamiento por burbuja.

El tercer parámetro de la función *burbuja* (línea 17), indica que se recibirá un apuntador a funciones que tengan la siguiente estructura:

- Reciban dos argumentos de tipo **int**.
- Regresen un valor de tipo **int**.

Cualquier función que cumpla con esta plantilla, podrá ser referida por el apuntador *orden*.

Note que en la expresión:

```
int (*orden)(int, int)
```

los paréntesis que circundan a **orden* son muy importantes, debido a que le indican al compilador que *orden* es un apuntador a funciones que reciben dos argumentos de tipo **int**, y que regresen un valor de tipo **int**.

```

1  /* Programa de prueba para la biblioteca de funciones que implementa
2  el ordenamiento por burbuja utilizando un apuntador a funcion.
3  @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include "burbujaPtr.h"
7
8  void imprimeArreglo(const int *, const int);
9
10 int main(){
11     int a[ ] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
12
13     printf("Datos en el orden original:\n");
14     imprimeArreglo(a, 10);
15     printf("Datos en orden ascendente:\n");
16     burbuja(a, 10, ascendente);
17     imprimeArreglo(a, 10);
18     printf("Datos en orden descendente:\n");
19     burbuja(a, 10, descendente);
20     imprimeArreglo(a, 10);
21
22     return 0;
23 }
24
25 void imprimeArreglo(const int *a, const int n){
26     int i;
27
28     for(i = 0; i < n; i++)
29         printf("%d ", a[i]);
30     putchar('\n');
31 }
```

Ejemplo 7.12: Uso del ordenamiento por burbuja con apuntador a funciones

Por otro lado, la expresión:

```
int *orden(int, int)
```

```

Datos en el orden original:
2 6 4 8 10 12 89 68 45 37
Datos en orden ascendente:
2 4 6 8 10 12 37 45 68 89
Datos en orden descendente:
89 68 45 37 12 10 8 6 4 2

```

Figura 7.15: Salida del Ejemplo 7.12

le indicaría al compilador que *orden* es una función que recibe dos argumentos de tipo **int**, y regresa un apuntador a **int**, lo cual es definitiva y completamente diferente.

Observe ahora la línea 23 del Ejemplo 7.11, en la cual ocurre la desreferencia del apuntador a función, es decir, en ella se invocará a la función cuyo identificador coincida con el tercer argumento enviado a la función *burbuja* en su invocación (líneas 16 y 19 del Ejemplo 7.12).

Ahora bien, el Ejemplo 7.12 muestra la forma de *pasar* el apuntador denotado por el identificador respectivo, a la función *burbuja* en las líneas 16 y 19. Note que en la línea 6 se está incluyendo la biblioteca de funciones del Ejemplo 7.11.

Observe que el apuntador a funciones *orden*, hace que la línea 23 del Ejemplo 7.11 se transforme en un llamado explícito a las funciones *ascendente* o *descendente*, es decir, convierte la expresión:

```
if((*orden)(a[i], a[i + 1]))
```

en la expresión:

```
if(ascendente(a[i], a[i + 1]))
```

para el llamado a la función *burbuja* (línea 16) del Ejemplo 7.12. Mientras que la expresión:

```
if((*orden)(a[i], a[i + 1]))
```

es convertida en la expresión:

```
if(descendente(a[i], a[i + 1]))
```

para el llamado a la función *burbuja* (línea 19) del Ejemplo 7.12.

Por último, la función *ascendente* (líneas 9-11) del Ejemplo 7.11, regresa 1 (verdadero) si *a* es mayor que *b*, y 0 (falso) si no lo es, lo cual corresponde al criterio requerido para un ordenamiento ascendente. Observe que la expresión de la línea 10:

```
return a > b;
```

es lógicamente equivalente a la estructura de selección **if-else**:

```
if(a > b)
    return 1;
else
    return 0;
```

y que la función *descendente* (líneas 13-15) se comporta de manera análoga a la función *ascendente*. La salida del Ejemplo 7.12 se muestra en la Figura 7.15.

En resumen, en el contexto de apuntadores a variables, los apuntadores almacenan las direcciones de memoria de otras variables, mientras que en el contexto de funciones, los apuntadores almacenan la dirección de memoria del inicio de las instrucciones de una determinada función. Note que en ambos casos, lo que el compilador y la computadora controlan son *direcciones de memoria*, mientras que el programador controla *identificadores*, lo cual es una abstracción sumamente conveniente.

7.5.1. Arreglos de apuntadores a funciones

En diferentes tipos de programas y aplicaciones, es frecuente encontrar menús de opciones para seleccionar una determinada acción u operación a realizar.

Cuando muchas o todas las opciones presentadas en un menú comparten características similares, una excelente opción, aunque no la única, es implementar dichas opciones a través de un arreglo de apuntadores a función.

En el Ejemplo 4.3 de la Sección 4.2.2, se desarrolló una calculadora con las operaciones aritméticas básicas de suma, resta, multiplicación y división. En esta sección se retomará dicho ejemplo para reescribirlo utilizando un arreglo de apuntadores a funciones.

El Ejemplo 7.13 muestra que las cuatro funciones de *suma*, *resta*, *multiplica* y *divide*, comparten un patrón común: todas reciben dos argumentos de tipo **float** y regresan un valor de tipo **float**.

```

*** MENU ***
[1] Suma
[2] Resta
[3] Multiplicacion
[4] Division
[5] Salir
Opcion: 4

Operandos (a b)? 1 3

1.00 / 3.000000 = 0.33

```

Figura 7.16: Salida del Ejemplo 7.13

El patrón identificado se aprovecha en la línea 15, la cual declara un arreglo *funciones* de cuatro apuntes a funciones que reciben dos argumentos de tipo **float** y regresan un valor de tipo **float**. El arreglo es inicializado en la declaración con una lista de identificadores: *suma*, *resta*, *multiplica* y *divide*.

El uso del arreglo de apuntes *funciones* se presenta en la línea 29, en donde el valor de la variable *op* se utiliza como índice para invocar a la función correspondiente que realizará la operación seleccionada, en base al menú de opciones (líneas 37-51).

En este sentido, si *op* tiene el valor tres por ejemplo, la expresión:

$$(*funciones[op-1])(a, b)$$

se transforma en la expresión:

$$multiplica(a, b)$$

debido a que el identificador que se encuentra en la posición dos (*op - 1*) del arreglo de apuntes a funciones *funciones*, es precisamente *multiplica*. La resolución de los otros identificadores ocurre de manera análoga.

El llamado a la función a invocar se resuelve en tiempo de ejecución, ya que el llamado a través del arreglo de apuntes a funciones de la línea 29 es un llamado implícito.

Finalmente, la función *menu* indica con **void** que no recibirá ningún tipo de argumento. Todos los demás detalles del programa de ejemplo ya deberían resultarle familiares al lector. Una posible salida para el Ejemplo 7.13 se muestra en la Figura 7.16.


```

1  /* Programa que simula una calculadora basica y muestra el uso
2     de un arreglo de apuntadores a funciones.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define SALIR 5
7
8  int menu(void);
9  float suma(float , float);
10 float resta(float , float);
11 float multiplica(float , float);
12 float divide(float , float);
13
14 int main(){
15     float (*funciones[4])(float , float) = {suma, resta, multiplica, divide};
16     float a, b;
17     char operadores[] = "+-*/";
18     int op;
19
20     do{
21         op = menu();
22         if(op != SALIR){
23             printf("\nOperandos (a b)? ");
24             scanf("%f %f", &a, &b);
25             if(op == 4 && b == 0)
26                 printf("\tError: intento de division por cero\n");
27             else
28                 printf("\n\t%.2f %c %.2f = %f\n",
29                     a, operadores[op-1], b, (*funciones[op-1])(a, b));
30         }
31     }while(op != SALIR);
32
33     return 0;
34 }
35
36 int menu(void){
37     int op;
38
39     do{
40         printf("\n\t\t*** MENU ***\n");
41         printf("\t[1] Suma\n");
42         printf("\t[2] Resta\n");
43         printf("\t[3] Multiplicacion\n");
44         printf("\t[4] Division\n");
45         printf("\t[5] Salir\n");
46         printf("\tOpcion: ");
47         scanf("%d", &op);
48     }while(op < 1 || op > 5);
49
50     return op;
51 }
52
53 float suma(float a, float b){
54     return a + b;
55 }
56

```

```
57 float resta(float a, float b){  
58     return a - b;  
59 }  
60  
61 float multiplica(float a, float b){  
62     return a * b;  
63 }  
64  
65 float divide(float a, float b){  
66     return a / b;  
67 }
```

Ejemplo 7.13: Arreglo de apuntadores a funciones

Cabe mencionar por último, que la forma de utilizar los arreglos de apuntadores a funciones mostrada en el Ejemplo 7.13, no es su única posibilidad de uso, sino que es una de tantas en el amplio abanico de opciones en la programación estructurada; en éste sentido, la limitante respecto al uso de apuntadores a funciones en general, estará más en función de la imaginación y capacidad del programador, que de las múltiples posibilidades de su uso y aplicación.

7.6. Ejercicios

1. Modifique el Ejemplo 7.3 para que lea una cadena de la entrada estándar, y la envíe a las funciones *imprimeAlReves* e *imprimeSinEspacios*.
2. Determine lo que realiza la siguiente función sin ejecutarla, posteriormente, corrobore su respuesta con un programa que la pruebe:

```
int queHace(const char *cPtr){
    const char *inicioPtr = cPtr;

    while(*cPtr != '\0')
        cPtr++;
    return cPtr - inicioPtr;
}
```

3. Escriba un programa que defina y pruebe una función con la siguiente firma:

void rellenaArreglo(int *a, int n, int rango)

La función deberá inicializar los n elementos referidos por a , con números **aleatorios** definidos entre 0 y $rango - 1$.

4. Escriba un programa que defina y pruebe una función con la siguiente firma:

void inicializaArreglo(int *a, int n, int valor)

La función deberá inicializar los n elementos referidos por a , con el valor *valor*.

5. Escriba un programa que defina y pruebe una función con la siguiente firma:

void imprimeArreglo(const int *a, int n, int c)

La función deberá imprimir en la salida estándar, los n elementos referidos por a , con c elementos por renglón.

6. Escriba un programa que defina y pruebe una función con la siguiente firma:

int *copiaArreglo(const int *original, int n)

La función deberá copiar los n elementos referidos por *original*, en un

arreglo que haya sido creado utilizando memoria dinámica dentro de la función, mismo que será utilizado como valor de retorno para la función.

7. Escriba un programa que pruebe cada una de las funciones de la biblioteca *ctype*, su programa deberá probar al menos las funciones descritas en la Sección 7.3, pero recuerde que dicha biblioteca contiene más funciones.
8. En base al programa del Ejemplo 7.4, escriba una función análoga a la función *convierteAMayusculas*, que convierta la cadena referida por *cPtr* a minúsculas, utilizando las funciones de la biblioteca *ctype* descritas en la Sección 7.3.
9. Escriba un programa que defina y pruebe una función con la siguiente firma:

char *copiaCadena(const char *original)

La función deberá copiar los elementos referidos por *original*, en un arreglo de caracteres que haya sido creado utilizando memoria dinámica dentro de la función, mismo que será utilizado como valor de retorno para la función.

10. Escriba un programa que defina y pruebe una función con la siguiente firma:

int compara(const char *c1, const char *c2)

La función deberá determinar si los elementos referidos por *c1*, son iguales (regresa 1) o no (regresa 0) a los elementos referidos por *c2*.

11. Escriba un programa que defina y pruebe una función con la siguiente firma:

void imprimeAlReves(const char *original)

La función deberá imprimir **invertidos** los elementos referidos por *original*. La función deberá usar recursividad para imprimir al revés los elementos de *original*.

12. Escriba un programa que defina y pruebe una función con la siguiente firma:

char *invierteCadena(const char *original)

La función deberá copiar **invertidos** los elementos referidos por *original*, en un arreglo de caracteres que haya sido creado utilizando memoria dinámica dentro de la función, mismo que será utilizado como valor de retorno para la función.

13. Escriba un programa que defina y pruebe una función con la siguiente firma:

void quitaEspacios(char *c)

La función deberá eliminar (si los hubiera) los espacios de los elementos referidos por *c*. Note que la función *modifica* la cadena original referida por *c*, por lo que deberá asegurarse de terminar adecuadamente la cadena con `'\0'`.

14. Escriba un programa que defina y pruebe una función con la siguiente firma:

int esPalindromo(const char *c)

La función deberá determinar si los elementos referidos por *c*, constituyen (regresa *1*) o no (regresa *0*) un palíndromo.

La función deberá ignorar espacios, signos de puntuación, etcétera, y no hará distinción entre mayúsculas y minúsculas. Puede apoyarse de las funciones desarrolladas hasta el momento en los ejercicios y en el texto, como la función *convierteAMayusculas* del Ejemplo 7.4 y las funciones de la biblioteca *ctype* descritas en la Sección 7.3.

15. Utilice el Ejemplo 7.5, para escribir una función con la siguiente firma:

void aMorse(const char *c)

La función deberá imprimir en la salida estándar, la representación en código Morse de la cadena referida por *c*. La representación en código Morse deberá estar separada por comas “,” por cada carácter de *c*.

16. En base al Ejemplo 7.6 y a la notación de aritmética de apuntadores, sustituya la línea 24 por el comentario correspondiente de la misma línea, y compruebe la equivalencia de la Notación 7.1. Realice lo mismo intercambiando las líneas 26 y 27, y las líneas 28 y 29 respectivamente.
17. En base al Ejemplo 7.7, pruebe que las líneas 25 y 26 son intercambiables.
18. En base al Ejemplo 7.7, reescriba la línea 27 utilizando notación de apuntadores en lugar de la notación de arreglos.

19. La línea 30 del programa del Ejemplo 7.8 está comentada, ¿qué sucedería si se descomenta dicha línea, se compila y ejecuta nuevamente el programa?. Determine su respuesta y posteriormente compruébela.
20. Modifique la biblioteca de funciones del Ejemplo 7.7 para que las funciones trabajen con valores de tipo **double**, y genere una nueva biblioteca de funciones que gestione matrices con dicho tipo de dato.
21. Escriba un programa que defina y pruebe una función con la siguiente firma:

double **resta(double **a, double **b, int m, int n)

La función deberá calcular la resta de la matriz a menos la matriz b , de manera análoga a como lo hace la función *suma* del Ejemplo 7.7.

22. Agregue la función *resta* del ejercicio anterior, a la biblioteca de funciones creada en el Ejercicio 20.
23. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad (7.3)$$

La matriz transpuesta de A , denotada por A^T , está definida como:

$$(A^T)_{ij} = A_{ji}, \quad 0 \leq i \leq n-1, \quad 0 \leq j \leq m-1 \quad (7.4)$$

es decir:

$$\begin{pmatrix} a_{0,0} & \dots & a_{m-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,n-1} & \dots & a_{m-1,n-1} \end{pmatrix}_{n \times m} \quad (7.5)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

void imprimeTranspuesta(double **a, int m, int n)

La función deberá **imprimir** en la salida estándar, la matriz transpuesta de a , siguiendo los lineamientos expresados en las Ecuaciones 7.4 y 7.5.

24. Escriba un programa que defina y pruebe una función con la siguiente firma:

double **transpuesta(double **a, int m, int n)

La función deberá **regresar** la matriz transpuesta de a , siguiendo los lineamientos expresados en las Ecuaciones 7.4 y 7.5.

25. Agregue la función *transpuesta* del ejercicio anterior, a la biblioteca de funciones creada en el Ejercicio 20.
26. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix}_{n \times n} \quad (7.6)$$

Se dice que la matriz A es simétrica respecto a la diagonal principal si:

$$a_{ij} = a_{ji}, \forall i \neq j \quad (7.7)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

int esSimetrica(double **a, int n)

La función deberá determinar si la matriz a , es o no simétrica en base a lo expuesto en la Ecuación 7.7.

27. Agregue la función *esSimetrica* del ejercicio anterior, a la biblioteca de funciones creada en el Ejercicio 20.
28. Sean A y B dos matrices definidas de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad B = \begin{pmatrix} b_{0,0} & \cdots & b_{0,l-1} \\ \vdots & \ddots & \vdots \\ b_{n-1,0} & \cdots & b_{n-1,l-1} \end{pmatrix}_{n \times l} \quad (7.8)$$

El producto de $A_{m \times n} \times B_{n \times l}$ denotado por $C_{m \times l}$, se define como:

$$C_{ij} = \sum_{r=0}^{n-1} a_{ir} b_{rj} \quad (7.9)$$

es decir:

$$\begin{pmatrix} a_{0,0}b_{0,0}+\dots+a_{0,n-1}b_{n-1,0} & \dots & a_{0,0}b_{0,l-1}+\dots+a_{0,n-1}b_{n-1,l-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}b_{0,0}+\dots+a_{m-1,n-1}b_{n-1,0} & \dots & a_{m-1,0}b_{0,l-1}+\dots+a_{m-1,n-1}b_{n-1,l-1} \end{pmatrix}_{n \times l} \quad (7.10)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

```
double **producto(double **a, double **b,  
int m, int n, int l)
```

de tal forma que la función calcule el producto de las matrices a y b regresando su resultado. Las matrices tienen la forma descrita en la Ecuación 7.8; y el producto se calcula siguiendo los lineamientos expuestos en las Ecuaciones 7.9 y 7.10.

29. Agregue la función *producto* del ejercicio anterior, a la biblioteca de funciones creada en el Ejercicio 20.
30. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \dots & a_{n-1,n-1} \end{pmatrix}_{n \times n} \quad (7.11)$$

La matriz inversa de A , denotada por A^{-1} , se define como:

$$A \times A^{-1} = A^{-1} \times A = I \quad (7.12)$$

donde I es la matriz identidad.

Escriba un programa que calcule, si existe, la matriz inversa A^{-1} de la matriz A . Investigue cómo se calcula la matriz inversa de una matriz, y utilice el método que más le convenga; se sugiere el método de Gauss-Jordan:

$$[A|I] \xrightarrow{\text{Gauss-Jordan}} [I|A^{-1}] \quad (7.13)$$

Utilice la siguiente firma para la función:

0	0	1	1	0
0	0	0	1	0
1	1	0	0	0
0	0	0	1	1
0	1	0	0	0

Tabla 7.1: Laberinto de 5×5

double **inversa(double **a, int n)

Si la matriz inversa de a no existe, la función deberá regresar *NULL*.

31. Agregue la función *inversa* del ejercicio anterior, a la biblioteca de funciones creada en el Ejercicio 20.
32. Escriba un programa que represente un laberinto por medio de una matriz, de tal forma que su programa:
 - a) Permita generar con asignación dinámica de memoria, una matriz cuadrada de tamaño n donde $n > 2$. La matriz almacenará únicamente 0's y 1's, y se identificará como *laberinto*.
 - b) Lea un entero que represente el porcentaje de obstáculos o de 1's que contendrá la matriz, de tal forma que si la matriz es de 10 x 10 por ejemplo y el porcentaje de 40, se deberán generar 40 1's de los 100 posibles valores a almacenar en la matriz.
 - c) Llene la matriz en un recorrido por renglones de manera aleatoria, tomando en consideración el porcentaje de 1's especificado en el punto anterior. La tabla 7.1 muestra un laberinto de 5×5 con un porcentaje de obstáculos de 35 %¹⁷.
 - d) Genere una sucesión de laberintos distintos en cada ejecución.

Asegúrese de que en la posición $(0, 0)$ y $(n-1, n-1)$ haya siempre 0's, ya que representan, el inicio y la meta del laberinto respectivamente.

Las reglas para caminar por el laberinto son las siguientes:

¹⁷Determine el porcentaje con una división entera, es decir, sin considerar fracciones.

- a) Se intenta avanzar primero hacia la derecha, si se puede avanzar se continúa con este criterio, en otro caso, se prueba el siguiente criterio.
- b) Se intenta avanzar hacia abajo, si se puede avanzar se inicia recursivamente el intento de avance, si no se puede hacia abajo, se prueba el siguiente criterio.
- c) Se intenta avanzar hacia la izquierda, si se puede avanzar se inicia recursivamente el intento de avance, en caso de que no, se intenta con el siguiente criterio.
- d) Se intenta avanzar hacia arriba, si se puede, se inicia recursivamente el intento de avance hasta encontrar la meta (posición $(n-1, n-1)$).

Su programa deberá encontrar, si existe, una solución al laberinto representado por la matriz *laberinto* haciendo uso de recursividad, donde los 0's representan caminos y los 1's obstáculos¹⁸.

La solución estará indicada en formato de coordenadas de renglones y columnas, y por la naturaleza recursiva, la salida puede estar dada de la meta al inicio del laberinto. Para el laberinto representado en la tabla 7.1, la solución es:

$(4, 4) \leq (4, 3) \leq (4, 2) \leq (3, 2) \leq (2, 2) \leq (1, 2) \leq (1, 1) \leq (0, 1) \leq (0, 0)$

33. En base a los Ejemplos 7.8 y 7.7, generalice las ideas expuestas en el texto para generar una estructura tridimensional como la de la Sección 6.3. La idea es que su programa permita leer valores de la entrada estándar para que sean asignados a su estructura tridimensional; así mismo, se requiere poder visualizar dichos valores en la salida estándar.

Note que tendrá que hacer uso de un triple apuntador.

34. Usando como referencia el Ejemplo 7.9 para procesar los argumentos en la invocación, escriba un programa que procese entradas como las siguientes:

```
$cambiaBase 101010101010 2 5
```

¹⁸Obviamente, los límites de la matriz representan también obstáculos.

```
$cambiaBase 1a4DEA 15 4
$cambiaBase Fea 16 12
$cambiaBase 12343102 5
```

En general:

```
$cambiaBase Nb1 b1 b2
```

Donde *Nb1* es un número en base *b1* y *b2* representa la base a la que se convertirá *Nb1* por medio de su programa. Su programa deberá también:

- a) Verificar que *Nb1* represente un número válido en la base *b1*, en caso de que no, deberá indicar dicha situación e identificar el primer dígito inválido. Ejemplo:

```
$cambiaBase 12383102 5
12383102
^Incorrecto
$
```

- b) Verificar que $b1, b2 \in \{2, 3, \dots, 16\}$, ya que cualquier otra base se procesará e indicará como incorrecta.
 - c) Considerar que si *b2* no se especifica, entonces *Nb1* se convertirá a base 10 por omisión.
 - d) Procesar mayúsculas o minúsculas en *Nb1* para las bases que contienen letras como parte de sus “dígitos”.
35. Escriba un programa que permita realizar las cuatro operaciones aritméticas básicas de suma, resta, multiplicación y división, para dos números de cualquier cantidad de “dígitos” representados en la misma base *b*, donde $b \in \{2, 3, \dots, 16\}$.

Tome en cuenta que ninguno de los tipos de datos de C tiene la capacidad solicitada, por lo que su programa deberá valerse de cadenas para la representación de dichos números, y realizar la implementación de

las operaciones aritméticas tomando en cuenta dicha representación y la manipulación simbólica de los “dígitos”. En éste sentido, es de esperarse que el resultado esté dado también por medio de una cadena, pero esto será transparente para el usuario.

Al igual que en el ejercicio anterior, su programa deberá:

- a) Procesar y obtener antes que nada la base b de los operandos.
- b) Presentar un menú con las opciones de operaciones aritméticas al estilo del Ejemplo 7.13, y utilizar un arreglo de apuntadores a funciones para invocar a la operación correspondiente de suma, resta, multiplicación o división.
- c) Verificar que $b \in \{2, 3, \dots, 16\}$, ya que cualquier otra base se procesará e indicará como incorrecta.
- d) Verificar que los operandos representen un número válido en la base b , en caso de que no, deberá indicar dicha situación e identificar el primer dígito inválido.
- e) Procesar mayúsculas o minúsculas en los operandos para las bases que contienen letras como parte de sus “dígitos”.

El programa no debe convertir los números a base 10, sino realizar las operaciones en la misma base en la que se expresan.

Sugerencias:

- a) Analice y resuelva primero el caso para la base 10, para poder generalizar su solución hacia las demás bases.
- b) Resuelva primero las operaciones de suma y resta, y las de multiplicación y división resuélvalas con sumas y restas sucesivas respectivamente.

Capítulo 8

Abstracción de datos

La abstracción es una característica inherente a la especie humana.

Desde el punto de vista de la programación, por medio de la abstracción se puede comprender o describir un concepto, sin tener que entrar en los detalles de su representación o especificación.

A lo largo del texto, se ha estado echando mano de dicho concepto, a través de diagramas o figuras, de las representaciones en la memoria de la computadora de algunos de los tipos de datos, o de las estructuras de datos analizadas, como los arreglos y las matrices por mencionar algunas.

Al escuchar conceptos como casa, automóvil, etc., se genera de manera casi inconsciente en nuestra mente, una representación de ellos, y dicha representación podría variar de persona a persona, pero en esencia, la idea sería sustancialmente la misma.

En este sentido, si alguien nos solicitara dibujar en una hoja de papel nuestro concepto de casa o de automóvil, con toda seguridad habría unos dibujos más elaborados que otros, algunos con más detalles, y otros con menos, pero sería poco probable que, en condiciones “normales”, alguien dibujara una casa con una vaca en lugar de un techo o una pared; alguien podría quizá colocarle a su dibujo la silueta de una chimenea, pero sería poco probable que alguien dibuje un piano sobre el techo.

Para el caso de un automóvil, alguna persona podría dibujar un auto deportivo o quizá algo más modesto, pero al igual que antes, sería poco probable que alguien dibujara un automóvil que, en lugar de ruedas, tuviera unas guitarras eléctricas.

La abstracción es en este sentido, lo que nos permite comunicarnos con otras personas o colegas a un nivel conceptual.

En este capítulo, se echará mano de un par de mecanismos utilizados en C para reforzar la abstracción en la programación a través de la representación de “nuevos”¹ tipos de datos: **struct** y **typedef**.

8.1. Conceptos, representación y estructura

Una **estructura** (**struct**) es una colección de una o más variables de tipos de datos posiblemente diferentes, agrupadas bajo un solo nombre para un mejor y más conveniente manejo de las mismas.

Las estructuras ayudan a organizar mejor los datos, mejorando así la abstracción y la representación de información para el programador.

Un ejemplo clásico de una estructura es la representación de los empleados de una nómina. Un empleado se describe en dicho contexto por un conjunto de atributos o características, tales como:

- Número de empleado.
- Nombre(s).
- Apellido paterno.
- Apellido materno.
- Clave Única de Registro de Población (CURP).
- Número de seguro social.
- Puesto.
- Domicilio.
- Teléfono.
- Salario.

¹En realidad no se generarán nuevos tipos de datos, ya que un tipo de dato requiere de una representación, una interpretación, y un conjunto de operaciones bien definidas y establecidas asociadas a él; de ahí que sólo se utilizarán, y en todo caso se agruparán, los tipos de datos existentes de manera conveniente, para proporcionar o dotar al programador de una mejor abstracción.

entre otros.

En C no es posible declarar una variable de tipo *empleado* o *persona* por ejemplo, ya que no existen dichos tipos de datos; sin embargo es posible, con los tipos de datos existentes, abstraer de manera conveniente variables que permitan representar objetos que sirvan como un tipo de datos.

Una estructura (**struct**) en C, tiene la siguiente estructura (la redundancia es necesaria):

```
struct identificador{
    tipo_de_dato1 lista_de_identificadores1;
    tipo_de_dato2 lista_de_identificadores2;
    .
    .
    .
    tipo_de_datoN lista_de_identificadoresN;
};
```

donde:

- **identificador** es un identificador válido en C y denota el nombre de la estructura.
- **tipo_de_dato** es alguno de los tipos de datos de C.
- **lista_de_identificadores** es una lista de identificadores separada por comas, para cada una de las variables definidas.

Observe que una estructura en C puede contener diferentes tipos de datos, y distintas variables de cada uno de ellos.

A cada una de las variables que se declaran dentro de una estructura se les denomina: **elemento miembro** de la estructura, y cada elemento miembro se distingue por su respectivo identificador, el cual fue definido en la *lista_de_identificadores*.

Al igual que antes, no puede haber identificadores repetidos para los elementos miembro de una estructura determinada. Los identificadores de los elementos miembro de una estructura, son locales a ella, y sólo puede accederse a través de la estructura, como se mostrará en el Ejemplo 8.1.

Los elementos miembro de una estructura e_1 pueden ser arreglos, apun­tadores e incluso variables cuyo tipo de dato se derive de otra estructura e_2 por ejemplo, el único requisito será que e_2 esté definida *antes* que e_1 .

El número de estructuras que se pueden definir está limitado por las necesidades del programador, C no impone ninguna restricción en éste sentido.

Una estructura sólo define una *plantilla* o *molde* que agrupa otras variables, por sí misma no crea las variables de sus elementos miembro. Los elementos miembro de la estructura se crean cuando se declara una variable cuyo tipo es una estructura, de ahí que se diga que las estructuras definen nuevos tipos de datos.

Las estructuras en C se pueden copiar, asignar, pasar a funciones, y ser regresadas como valor de retorno de una función; sin embargo, no existe una *relación de orden* predefinida, es decir, no pueden ser comparadas.

Sean e_1 y e_2 dos variables derivadas de la misma estructura, las operaciones:

$$e_1 < e_2, \quad e_1 > e_2, \quad e_1 \leq e_2, \quad e_1 \geq e_2, \quad e_1 = e_2, \quad e_1 \neq e_2 \quad (8.1)$$

son inválidas.

8.2. Abstracción en acción

El segundo mecanismo de abstracción, aunque simple, resulta bastante útil tanto para la abstracción, como para la auto documentación de programas.

La sentencia **typedef** de C se utiliza para *renombrar* tipos de datos existentes, y en armonía con **struct**, constituyen los aliados perfectos para elevar el nivel de abstracción en los programas.

8.2.1. Números racionales

Un **número racional** ($\frac{p}{q}$) se define como el cociente de dos números enteros, donde $p, q \in \mathbb{Z}$.

El lenguaje de programación C no tiene un tipo de datos racional que permita manipular variables de la forma $\frac{p}{q}$, sin embargo, es posible *abstraer* su representación con los tipos de datos existentes

Observe las líneas 6-9 del Ejemplo 8.1, en las cuales se ha definido una estructura de nombre *r* con dos elementos miembro de tipo **int** *p* y *q*. Note que lo que se ha hecho es representar la definición de un número racional utilizando los elementos del lenguaje C.


```

1  /* Abstraccion de numeros racionales (primera version).
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5
6  struct r{
7      int p;
8      int q;
9  };
10
11  typedef struct r RACIONAL;
12
13  int main(){
14      struct r r1;
15      RACIONAL r2;
16      RACIONAL res;
17      char op;
18
19      do{
20          printf("Expresion: ");
21          scanf("%d %c %d %c %d %c %d", &r1.p, &r1.q, &op, &r2.p,
22              &r2.q);
23          if(r1.q == 0 || r2.q == 0)
24              printf("\tEl denominador no puede ser cero!\n");
25      }while(r1.q == 0 || r2.q == 0);
26
27      switch(op){
28          case '+':
29              break;
30          case '-':
31              break;
32          case '*':
33              res.p = r1.p * r2.p;
34              res.q = r1.q * r2.q;
35              break;
36          case '/':
37              res.p = r1.p * r2.q;
38              res.q = r1.q * r2.p;
39              break;
40          default:
41              printf("\aOperador NO valido:\n" "% c"\n", op);
42              return 0;
43      }
44
45      printf("%d/%d %c %d/%d = %d/%d\n",
46          r1.p, r1.q, op, r2.p, r2.q, res.p, res.q);
47
48      return 0;
49  }

```

Ejemplo 8.1: Abstracción de números racionales (primera version)

La plantilla o molde que se ha definido para generar variables que contengan dos números enteros es **struct** *r*. Observe que las variables *p* (línea 7) y *q* (línea 8) se han colocado por separado aunque su tipo de dato (**int**) es

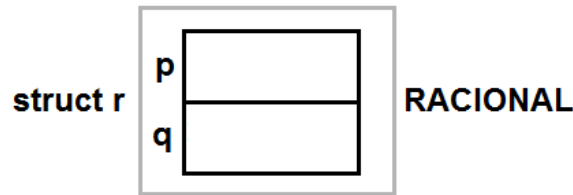


Figura 8.1: Representación de la estructura de las variables de tipo **struct** *r* o *RACIONAL* del Ejemplo 8.1

el mismo, esto es más una cuestión de estilo que una característica inherente a la definición de la estructura.

Por otro lado, note que en la línea 14 se declara la variable *r1* de tipo **struct** *r*. Es hasta ese momento cuando se genera el espacio en memoria necesario para almacenar y gestionar un objeto *r1* de tipo **struct** *r*.

Es momento de mostrar la utilidad de **typedef**. La línea 11 le indica al compilador que de ahí en adelante, la expresión *RACIONAL* será exactamente equivalente a la expresión **struct** *r*, y por ésta razón, la variable *r2* (línea 15) y *res* (línea 16) son idénticas en estructura, forma, y tipo, a la variable *r1* de la línea 14. El **typedef** *renombra* la estructura con un nuevo identificador².

Las variables del tipo **struct** *r* o *RACIONAL* lucen como se muestra en la Figura 8.1.

Ahora bien, el acceso a los elementos miembro de una estructura se realiza a través del **operador punto** (**.**). El acceso a los elementos miembro de una estructura no sigue ningún orden específico, ni está relacionado con el orden en que aparecen dichos elementos en su declaración; el acceso a los elementos miembro está únicamente condicionado a las necesidades específicas del problema en cuestión, y la sintaxis general es:

```
estructura.elemento_miembro;
```

en donde:

- *estructura* es el identificador de una variable de tipo estructura (**struct**).

²Aunque no es una restricción del lenguaje, se recomienda, como estándar de programación, que los identificadores que representan “nuevos” tipos de datos se escriban con letras mayúsculas.

<pre>Expresion: 1/5 # 3/2 Operador NO valido:"#"</pre>	<pre>Expresion: 1/5 * 3/2 1/5 * 3/2 = 3/10</pre>
(a) Operador no válido	(b) Operador válido

Figura 8.2: Salida del Ejemplo 8.1

- *elemento_miembro* es alguno de los elementos miembro de la estructura *estructura*.

La línea 21 muestra la forma de acceder a los elementos miembro de una estructura. Note que el acceso a los elementos miembro es para leer los números racionales $r1$ y $r2$ desde la entrada estándar, que para cada uno se lee su numerador (p) y denominador (q) respectivamente, y que el acceso a los elementos miembro va precedido por el operador `&`.

La cadena con los especificadores de formato de la función **scanf** de la línea 21 (“`%d %*c %d %c %d %*c %d`”), requiere de una explicación para el especificador `%*c`, el cual, le indica a la función **scanf** que ignore cualquier símbolo introducido en esa posición. Note que de hecho no es almacenado, ya que resulta útil y conveniente en la representación de los números racionales, pero no para el almacenamiento, tal y como lo muestra la Figura 8.2, donde se ilustra la forma de introducir números racionales para una posible ejecución del Ejemplo 8.1.

El ciclo **do-while** de las líneas 19-24 realiza una validación básica: el denominador no puede ser cero para ningún número racional.

La sentencia **switch** de las líneas 26-42 determina la operación a realizar. Note que las operaciones de suma y resta no han sido especificadas y se dejan como ejercicio para el lector.

Finalmente, observe que las líneas 31-34 representan la Ecuación 8.6, y que las líneas 35-38 representan la Ecuación 8.7; mientras que las líneas 44-45 presentan el resultado de las operaciones en un formato de salida intuitivo y conveniente para el usuario (vea nuevamente la Figura 8.2).

Las operaciones aritméticas con números racionales se definen a continuación.

Sean r_1 y r_2 dos números racionales, es decir: $r_1, r_2 \in \mathbb{Q}$ definidos como:

$$r_1 = \frac{p_1}{q_1}; \quad p_1, q_1 \in \mathbb{Z}, \quad q_1 \neq 0 \quad (8.2)$$

$$r_2 = \frac{p_2}{q_2}; \quad p_2, q_2 \in \mathbb{Z}, \quad q_2 \neq 0 \quad (8.3)$$

la suma de r_1 y r_2 se define de la siguiente manera:

$$r_1 + r_2 = \frac{p_1}{q_1} + \frac{p_2}{q_2} = \frac{(p_1 * q_2) + (p_2 * q_1)}{q_1 * q_2} \quad (8.4)$$

y la resta de manera análoga:

$$r_1 - r_2 = \frac{p_1}{q_1} - \frac{p_2}{q_2} = \frac{(p_1 * q_2) - (p_2 * q_1)}{q_1 * q_2} \quad (8.5)$$

mientras que la multiplicación está dada por:

$$r_1 * r_2 = \frac{p_1}{q_1} * \frac{p_2}{q_2} = \frac{p_1 * p_2}{q_1 * q_2} \quad (8.6)$$

y la división por:

$$\frac{r_1}{r_2} = \frac{\frac{p_1}{q_1}}{\frac{p_2}{q_2}} = \frac{p_1 * q_2}{q_1 * p_2} \quad (8.7)$$

Evolución

El Ejemplo 8.2 es una evolución del Ejemplo 8.1, pero son en esencia iguales, por lo que no se ahondará más de lo necesario en su descripción, y sólo se destacarán las siguientes diferencias:

```

1  /* Abstraccion de numeros racionales (segunda version).
2  Se ejemplifica el paso de estructuras por valor y referencia,
3  asi como la notacion con operador punto "." y flecha "->".
4  @autor: Ricardo Ruiz Rodriguez
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  typedef struct{
10     int p, q;
11 } RACIONAL;
12
13 char leeExpresion(RACIONAL *, RACIONAL *);
14 RACIONAL hazOperacion(RACIONAL, RACIONAL, char);
15
16 int main() {
17     RACIONAL r1;
18     RACIONAL r2;
19     RACIONAL res;
20     char op;

```

```

21
22     op = leeExpresion(&r1, &r2);
23     res = hazOperacion(r1, r2, op);
24
25     printf(" %d/ %d %c %d/ %d = %d/ %d\n",
26           r1.p, r1.q, op, r2.p, r2.q, res.p, res.q);
27
28     return 0;
29 }
30
31 RACIONAL hazOperacion(RACIONAL r1, RACIONAL r2, char op){
32     RACIONAL r;
33
34     switch(op){
35         case '+':
36             break;
37         case '-':
38             break;
39         case '*':
40             r.p = r1.p * r2.p;
41             r.q = r1.q * r2.q;
42             break;
43         case '/':
44             r.p = r1.p * r2.q;
45             r.q = r1.q * r2.p;
46             break;
47         default:
48             printf("\aOperador NO valido:\n" % c "\n", op);
49             exit(0);
50     }
51
52     return r;
53 }
54
55 char leeExpresion(RACIONAL *r1, RACIONAL *r2){
56     char op;
57
58     do{
59         printf("Expresion: ");
60         scanf(" %d %c %d %c %d %c %d", &r1->p, &r1->q, &op, &r2->p,
&r2->q);
61         if(r1->q == 0 || r2->q == 0)
62             printf("\tEl denominador no puede ser cero!\n");
63     }while(r1->q == 0 || r2->q == 0);
64
65     return op;
66 }

```

Ejemplo 8.2: Abstracción de números racionales (segunda versión)

1. La estructura que modela o representa a los números racionales (líneas 9-11), ha sido simplificada a un renglón y fusionada con el **typedef**. Note que también se ha omitido el identificador del **struct** debido a que la estructura es renombrada desde su definición.

2. Se incluye la función *leeExpresion* (líneas 13 y 55-66), la cual recibe dos argumentos de tipo apuntador a *RACIONAL*. Note que la función encapsula las sentencias para la lectura de datos (líneas 17-24) del Ejemplo 8.1. La función muestra el uso del **operador flecha** (`->`), el cual sirve para acceder a los elementos miembro de una estructura cuando son referidos por medio de una variable de tipo apuntador³.
3. La función *hazOperacion* (líneas 14 y 31-53), la cual recibe dos argumentos de tipo *RACIONAL*, y uno de tipo **char**. La función regresa un valor de tipo *RACIONAL*, mismo que representa el resultado de la operación que se realiza. Observe que la función encapsula las sentencias que determinan y llevan a cabo la operación aritmética (líneas 26-42) del Ejemplo 8.1.⁴

Compruebe que la salida del Ejemplo 8.2 es idéntica a la mostrada en la Figura 8.2 si se procesan los mismos datos.

8.3. Estructuras compuestas

En muchos casos, el modelado de un problema requiere de elementos compuestos:

- El nombre completo de una persona por ejemplo está compuesto por:
 1. Nombre(s).
 2. Primer apellido.
 3. Segundo apellido.

El nombre completo de una persona podría ser un elemento miembro de otra estructura, y si el modelado del problema requiere de dicho nivel de especificación, convendría considerarlo como una entidad aparte.

- Un automóvil está compuesto de varias partes, que a su vez están compuestas por otras tantas:

³Compare la línea 21 del Ejemplo 8.1 con la línea 60 del Ejemplo 8.2.

⁴Note que la línea 49 del Ejemplo 8.2 hace uso de la función **exit**, la cual hace que el programa termine como si llegara a su fin, y es equivalente al *return 0* de la línea 41 del Ejemplo 8.1. La función **exit** se encuentra definida en la biblioteca estándar *stdlib.h* (línea 7 del Ejemplo 8.2).

- Motor (combustión interna de gasolina):
 - Sistema de enfriamiento.
 - Sistema eléctrico.
 - Bujías.
 - Cilindros.
 - Pistones.
 - Válvulas.
 - Levas.
 - Inyectores.
 - etcétera (bastante amplio por cierto).
 - Puertas:
 - Sistema de apertura y cierre.
 - Sistema eléctrico.
 - Cristales.
 - Seguros.
 - Barras de protección.
 - etcétera.
 - etcétera.
- Los elementos geométricos son un excelente ejemplo de composición:
 - Un vértice en el espacio bidimensional está compuesto por un par ordenado de coordenadas o puntos (x, y) .
 - Un triángulo en el espacio bidimensional está definido por un conjunto de tres vértices.
 - Un cuadrado está compuesto por cuatro vértices equidistantes entre sí.

En el lenguaje de programación C, no es posible definir una estructura dentro de otra estructura, lo cual quiere decir que no se pueden definir estructuras anidadas, sin embargo, sí es posible definir estructuras cuyos elementos miembro sean otras estructuras, es decir, elementos miembro cuyo

tipo de dato es algún otro tipo de estructura distinta de ella misma⁵, siempre y cuando ésta última haya sido definida antes que la primera.

El Ejemplo 8.3 muestra la definición y el uso de estructuras compuestas para un par de elementos geométricos definidos en el espacio bidimensional: el vértice y el triángulo.

```

1  /* Programa que muestra la abstraccion en la definicion
2     y uso de estructuras compuestas.
3     @autor: Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6
7  typedef struct{
8      int x;
9      int y;
10 } VERTICE;
11
12 typedef struct{
13     VERTICE v1;
14     VERTICE v2;
15     VERTICE v3;
16 } TRIANGULO;
17
18 void leeVertice(VERTICE *);
19 void leeTriangulo(TRIANGULO *);
20 void imprimeTriangulo(TRIANGULO);
21
22 int main(){
23     TRIANGULO t;
24
25     leeTriangulo(&t);
26     imprimeTriangulo(t);
27
28     return 0;
29 }
30
31 void leeTriangulo(TRIANGULO *t){
32     printf(" Vertices del triangulo:\nVertice1:\n");
33     leeVertice(&t->v1);
34     printf(" Vertice2:\n");
35     leeVertice(&t->v2);
36     printf(" Vertice3:\n");
37     leeVertice(&t->v3);
38 }
39
40 void leeVertice(VERTICE *v){
41     printf("Proporcione el vertice (Ejemplo: 3, 2): ");
42     scanf("%d, %d", &v->x, &v->y);
43 }

```

⁵Existe una excepción: se puede definir una variable que haga referencia a una estructura del mismo tipo, es decir, una variable de tipo apuntador a la estructura que la define, lo cual es útil para la construcción de estructuras de datos como listas, pilas, colas, árboles, etcétera, pero estos conceptos quedan fuera del alcance de este libro y no se tratarán aquí.


```

44
45 void imprimeTriangulo(TRIANGULO t){
46     printf(" Vertice1: (%d, %d)\n", t.v1.x, t.v1.y);
47     printf(" Vertice2: (%d, %d)\n", t.v2.x, t.v2.y);
48     printf(" Vertice3: (%d, %d)\n", t.v3.x, t.v3.y);
49 }

```

Ejemplo 8.3: Estructuras compuestas

La definición de las estructuras *VERTICE* y *TRIANGULO* ocurren en las líneas 7-10 y 12-16 respectivamente. Note cómo los elementos miembro de la estructura *TRIANGULO* son del tipo *VERTICE*, por lo que la estructura *VERTICE* ha tenido que definirse antes que la estructura *TRIANGULO*.

La función principal **main** declara, en la línea 23, una variable *t* de tipo *TRIANGULO*, lee dicho triángulo (línea 25), y presenta el triángulo que acaba de ser leído en la salida estándar (línea 26).

Por otro lado, la función *leeVertice* (líneas 40 - 43) recibe una referencia a una variable de tipo *VERTICE*, la cual es necesaria, debido a que la función es para la lectura de datos. Observe una vez más el uso del operador flecha (línea 42) para la lectura de datos de los elementos miembro referidos por *v*; y que la cadena de especificación de formato de entrada ("%d,%d") contiene una coma (,), la cual no tiene otro fin más allá del estético en base al formato de entrada sugerido, tal y como se muestra en la Figura 8.3.

La función *leeTriangulo* (líneas 31 - 38) por otra parte, se basa en la función *leeVertice* descrita con anterioridad para leer los tres vértices que conforman un triángulo. Note que la función también recibe una referencia almacenada en *t*, y que dicha referencia es utilizada en las líneas 33, 35 y 37 para pasar a su vez la referencia respectiva al vértice correspondiente a la función *leeVertice*. Asegúrese de comprender esto.

Por último, la función *imprimeTriangulo* (líneas 45 - 49) imprime en la salida estándar, los vértices del triángulo utilizando el operador punto, para acceder, de izquierda a derecha, a los elementos miembro de la estructura compuesta:

- Línea 46: para el triángulo *t*, de su vértice *v1*, el valor correspondiente a *x* (*t.v1.x*); y para el triángulo *t*, de su vértice *v1*, el valor correspondiente a *y* (*t.v1.y*).
- Línea 47: para el triángulo *t*, de su vértice *v2*, el valor correspondiente a *x* (*t.v2.x*); y para el triángulo *t*, de su vértice *v2*, el valor correspondiente a *y* (*t.v2.y*).

```

Vertices del triangulo:
Vertice1:
Proporcione el vertice (Ejemplo: 3, 2): 0, 0
Vertice2:
Proporcione el vertice (Ejemplo: 3, 2): 10, 0
Vertice3:
Proporcione el vertice (Ejemplo: 3, 2): 5, 15
Vertice1: (0, 0)
Vertice2: (10, 0)
Vertice3: (5, 15)

```

Figura 8.3: Salida del Ejemplo 8.3

- Línea 48: para el triángulo t , de su vértice $v3$, el valor correspondiente a x ($t.v3.x$); y para el triángulo t , de su vértice $v3$, el valor correspondiente a y ($t.v3.y$).

8.4. Arreglos de estructuras

Las estructuras, al igual que los tipos de datos básicos de C, también pueden ser agrupadas en arreglos, y el Ejemplo 8.4 es muestra de ello, el cual modela, por medio de un arreglo (línea 16), y una estructura (líneas 10 - 13), un directorio de contactos con los datos elementales.

El ciclo **for** de las líneas 19-25, recorre el arreglo *contactos* para su lectura. Note que cada elemento del arreglo *contactos* es una entidad de tipo *CONTACTO*, y que por lo tanto, se leen de la entrada estándar sus dos elementos miembro: *nombre* (línea 21), y *telefono* (línea 23), a través de la función **gets**.

Por otro lado, el ciclo **for** de las líneas 27 - 31, realiza un recorrido del arreglo *contactos* para su impresión en la salida estándar, de manera análoga a lo descrito en el párrafo anterior. La Figura 8.4 muestra una posible salida del Ejemplo 8.4.

El nivel de detalle para el directorio de contactos puede ser tan elaborado como se requiera, el Ejemplo 8.4 es sólo una muestra del uso de un arreglo de estructuras, el cual puede ser generalizado y extendido hacia otros tipos de datos.

Los arreglos de estructuras siguen las mismas reglas que se analizaron y presentaron para los arreglos en el Capítulo 6. En la sección de ejercicios

tendrá la oportunidad de poner en práctica sus conocimientos, aplicados ahora a un arreglo de estructuras.

```
1  /* Programa que muestra la definicion , uso y recorrido
2     de un arreglo de estructuras .
3     @autor: Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #define N 3
7  #define TAMNOM 50
8  #define TAMTEL 20
9
10 typedef struct{
11     char nombre[TAMNOM];
12     char telefono [TAMTEL];
13 } CONTACTO;
14
15 int main(){
16     CONTACTO contactos [N];
17     int i;
18
19     for(i = 0; i < N; i++){
20         printf("%d.\tNombre: ", i+1);
21         gets(contactos[i].nombre);
22         printf("\tTelefono: ");
23         gets(contactos[i].telefono);
24         putchar('\n');
25     }
26     printf("Lista de contactos:\n");
27     for(i = 0; i < N; i++){
28         printf(" %d.  %s\n", i+1, contactos[i].nombre);
29         printf("Tel.  %s\n", contactos[i].telefono);
30         putchar('\n');
31     }
32     return 0;
33 }
```

Ejemplo 8.4: Arreglo de estructuras

Por otro lado, en el Capítulo 9 se retomará el Ejemplo 8.4, y se relacionará su uso con el almacenamiento de los datos en un archivo de texto.

8.5. Abstracción de matrices

En la Sección 7.4.1 se presentó, en el Ejemplo 7.7, una biblioteca de funciones para la creación y manipulación de matrices utilizando asignación dinámica de memoria. En esta sección se retomarán las ideas presentadas en las secciones 7.4.1 y 8.2.1, para elevar el nivel de abstracción en la representación de matrices, para poder crear y manipular matrices de números racionales.

```

1.      Nombre: Ricardo Ruiz Rodriguez
      Telefono: 222 22 12 1974

2.      Nombre: Ricardo Ruiz Sanchez
      Telefono: 222 20 04 2007

3.      Nombre: Bruno Ruiz Sanchez
      Telefono: 222 18 09 2008

Lista de amigos:
1. Ricardo Ruiz Rodriguez
Tel. 222 22 12 1974

2. Ricardo Ruiz Sanchez
Tel. 222 20 04 2007

3. Bruno Ruiz Sanchez
Tel. 222 18 09 2008

```

Figura 8.4: Salida del Ejemplo 8.4

```

1  /* Biblioteca de funciones que abstrae e implementa funciones
2     para trabajar con matrices de numeros racionales.
3     @autor Ricardo Ruiz Rodriguez
4  */
5  #include <stdlib.h>
6
7  typedef struct{
8     int p, q;
9  } RACIONAL;
10
11 typedef RACIONAL ** MATRIZ_RACIONAL;
12
13 void liberaMatriz(MATRIZ_RACIONAL matriz, int m){
14     int i;
15
16     for(i = 0; i < m; i++)
17         free(matriz[i]);
18     free(matriz);
19     matriz = NULL;
20 }
21
22 MATRIZ_RACIONAL creaMatriz(int m, int n){
23     MATRIZ_RACIONAL matriz;
24     int i;
25
26     matriz = (MATRIZ_RACIONAL) malloc(m * sizeof(RACIONAL *));

```

```

27     if(matriz != NULL)
28         for(i = 0; i < m; i++){
29             matriz[i] = (RACIONAL *) malloc(n * sizeof(RACIONAL *));
30             if (matriz[i] == NULL)
31                 liberaMatriz(matriz, m);
32         }
33     return matriz;
34 }
35
36 void leeMatriz(MATRIZ_RACIONAL matriz, int m, int n, char *c){
37     int i, j;
38
39     for(i = 0; i < m; i++){
40         printf("Renglon %d/%d:\n", i, m - 1);
41         for(j = 0; j < n; j++){
42             printf("%s[%d][%d] = ", c, i, j);
43             scanf("%d %c %d", &matriz[i][j].p, &matriz[i][j].q);
44         }
45         putchar('\n');
46     }
47 }
48
49 void imprimeMatriz(MATRIZ_RACIONAL matriz, int m, int n, char *c){
50     int i, j;
51
52     putchar('\n');
53     for(i = 0; i < m; i++){
54         for(j = 0; j < n; j++)
55             printf("%s[%d][%d] = %d/%d ", c, i, j,
56                 matriz[i][j].p, matriz[i][j].q);
57         putchar('\n');
58     }
59 }
60
61 RACIONAL sumaRacional(RACIONAL r1, RACIONAL r2){
62     RACIONAL r3;
63
64     r3.p = (r1.p * r2.q) + (r2.p * r1.q);
65     r3.q = r1.q * r2.q;
66
67     return r3;
68 }
69
70 MATRIZ_RACIONAL suma(MATRIZ_RACIONAL a, MATRIZ_RACIONAL b, int m, int n){
71     MATRIZ_RACIONAL c;
72     int i, j;
73
74     if((c = creaMatriz(m, n)) != NULL)
75         for(i = 0; i < m; i++)
76             for(j = 0; j < n; j++)
77                 c[i][j] = sumaRacional(a[i][j], b[i][j]);
78     return c;
79 }

```

Ejemplo 8.5: Biblioteca de funciones para matrices de números racionales

El Ejemplo 8.5 es esencialmente igual al Ejemplo 7.7, tómese el tiempo de compararlos línea a línea y función por función, aquí sólo se remarcarán los aspectos más sobresalientes, dado que la creación de matrices ya no es tema ni de la sección ni del capítulo.

La clave de la abstracción ocurre en las líneas 7 - 11 del Ejemplo 8.5, con la definición de la estructura *RACIONAL* (líneas 7 - 9), y la re definición del tipo doble apuntador a *RACIONAL* por *MATRIZ_RACIONAL* (línea 11).

La re definición del tipo doble apuntador a *RACIONAL* por el nuevo tipo de dato *MATRIZ_RACIONAL*, hace que ahora las funciones, en lugar de recibir un doble apuntador a entero (como en el Ejemplo 7.7), reciban una variable de tipo *MATRIZ_RACIONAL*, lo cual hace, desde su lectura e interpretación, un enfoque mucho más conveniente para la programación, reforzando, no sólo la naturaleza de los tipos de datos involucrados, sino también la auto documentación del código fuente.

```

1  /* Uso de matrices de numeros racionales definidas en
2     la biblioteca de funciones matrizRacional.h.
3     @autor: Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include "matrizRacional.h"
7
8  int main(){
9      MATRIZ_RACIONAL matA, matB, matC;
10     int m, n;
11
12     do{
13         printf("Escribe m y n (ambos mayores a 1): ");
14         scanf("%d %d", &m, &n);
15     }while(m < 2 || n < 2);
16
17     matA = creaMatriz(m, n);
18     matB = creaMatriz(m, n);
19
20     if(matA != NULL && matB != NULL){
21         leeMatriz(matA, m, n, "A");
22         leeMatriz(matB, m, n, "B");
23         matC = suma(matA, matB, m, n);
24         imprimeMatriz(matA, m, n, "A");
25         imprimeMatriz(matB, m, n, "B");
26         imprimeMatriz(matC, m, n, "C");
27         liberaMatriz(matA, m);
28         liberaMatriz(matB, m);
29         liberaMatriz(matC, m);
30     }
31
32     return 0;
33 }
```

Ejemplo 8.6: Uso de matrices de números racionales

Observe cómo todos los tipos de datos han sido ajustados en correspondencia al nuevo enfoque de abstracción, tal y como se muestra en las líneas 13, 22, 23, 26, 29, 36, 49, 70 y 71.

Por otro lado, los datos almacenados en la matriz son ahora del tipo *RACIONAL*, por lo que cada uno de ellos tiene dos elementos miembro: p y q , los cuales son considerados tanto en la lectura (línea 43), como en la salida de datos (línea 56).

Por último, la función *suma* de las líneas 70 - 79 se apoya (línea 77), de la función *sumaRacional* de las líneas 61 - 68 para realizar la suma de dos números racionales, ya que la suma de racionales no es la suma tradicional de elemento a elemento, sino la suma expresada en la Ecuación 8.4.

El Ejemplo 8.6 es análogo al Ejemplo 7.8, tómese también el tiempo para compararlos y comprender sus diferencias. Note cómo la abstracción definida en la biblioteca de funciones *matrizRacional.h*, es utilizada en la línea 9 del Ejemplo 8.6, mostrando con ello que el uso apropiado de bibliotecas de funciones, también refuerza la abstracción a través de la modularidad.

Finalmente, una posible salida para el Ejemplo 8.6 se muestra en la Figura 8.5.

```

Escribe m y n (ambos mayores a 1): 1 1
Escribe m y n (ambos mayores a 1): 0 3
Escribe m y n (ambos mayores a 1): 2 1
Escribe m y n (ambos mayores a 1): 2 2
Renglon 0/1:
A[0][0] = 1/2
A[0][1] = 1/3

Renglon 1/1:
A[1][0] = 1/4
A[1][1] = 1/5

Renglon 0/1:
B[0][0] = 1/5
B[0][1] = 1/4

Renglon 1/1:
B[1][0] = 1/3
B[1][1] = 1/2

A[0][0] = 1/2 A[0][1] = 1/3
A[1][0] = 1/4 A[1][1] = 1/5

B[0][0] = 1/5 B[0][1] = 1/4
B[1][0] = 1/3 B[1][1] = 1/2

C[0][0] = 7/10 C[0][1] = 7/12
C[1][0] = 7/12 C[1][1] = 7/10

```

Figura 8.5: Salida del Ejemplo 8.6

8.6. Ejercicios

1. Compare la estructura del Ejemplo 8.1 con la estructura general de un programa en C descrita en el Ejemplo 1.1 del Capítulo 1.
2. En base al programa del Ejemplo 8.1 y a lo expuesto en el texto de la Sección 8.2.1, complete las operaciones de suma y resta para las sentencias **case** de las líneas 27 y 29 respectivamente, tomando en cuenta las Ecuaciones 8.2, 8.3, 8.4 y 8.5.

Nota: Considere el caso de la siguiente operación o una similar:

$$1/2 - 1/2$$

¿Qué pasaría?, ¿cómo manejaría esta situación?

Sugerencia: Si el resultado de alguna operación genera un numerador con cero, no presente el resultado 0/4 por ejemplo, sino sólo 0. Utilice una sentencia **if/else** en la presentación de datos.

3. Reescriba la declaración de variables *r1*, *r2* y *res* de las líneas 17 - 19 del Ejemplo 8.2, en una sola línea.
4. Considere la función *leeExpresion* del Ejemplo 8.2. La expresión de la línea 61:

```
r1 -> q == 0
```

puede reescribirse con notación de apuntadores como:

```
(*r1).q == 0
```

Reescriba, en su respectiva notación de apuntador en otro programa distinto, todas las expresiones que utilizan el operador flecha en la función *leeExpresion*, y pruebe que dicho programa funciona exactamente igual.

5. Reescriba la función *hazOperacion* del Ejemplo 8.2 para que utilice funciones para realizar la suma, resta, producto y división de números racionales, utilice las siguientes firmas:

a) **RACIONAL** suma(**RACIONAL** r1, **RACIONAL** r2)

- b) **RACIONAL resta(RACIONAL r1, RACIONAL r2)**
- c) **RACIONAL producto(RACIONAL r1, RACIONAL r2)**
- d) **RACIONAL division(RACIONAL r1, RACIONAL r2)**

6. Escriba una función que dado un número racional, regrese si es posible, dicho número racional simplificado, e incorpórela al Ejemplo 8.2. La función deberá tener la siguiente firma:

RACIONAL simplifica(RACIONAL r)

7. Reescriba el Ejemplo 8.2 para que utilice un arreglo de apuntadores a funciones en lugar de una estructura **switch**.
8. Sean c_1 y c_2 dos números complejos es decir: $c_1, c_2 \in \mathbb{C}$ definidos de la siguiente manera:

$$c_1 = (a + bi) \text{ y } c_2 = (c + di)$$

donde $a, b, c, d \in \mathbb{R}$.

La suma de c_1 y c_2 se define como:

$$c_1 + c_2 = (a + bi) + (c + di) = (a + c) + (b + d)i \quad (8.8)$$

y la resta de manera análoga:

$$c_1 - c_2 = (a + bi) - (c + di) = (a - c) + (b - d)i \quad (8.9)$$

mientras que la multiplicación está dada por:

$$c_1 * c_2 = (a + bi) * (c + di) = (ac - bd) + (ad + bc)i \quad (8.10)$$

La división es un poco más elaborada debido a que se racionaliza el denominador, es decir, se multiplica el numerador y el denominador por el conjugado del denominador⁶:

$$\frac{c_1}{c_2} = \frac{(a + bi)}{(c + di)} = \frac{(a + bi) * (c - di)}{(c + di) * (c - di)} = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2} \quad (8.11)$$

⁶El conjugado de un número complejo se obtiene cambiando el signo de su componente imaginaria, es decir: si $c = a + bi$ es un número complejo, su conjugado está dado por $\bar{c} = a - bi$.

Escriba un programa que permita representar números complejos, así como sus respectivas operaciones aritméticas presentadas en las Ecuaciones 8.8, 8.9, 8.10 y 8.11.

9. Extienda el Ejemplo 8.3 para que realice lo siguiente⁷:

- a) Presente un menú que permita al usuario trabajar con tres tipos de figuras geométricas: triángulos, rectángulos y cuadrados. Los rectángulos y los cuadrados serán ortogonales, y sus lados paralelos a los ejes x y y respectivamente.
- b) Para el caso de los triángulos, considerar:
 - 1) Si los vértices introducidos constituyen o no un triángulo. Si los vértices no conforman un triángulo se reporta, y se repite la lectura de los mismos hasta que constituyan un triángulo.
 - 2) Determinar el tipo de triángulo leído: isósceles, equilátero o escaleno.
- c) Represente, por medio de una estructura compuesta, un cuadrado, en cuyo caso deberá:
 - 1) Determinar si los vértices introducidos constituyen o no un cuadrado. Si los vértices no conforman un cuadrado se reporta, y se repite la lectura de los mismos hasta que constituyan un cuadrado.
 - 2) Dado un punto en el espacio bidimensional (vértice), determinar si dicho punto está dentro, fuera, o sobre el último cuadrado introducido.
- d) Represente, por medio de una estructura compuesta, un rectángulo, en cuyo caso deberá:
 - 1) Determinar si los vértices introducidos constituyen o no un rectángulo. Si los vértices no conforman un rectángulo se reporta, y se repite la lectura de los mismos hasta que constituyan un rectángulo.
 - 2) Dado un punto en el espacio bidimensional (vértice), determinar si dicho punto está dentro, fuera, o sobre el último rectángulo introducido.

⁷Para este ejercicio, asuma que todos los vértices serán introducidos en el sentido de las manecillas del reloj.

El menú deberá tener el siguiente formato:

```
****      M E N U      ****
[1] Triangulo.
[2] Cuadrado.
[3] Rectangulo.
[4] Relacion punto cuadrado.
[5] Relacion punto rectangulo.
[6] Salir.
```

Opcion?:

10. Reescriba el Ejemplo 8.4, para que tanto la lectura de datos de la entrada estándar, como la presentación de los mismos en la salida estándar, se lleve a cabo por medio de funciones, para ello, utilice las siguientes firmas de funciones:

- **void llenaDirectorio(CONTACTO contactos[], int n)**
- **void imprimeDirectorio(CONTACTO *contactos, int n)**

donde *contactos* representa el directorio de contactos y *n* el número de contactos a leer o presentar respectivamente. Note que las expresiones *contactos[]* y **contactos* son equivalentes.

11. Escriba una función que permita ordenar, en base al nombre, el directorio de contactos del Ejemplo 8.4. Utilice el ordenamiento por burbuja y la siguiente firma de función:

void ordena(CONTACTO *contactos, int n)

12. Complete la biblioteca de funciones del Ejemplo 8.5 con la función *simplifica* realizada en el Ejercicio 6, de tal forma que los valores almacenados en las matrices estén siempre reducidos a su expresión mínima, es decir, que estén simplificados.
13. Escriba un programa que defina y pruebe una función con la siguiente firma:

**MATRIZ_RACIONAL resta(MATRIZ_RACIONAL a,
MATRIZ_RACIONAL b, int m, int n)**

La función deberá calcular la resta de la matriz a menos la matriz b , de manera análoga a como lo hace la función *suma* del Ejemplo 8.5. Tome en cuenta que necesitará de una función *restaRacional*⁸ análoga a la función *sumaRacional* del Ejemplo 8.5.

14. Agregue la función *resta* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.
15. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad (8.12)$$

donde $a_{i,j} \in \mathbb{Q}$.

La matriz transpuesta de A , denotada por A^T , se define como:

$$(A^T)_{ij} = A_{ji}, \quad 0 \leq i \leq n-1, \quad 0 \leq j \leq m-1 \quad (8.13)$$

es decir:

$$\begin{pmatrix} a_{0,0} & \dots & a_{m-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,n-1} & \dots & a_{m-1,n-1} \end{pmatrix}_{n \times m} \quad (8.14)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

**void imprimeTranspuesta(MATRIZ_RACIONAL a, int m,
int n)**

La función deberá **imprimir** en la salida estándar, la matriz transpuesta de a , siguiendo los lineamientos expresados en las Ecuaciones 8.13 y 8.14.

⁸Vea la ecuación 8.5.

16. Agregue la función *imprimeTranspuesta* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.
17. Escriba un programa que defina y pruebe una función con la siguiente firma:

MATRIZ_RACIONAL *transpuesta*(**MATRIZ_RACIONAL** *a*,
int *m*, **int** *n*)

La función deberá **regresar** la matriz transpuesta de *a*, siguiendo los lineamientos expresados en las Ecuaciones 8.13 y 8.14.

18. Agregue la función *transpuesta* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.
19. Sea *A* una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{n \times n} \quad (8.15)$$

donde $a_{i,j} \in \mathbb{Q}$.

Se dice que la matriz *A* es simétrica respecto a la diagonal principal si:

$$a_{ij} = a_{ji}, \forall i \neq j \quad (8.16)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

int *esSimetrica*(**MATRIZ_RACIONAL** *a*, **int** *n*)

La función deberá determinar si la matriz *a*, es o no simétrica en base a lo expuesto en la Ecuación 8.16.

20. Agregue la función *esSimetrica* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.

21. Sean A y B dos matrices definidas de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}_{m \times n} \quad B = \begin{pmatrix} b_{0,0} & \dots & b_{0,l-1} \\ \vdots & \ddots & \vdots \\ b_{n-1,0} & \dots & b_{n-1,l-1} \end{pmatrix}_{n \times l} \quad (8.17)$$

donde $a_{i,j}, b_{i,j} \in \mathbb{Q}$.

El producto de $A_{m \times n} \times B_{n \times l}$ denotado por $C_{m \times l}$, se define como:

$$C_{ij} = \sum_{r=0}^{n-1} a_{ir} b_{rj} \quad (8.18)$$

es decir:

$$\begin{pmatrix} a_{0,0}b_{0,0} + \dots + a_{0,n-1}b_{n-1,0} & \dots & a_{0,0}b_{0,l-1} + \dots + a_{0,n-1}b_{n-1,l-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}b_{0,0} + \dots + a_{m-1,n-1}b_{n-1,0} & \dots & a_{m-1,0}b_{0,l-1} + \dots + a_{m-1,n-1}b_{n-1,l-1} \end{pmatrix}_{m \times l} \quad (8.19)$$

Escriba un programa que defina y pruebe una función con la siguiente firma:

**MATRIZ_RACIONAL producto(MATRIZ_RACIONAL a,
MATRIZ_RACIONAL b, int m, int n, int l)**

de tal forma que la función calcule el producto de las matrices a y b regresando su resultado. Las matrices tienen la forma descrita en 8.17, pero no olvide que los elementos de las matrices son números racionales, y que por lo tanto lo expresado en las Ecuaciones 8.18 y 8.19, deberá ajustarse también a los lineamientos dados en las ecuaciones 8.4 y 8.6, respecto a la suma y producto respectivamente, de números racionales.

22. Agregue la función *producto* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.

23. Sea A una matriz definida de la siguiente manera:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \dots & a_{n-1,n-1} \end{pmatrix}_{n \times n} \quad (8.20)$$

donde $a_{i,j} \in \mathbb{Q}$.

La matriz inversa de A , denotada por A^{-1} , se define como:

$$A \times A^{-1} = A^{-1} \times A = I \quad (8.21)$$

donde I es la matriz identidad.

Escriba un programa que calcule, si existe, la matriz inversa A^{-1} de la matriz A , por el método de Gauss-Jordan:

$$[A|I] \xrightarrow{\text{Gauss-Jordan}} [I|A^{-1}] \quad (8.22)$$

Utilice la siguiente firma para la función:

**MATRIZ_RACIONAL inversa(MATRIZ_RACIONAL a,
int n)**

Si la matriz inversa de a no existe, la función deberá regresar *NULL*.

24. Agregue la función *inversa* del ejercicio anterior, a la biblioteca de funciones del Ejemplo 8.5.
25. En base a lo expuesto en los Ejemplos 8.5 y 8.6, y al Ejercicio 8, escriba un programa que permita representar matrices de números complejos (\mathbb{Q}).
26. Genere una biblioteca de funciones como la del Ejemplo 8.5, que permita manipular matrices de números complejos (\mathbb{Q}). La biblioteca deberá contener todas las funciones presentadas para los números racionales del Ejemplo 8.5, más las funciones generadas en los ejercicios anteriores respecto a números racionales pero para números complejos, es decir: *resta*, *imprimeTranspuesta*, *transpuesta*, *esSimetrica*, *producto*, *inversa*, etc.
27. Investigue y estudie el concepto de **union** en C. Las uniones (**union**) son parecidas en su estructura y manipulación a las estructuras (**struct**) pero son esencialmente diferentes, debido a que en las uniones, los elementos miembro que la conforman *comparten* su espacio de memoria, mientras que en las estructuras, cada elemento miembro tiene su propio espacio de memoria independiente.

Capítulo 9

Archivos

Los programas desarrollados hasta ahora, han utilizado a la memoria principal tanto para su ejecución, como para el almacenamiento de los datos que administran, sin embargo, una vez que los procesos terminan su ejecución, los datos leídos, procesados y utilizados por ellos desaparecen, y en muchas situaciones es conveniente, tanto preservar los datos, como obtener grandes cantidades de datos como entrada para los procesos. Para estos casos, el manejo de archivos resulta un elemento clave.

9.1. Definición y conceptos

Un **archivo** es, de manera general, un conjunto de *bytes* relacionados entre sí, y referidos por un nombre.

Dependiendo del esquema de almacenamiento utilizado, el conjunto de *bytes* puede estar agrupado en sectores, como en el caso de los discos duros por ejemplo. Los sectores pueden ser de distintos tamaños, y dependen tanto de la geometría del disco, como de la configuración del Sistema Operativo.

Los archivos son el esquema más conveniente de preservación de datos más allá de la vida de un proceso, y también sirven como medio de comunicación entre los procesos que se ejecutan en una computadora.

En la sección 2.3, se mencionaron los tres tipos de archivos o flujos asociados a los procesos: *stdin*, *stdout* y *stderr*. En este capítulo aprenderá cómo asociar más archivos, representados por su respectivo flujo, a sus programas.

9.2. Archivos de texto

En C, los archivos están representados por flujos (*streams*) de *bytes*, por esta razón, cuando un archivo se abre, le es asociado un flujo a dicho archivo.

C no impone o establece algún tipo de estructura específica para los archivos, la estructura asociada a cada archivo es responsabilidad del programador.

9.2.1. Archivos de acceso secuencial

Un **archivo de acceso secuencial** es un archivo de texto, en el que los datos se van almacenando, uno detrás de otro, sin tomar en consideración su tipo de datos. El acceso a los datos en este tipo de archivos se realiza de manera secuencial, es decir, no se puede acceder de manera directa a algún dato específico dentro del archivo, sin antes haber pasado por todos los datos anteriores a él.

Los Ejemplos 9.1 y 9.2 muestran el uso de archivos de acceso secuencial para almacenamiento y lectura de datos respectivamente, y se describen en las secciones siguientes.

Almacenamiento de datos

En la sección 8.4, se trabajó con un programa que administraba un directorio de contactos, en esta sección se retomará el planteamiento utilizado en el Ejemplo 8.4, para almacenar los datos del directorio de contactos en un archivo.

El Ejemplo 9.1 muestra la forma de utilizar un archivo de acceso secuencial para el almacenamiento de datos. La explicación del ejemplo se centrará en la función *guardaDirectorio* (líneas 38 - 53).

La línea 39 declara una variable de tipo apuntador a *FILE* de nombre *archivoPtr*. *FILE* es una estructura (**struct**) que contiene información indispensable para el control del archivo o flujo, incluyendo, entre otras cosas, un apuntador al *buffer* donde se almacenarán los datos que serán enviados o leídos del archivo, el indicador de posición del archivo, indicadores de estado del archivo, etc.

```

1  /* Uso de archivos secuenciales para el almacenamiento de datos.
2  @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>

```

```

5 #define N 3
6 #define TAMNOM 50
7 #define TAMTEL 20
8
9 typedef struct{
10     char nombre[TAMNOM];
11     char telefono[TAMTEL];
12 } CONTACTO;
13
14 void llenaDirectorio(CONTACTO *, int);
15 void guardaDirectorio(CONTACTO *, int);
16
17 int main(){
18     CONTACTO contactos[N];
19
20     llenaDirectorio(contactos, N);
21     guardaDirectorio(contactos, N);
22
23     return 0;
24 }
25
26 void llenaDirectorio(CONTACTO *listaContactos, int n){
27     int i;
28
29     for(i = 0; i < n; i++){
30         printf("%d.\tNombre: ", i+1);
31         gets(listaContactos[i].nombre);
32         printf("\tTelefono: ");
33         gets(listaContactos[i].telefono);
34         putchar('\n');
35     }
36 }
37
38 void guardaDirectorio(CONTACTO *listaContactos, int n){
39     FILE *archivoPtr;
40     int i;
41
42     printf("Guardando contactos en el archivo...\n");
43     if((archivoPtr = fopen("contactos.txt", "a")) == NULL)
44         printf("El archivo no pudo ser abierto...\n");
45     else{
46         for(i = 0; i < n; i++){
47             fprintf(archivoPtr, "%s\n", listaContactos[i].nombre);
48             fprintf(archivoPtr, "%s\n", listaContactos[i].telefono);
49         }
50         fclose(archivoPtr);
51         printf("Listo!\n");
52     }
53 }

```

Ejemplo 9.1: Uso de un archivo secuencial para el almacenamiento de datos

La función **fopen** (línea 43), abre el archivo especificado en la cadena de su primer argumento¹, lo asocia con un flujo, y regresa un apuntador a

¹La cadena puede incluir la ruta completa de acceso al archivo, si no se especifica una

“r”	leer (<i>read</i>): abre el archivo para lectura de datos. El archivo debe existir.
“w”	escribir (<i>write</i>): crea un archivo vacío para el almacenamiento de datos. Si el archivo existe, su contenido se descarta, y el archivo es tratado como si fuera un nuevo archivo.
“a”	agregar (<i>append</i>): abre un archivo para el almacenamiento de datos al final del mismo. Si el archivo no existe, se crea.
“r+”	leer/actualizar (<i>read/update</i>): abre el archivo para actualización (lectura y escritura) de datos. El archivo debe existir.

Tabla 9.1: Modos básicos de apertura de archivos para la función **fopen**

la estructura *FILE* que representa dicho flujo, mismo que será utilizado en accesos posteriores a través de la variable *archivoPtr*.

Las operaciones que se habilitarán sobre el flujo, así como la forma en que éstas se llevan a cabo, son especificadas en la cadena del segundo argumento: el **modo** de apertura. La Tabla 9.1 muestra los modos básicos de apertura utilizados por la función **fopen**.

Con los modos de apertura especificados en la Tabla 9.1, el archivo es abierto y tratado como un archivo de texto. Para que el archivo pueda ser tratado como un archivo binario, se debe agregar el símbolo *b* (*binary*) a la cadena de modo de apertura, tal y como se muestra en la 15 del Ejemplo 9.7, la línea 20 del Ejemplo 9.5, y la línea 25 del Ejemplo 9.6, de los cuales se hablará más adelante.

Continuando con el Ejemplo 9.1, note que la apertura del archivo ha sido solicitada en modo *agregar* y no en modo *escribir*². Si la función **fopen** pudo abrir el archivo, regresará, como ya se mencionó con anterioridad, una referencia al flujo que representa dicho archivo, en caso contrario, regresará un apuntador nulo, y ésta situación debe ser siempre verificada, como se muestra en la sentencia **if** de la línea 43.

ruta, se busca el archivo en el directorio de trabajo actual.

²Este modo de apertura debe utilizarse con mucha precaución, ya que se pueden perder archivos completos, debido a que se eliminan todos los datos de un archivo existente, aunque estos no hayan sido creados por un programa en C.

```
1.      Nombre: Ricardo Ruiz Rodriguez
        Telefono: 222 22 12 1974

2.      Nombre: Ricardo Ruiz Sanchez
        Telefono: 222 20 04 2007

3.      Nombre: Bruno Ruiz Sanchez
        Telefono: 222 18 09 2008

Lista de amigos:
1. Ricardo Ruiz Rodriguez
Tel. 222 22 12 1974

2. Ricardo Ruiz Sanchez
Tel. 222 20 04 2007

3. Bruno Ruiz Sanchez
Tel. 222 18 09 2008
```

Figura 9.1: Salida del Ejemplo 9.1

Una vez que el archivo ha sido abierto, almacenar datos en él es tan simple como mandar datos a la salida estándar, la función **fprintf** (líneas 47 y 48), es análoga a la función **printf**, por lo que su uso debe ser más que natural. La única diferencia radica en su primer parámetro, el cual es un apuntador al flujo que se utilizará para enviar los datos³. Note que el flujo al que se están enviando los datos es precisamente el apuntador a *FILE* *archivoPtr*, el cual es el *medio de acceso* para el archivo “*contactos.txt*”⁴.

Por último, la función **fclose** (línea 50) cierra el archivo asociado al flujo representado por *archivoPtr*, y elimina también dicha asociación. Una posible salida para el programa del Ejemplo 9.1 se muestra en la Figura 9.1.

Lectura de datos

En la sección anterior se utilizó un archivo de acceso secuencial para almacenar datos en él, en esta sección se hará uso de dicho archivo para

³Si el primer argumento de la función **fprintf** es *stdout*, **fprintf** se comporta exactamente igual que **printf**.

⁴Vea nuevamente la línea 43.

leer los datos almacenados en él y mostrarlos en la salida estándar. En éste sentido, el Ejemplo 9.2 es la parte complementaria del Ejemplo 9.1.

```

1  /* Uso de un archivo secuencial para la lectura de datos.
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #define N 30
6  #define TAMNOM 50
7  #define TAMTEL 20
8
9  typedef struct{
10     char nombre[TAMNOM];
11     char telefono[TAMTEL];
12 } CONTACTO;
13
14 void imprimeDirectorio(CONTACTO *, int);
15 int leeDirectorio(CONTACTO *);
16
17 int main(){
18     CONTACTO contactos[N];
19     int n;
20
21     n = leeDirectorio(contactos);
22     printf("Se leyeron %d contactos del directorio.\n", n);
23     imprimeDirectorio(contactos, n);
24
25     return 0;
26 }
27
28 int leeDirectorio(CONTACTO *contactos){
29     FILE *archivoPtr;
30     int i = 0;
31
32     if((archivoPtr = fopen("contactos.txt", "r")) == NULL)
33         printf("El archivo no pudo ser abierto.\n");
34     else{
35         for( ; fgets(contactos[i].nombre, TAMNOM, archivoPtr) != NULL; i++){
36             fgets(contactos[i].telefono, TAMTEL, archivoPtr);
37             fclose(archivoPtr);
38         }
39         return i;
40     }
41 }
42
43 void imprimeDirectorio(CONTACTO *contactos, int n){
44     int i;
45
46     printf("Lista de amigos:\n");
47     for(i = 0; i < n; i++){
48         printf("%d. %s", i+1, contactos[i].nombre);
49         printf("Tel. %s\n", contactos[i].telefono);
50     }
51 }

```

Ejemplo 9.2: Uso de un archivo secuencial para la lectura de datos

Al igual que antes, la explicación estará limitada a la función *leeDirectorio* (líneas 28 - 40), la cual gestiona los aspectos referentes al manejo del archivo para su lectura.

La función **fopen** abre el archivo “*contactos.txt*” (línea 32) en modo de lectura, debido a que no se requiere modificar los datos del mismo, sino sólo procesarlos para su almacenamiento en un arreglo.

Por otro lado, la función *leeDirectorio* muestra el uso de la función **fgets** en las líneas 35 y 36. La función **fgets** trabaja de manera análoga a la función **gets**, pero a diferencia de ésta última, **fgets** recibe tres parámetros:

1. La cadena donde se almacenarán los datos que se procesen (representada en el Ejemplo 9.2 por el argumento *contactos[i].nombre* y *contactos[i].telefono* respectivamente)
2. El número máximo de caracteres a almacenar en la cadena⁵ (representada en el Ejemplo 9.2 por el argumento *TAM_NOM* y *TAM_TEL* respectivamente)
3. El flujo del cual se procesarán los datos (representado en el Ejemplo 9.2 por el argumento *archivoPtr*)

La función **fgets** lee caracteres de un flujo y los almacena en la cadena correspondiente, hasta que se hayan leído a lo más el número de caracteres indicado menos uno (para almacenar el fin de cadena ‘\0’), se encuentre el símbolo de fin de archivo⁶, o se encuentre el símbolo de avance de línea y retorno de carro (‘\n’).

Tome en cuenta que aunque el símbolo ‘\n’ hace que la función **fgets** termine la lectura, éste es considerado un carácter válido, y en consecuencia, es también almacenado en la cadena.

Ahora bien, si la función **fgets** encuentra el símbolo *EOF* antes que cualquier otro carácter, la función regresa un apuntador *NULL*, situación que es verificada en la línea 35 del Ejemplo 9.2. Se recomienda revisar un manual de referencia para tener una mejor y más amplia comprensión de la función **fgets**.

⁵Esto resulta sumamente conveniente, debido a que la función **fgets** protege de desbordamientos: aunque la cantidad de datos que se procesen sea mayor que el tamaño de la cadena, dichos datos son descartados y sólo se almacena como máximo, el número de datos especificado en este argumento menos uno.

⁶El símbolo de fin de archivo está representado por la constante simbólica *EOF*.

```
1.      Nombre: Ricardo Ruiz Rodriguez
        Telefono: 222 22 12 1974

2.      Nombre: Ricardo Ruiz Sanchez
        Telefono: 222 20 04 2007

3.      Nombre: Bruno Ruiz Sanchez
        Telefono: 222 18 09 2008

Lista de amigos:
1. Ricardo Ruiz Rodriguez
Tel. 222 22 12 1974

2. Ricardo Ruiz Sanchez
Tel. 222 20 04 2007

3. Bruno Ruiz Sanchez
Tel. 222 18 09 2008
```

Figura 9.2: Salida del Ejemplo 9.2

Por último, note que la función *leeDirectorio* regresa el número (*i*) de contactos leídos del archivo (línea 39). Una posible salida para el Ejemplo 9.2 se muestra en la Figura 9.2.

9.2.2. Aplicaciones adicionales

Esta sección presenta dos sencillas pero prácticas aplicaciones de los archivos de texto, pero las posibilidades están únicamente limitadas a la imaginación del programador.

Implementación de *cat*

El comando *cat* (por concatenar) es un programa de Unix y GNU/Linux usado para concatenar y mostrar archivos en la salida estándar.

El programa del Ejemplo 9.3 realiza una implementación básica del comando *cat* respecto a la presentación de archivos en la salida estándar, no en cuanto a la concatenación.

La mayor parte de los elementos y sentencias utilizadas en el Ejemplo 9.3

ya han sido estudiados y presentados con anterioridad, por lo que debería comprenderse sin ningún problema. En éste sentido, sólo se enfatizarán los siguientes aspectos:

```

1  /* Implementacion basica del comando cat.
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #define TAM 150
6
7  int main(int argc, char *argv[]) {
8      FILE *archivoPtr;
9      char cadena[TAM];
10
11     if(argc != 2){
12         printf("Uso: %s archivo\n", argv[0]);
13     }else if((archivoPtr = fopen(argv[1], "r")) != NULL){
14         while(fgets(cadena, TAM, archivoPtr) != NULL)
15             fputs(cadena, stdout);
16         fclose(archivoPtr);
17     }else
18         printf("Error al intentar abrir el archivo \"%s\"\n", argv[1]);
19
20     return 0;
21 }

```

Ejemplo 9.3: Implementación básica del comando *cat*

1. En caso de que no se proporcionen o se exceda el número de argumentos requeridos para su ejecución (líneas 11 y 12), se presenta una sentencia de *uso* del comando al estilo de los comandos de Unix y GNU/Linux.
2. Se utiliza directamente la cadena almacenada en *argv[1]* para acceder al archivo indicado en la línea de comandos (línea 13).
3. Se hace uso de la función **fputs** (línea 15), la cual imprime lo que se le envía como primer argumento, en el flujo especificado como su segundo argumento, es decir: imprime en la salida estándar (*stdout*) la cadena *cadena*, misma que fue leída por la función **fgets** en la línea 14.

Pruebe el funcionamiento del programa del Ejemplo 9.3, y asegúrese de proporcionarle archivos de texto como argumento; puede probar con el código fuente del mismo programa por ejemplo.

Cambia caracteres

Suponga que se tiene la necesidad de cambiar cada ocurrencia de un carácter dentro de un archivo, por otro carácter distinto. Un escenario de

solución para esto es realizar manualmente el cambio, otro posible escenario, es escribir un programa en C que lleve a cabo dicha tarea por nosotros; el Ejemplo 9.4 muestra un programa que satisface dicha necesidad.

```

1  /* Cambia un caracter por otro dentro de un archivo.
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  void cambiaOcurrienciaCadena(char, char, char *);
8  void cambiaOcurrienciaArchivo(char, char, char *, char *);
9
10 int main(int argc, char *argv[]){
11     if(argc != 5)
12         printf("Uso: %s char_viejo char_nuevo archivo_fuente "
13             "archivo_destino\n", argv[0]);
14     else{
15         cambiaOcurrienciaArchivo(argv[1][0], argv[2][0], argv[3], argv[4]);
16         printf(" Listo!\n");
17     }
18
19     return 0;
20 }
21
22 void cambiaOcurrienciaCadena(char v, char n, char *cad){
23     while(*cad){
24         if(*cad == v)
25             *cad = n;
26         cad++;
27     }
28 }
29
30 void cambiaOcurrienciaArchivo(char v, char n, char *a1, char *a2){
31     #define TAM 200
32     FILE *archivoPtr1, *archivoPtr2;
33     char cadena[TAM];
34
35     if((archivoPtr1 = fopen(a1, "r")) == NULL){
36         printf("Error al abrir el archivo: %s\n", a1);
37         exit(1);
38     }else if((archivoPtr2 = fopen(a2, "w")) == NULL){
39         printf("Error al abrir el archivo: %s\n", a2);
40         exit(1);
41     }
42
43     while(fgets(cadena, TAM, archivoPtr1) != NULL){
44         cambiaOcurrienciaCadena(v, n, cadena);
45         fputs(cadena, archivoPtr2);
46     }
47     fclose(archivoPtr1);
48     fclose(archivoPtr2);
49 }

```

Ejemplo 9.4: Cambia un carácter por otro dentro de un archivo

Al igual que antes, las sentencias del Ejemplo 9.4 deben resultar familiares al lector, por lo que sólo se resaltarán los siguientes puntos:

1. La función **main** procesa los argumentos introducidos en la línea de comandos, y los utiliza para enviárselo a la función que realizará el cambio sobre el archivo: *cambiaOcurrenciaArchivo*.
2. La función *cambiaOcurrenciaArchivo* recibe cuatro parámetros:
 - a) El carácter a reemplazar *v*.
 - b) El carácter *n* que substituirá a *v*.
 - c) La cadena *a1* que contiene la ruta y el nombre del archivo sobre el que se buscará a *v*.
 - d) La cadena *a2* que contiene la ruta y el nombre del archivo sobre el que se realizará el cambio de *v* por *n*.
3. La constate simbólica *TAM*: en éste tipo de uso, su ámbito se define de la línea 31 a la 49.
4. La función *cambiaOcurrenciaCadena*, que se encarga de substituir el carácter *v* por el carácter *n* en la cadena *cad* utilizando notación de apuntadores.
5. El uso de la función *cambiaOcurrenciaCadena* de la línea 44 para cambiar la cadena *cadena* leída del archivo *archivoPtr1* (línea 43), en la cadena *cadena* que se almacenará en el archivo *archivoPtr2* (línea 45) por medio de la función **fputs**⁷.

Pruebe con distintos archivos de texto el programa del Ejemplo 9.4. La salida en pantalla es sólo informativa respecto a la terminación del programa, lo interesante ocurre en los archivos procesados, los cuales son proporcionados en la invocación del programa.

⁷La función **fputs** trabaja de manera análoga a la función **puts**, pero recibe como segundo parámetro, un apuntador al flujo sobre el que se desea escribir la cadena referida como su primer parámetro.

9.2.3. Consideraciones adicionales

Todos los ejemplos de la sección 9.2 han recorrido de principio a fin el archivo que procesan pero, ¿qué sucede si se necesita volver a procesar nuevamente los datos del archivo?.

Dentro de la información indispensable que se encuentra representada en la estructura *FILE* para el control del archivo asociado, se encuentra, como se mencionó con anterioridad, un **indicador de posición**, el cual hace referencia a la posición del último dato procesado en el archivo.

Cuando un archivo ha sido procesado de principio a fin, el indicador de posición se encontrará ubicado al final del archivo, de tal forma que si por alguna razón se procesara de nuevo el archivo para lectura por ejemplo, se estaría en la situación de una lectura vacía, dando la impresión de que el archivo no contiene datos, aún cuando sí los tenga, debido precisamente a que el indicador de posición del archivo se encuentra al final del mismo.

Una posible solución a la problemática anterior sería cerrar el archivo, abrirlo nuevamente y procesarlo; sin embargo, el abrir y cerrar un archivo sólo para regresar el indicador de posición a su posición inicial es, además de ineficiente, un pésimo enfoque de solución: ¿qué sucede si el archivo tendrá un uso intensivo?, ¿es práctico estarlo abriendo y cerrando frecuentemente?.

La función **rewind** coloca el indicador de posición del archivo al inicio del archivo. Así, un llamado como *rewind(archivoPtr)*; coloca el indicador de posición del archivo referido por *archivoPtr* al inicio de éste, de tal forma que se puedan procesar nuevamente el conjunto de datos almacenados en él, tal y como si el archivo se hubiera abierto por primera vez.

El Ejercicio 9 de la Sección 9.4, propone una modificación a uno de los programas de ejemplo para que tenga la oportunidad de trabajar y experimentar con la función **rewind**. Si no puede esperar más, vaya sin más preámbulo a dicho ejercicio para que pueda leer su descripción y resolverlo, le auguro éxito.

Finalmente, debido a que la función **fprintf** es equivalente a la función **printf**, quizá se haya preguntado si es posible escribir en un archivo de texto otros tipos de datos además de cadenas, esto es, ¿es posible escribir en un archivo de texto otros tipos de datos como los siguientes:

```
fprintf(archivoPtr, "%d\n%s\n%s\n%.2f\n",
        num, nombre, direccion, salario);
```

donde *num* es de tipo **int**, *nombre* y *direccion* son cadenas, y *salario* es de

tipo **float**?

La respuesta es sí. Sin embargo, debe tomar en cuenta que cualquier tipo de dato, al ser almacenado en un archivo de texto, pierde su representación como tipo de dato y pasa a ser sólo una sucesión de caracteres dentro del archivo, de tal forma que al leer *num* o *salario* por ejemplo, se podrían leer como cadenas, y esto sería igual de válido que leerlos como un **int** o como un **float** respectivamente. Por lo tanto, las sentencias:

```
fscanf(archivoPtr, "%d %s %s %f",
        &num, nombre, direccion, &salario);
fscanf(archivoPtr, "%f %s %s %f",
        &numero, nombre, direccion, &salario);
fscanf(archivoPtr, "%s %s %s %s",
        cad1, nombre, direccion, cad2);
```

serían igualmente válidas, suponiendo que *numero* fuera de tipo **float** y que *cad1* y *cad2* fueran cadenas; aunque la más congruente es la primera, en función de los datos que, según la idea planteada, se escribieron en el archivo referido por *archivoPtr*.

Cuando trabaje con archivos de texto, no olvide tener en mente todas las consideraciones mencionadas en esta sección.

9.3. Archivos binarios

Aunque C no impone una estructura a los archivos, es posible definir una estructura específica para los archivos. Sin embargo, la creación, la administración, y el acceso a dicha estructura en el archivo, son responsabilidad del programador.

Un **archivo binario** es un archivo con una estructura específica, la cual no puede ser visualizada ni interpretada de manera directa, sino únicamente como un conjunto de *bytes* relacionados entre sí, los cuales representan los tipos de datos que fueron almacenados en la estructura del archivo.

9.3.1. Archivos de acceso aleatorio

La creación de una estructura determinada sobre un archivo, tiene, como casi todo en la vida, ventajas y desventajas.

La principal ventaja es que es posible acceder a un elemento específico dentro del archivo, sin la necesidad de procesar todos los elementos anteriores a él, como en el caso de los archivos de texto de acceso secuencial.

Por otro lado, la principal desventaja es que, antes de poder acceder a los datos del archivo, se debe crear la estructura correspondiente, y por lo tanto, es preciso definir desde la creación del archivo, el número de elementos que almacenará.

Como analogía, puede decirse que los archivos de acceso aleatorio son a las unidades de almacenamiento (discos), lo que los arreglos son a la memoria principal, de hecho, note que tanto la ventaja como la desventaja mencionadas con anterioridad, se tienen también presentes en los arreglos.

En resumen, un **archivo de acceso aleatorio** es un archivo binario con una estructura específica determinada por el programador, al que se pueden acceder sus elementos de manera aleatoria, de manera análoga a como se acceden los elementos en un arreglo.

Creación de la estructura del archivo

El primer paso para la manipulación de archivos de acceso aleatorio, es la creación de la estructura del archivo.

La creación de la estructura consiste básicamente en definir cuáles y de qué tipo de dato serán los elementos almacenados en el archivo. Lo anterior se realiza, normalmente, encapsulando dentro de una estructura los elementos a almacenar en el archivo, sin embargo, es posible almacenar elementos de un solo tipo de datos en el archivo, sin necesidad de representarlos dentro de una estructura.

El Ejemplo 9.5 muestra la creación de la estructura de un archivo que contendrá un directorio de contactos, ésta idea ha sido ya utilizada en la Sección 9.2, y aquí se retoma para mostrar ahora la representación del directorio de contactos en un archivo de acceso aleatorio.

Note que la estructura (**struct**) de las líneas 9-13 ha sido ligeramente modificada respecto de las anteriores, ya que se le ha agregado el elemento miembro *num* (línea 10), el cual servirá como índice para localizar a una estructura específica dentro del archivo.

La línea 16 define la variable *contacto* de tipo *CONTACTO*, y la inicializa con el valor cero, y cadenas vacías para *num*, *nombre* y *telefono* respectivamente.

Como elemento clave del Ejemplo 9.5, observe que el modo de apertura seleccionado para el archivo *contactos.dat* en la línea 20 (**wb**), el cual especifica que se debe crear un archivo binario.

```

1  /* Creacion de un archivo de acceso aleatorio con datos en "blanco".
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #define N 100
6  #define TAMNOM 50
7  #define TAMTEL 20
8
9  typedef struct{
10     int num;
11     char nombre[TAMNOM];
12     char telefono[TAMTEL];
13 } CONTACTO;
14
15 int main(){
16     CONTACTO contacto = {0, "", ""};
17     FILE *archivoPtr;
18     int i;
19
20     if((archivoPtr = fopen("contactos.dat", "wb")) == NULL)
21         printf("El archivo no pudo ser abierto.\n");
22     else{
23         for(i = 0; i < N; i++)
24             fwrite(&contacto, sizeof(CONTACTO), 1, archivoPtr);
25         fclose(archivoPtr);
26     }
27     return 0;
28 }

```

Ejemplo 9.5: Creación de la estructura de un archivo binario de acceso aleatorio

Ahora bien, el ciclo **for** (línea 23) escribe N veces en el archivo referido por *archivoPtr*, $\text{sizeof}(\text{CONTACTO})$ bytes almacenados en la estructura *contacto*, a través de la función **fwrite** (línea 24). El número 1 (tercer argumento) de la función **fwrite**, le indica a la función cuantos elementos del tamaño⁸ especificado como segundo argumento obtendrá de la dirección especificada como primer argumento, para escribir en el flujo proporcionado como cuarto argumento. Éste es el uso más común de la función **fwrite**.

Por otro lado, si el tercer argumento n proporcionado a la función **fwrite** es mayor que uno, entonces el primer argumento debe corresponder al nombre de un arreglo del cual se obtendrán los n elementos. El número t de bytes de cada uno de los elementos del arreglo se proporciona como segun-

⁸El tamaño se especifica en número de *bytes*.

do argumento, y el flujo donde se desea que se escriban los $t * n$ bytes se proporciona como cuarto argumento.

Resumiendo: la ejecución del programa del Ejemplo 9.5, crea el archivo *contactos.dat*, cuya estructura está conformada por N elementos (línea 5) de tipo *CONTACTO*, es decir, genera una especie de arreglo de N elementos de tipo *CONTACTO* en el archivo *contactos.dat*.

Acceso aleatorio a la estructura del archivo

El programa del Ejemplo 9.6 muestra los pormenores respecto al manejo de archivos binarios de acceso aleatorio. Tome en cuenta que este ejemplo se basa en la estructura de archivo creada en el Ejemplo 9.5.

Las funciones *menu*, *leeContacto* e *imprimeContacto* se dejan como material de análisis y comprensión para el lector. Todos los detalles de éstas funciones deberían ser claramente comprendidos; asegúrese de que así sea antes de continuar.

```

1  /* Lectura y actualizacion de un archivo de acceso aleatorio.
2     @autor: Ricardo Ruiz Rodriguez
3  */
4  #include <stdio.h>
5  #define TAMNOM 50
6  #define TAMTEL 20
7  #define SALIR 4
8
9  typedef struct{
10     int num;
11     char nombre[TAMNOM];
12     char telefono[TAMTEL];
13 } CONTACTO;
14
15 int leeDirectorio(FILE *);
16 void agregaContacto(FILE *);
17 void imprimeContacto(const CONTACTO *);
18 void leeContacto(CONTACTO *);
19 int menu(void);
20
21 int main(){
22     FILE *archivoPtr;
23     int opcion;
24
25     if((archivoPtr = fopen("contactos.dat", "rb+")) == NULL)
26         printf("El archivo no pudo ser abierto.\n");
27     else{
28         leeDirectorio(archivoPtr);
29         while((opcion = menu()) != SALIR){
30             switch(opcion){
31                 case 1: agregaContacto(archivoPtr);
32                     break;
33                 case 2: /* Bajas */

```



```

34         break;
35         case 3: /* Modificaciones */
36             break;
37         default: printf("\n\tOpcion no valida.\n\n");
38     }
39 }
40 fclose(archivoPtr);
41 }
42 return 0;
43 }
44
45 void agregaContacto(FILE *archivoPtr){
46     int num;
47     CONTACTO contacto;
48
49     do{
50         printf("Contacto numero? (1 - 100): ");
51         scanf("%d", &num);
52     }while(num < 1 || num > 100);
53     getchar(); /* Se "come" el ENTER del numero de contacto */
54
55     fseek(archivoPtr, (num - 1) * sizeof(CONTACTO), SEEK_SET);
56     fread(&contacto, sizeof(CONTACTO), 1, archivoPtr);
57
58     if(contacto.num != 0)
59         printf("\n\tEl contacto %d ya tiene informacion.\n\n", num);
60     else{
61         contacto.num = num;
62         leeContacto(&contacto);
63         fseek(archivoPtr, (num - 1) * sizeof(CONTACTO), SEEK_SET);
64         fwrite(&contacto, sizeof(CONTACTO), 1, archivoPtr);
65         printf("\n\tContacto agregado...\n\n");
66     }
67 }
68
69 int leeDirectorio(FILE *archivoPtr){
70     int n = 0;
71     CONTACTO contacto;
72
73     printf("Leyendo archivo...\n");
74     while(fread(&contacto, sizeof(CONTACTO), 1, archivoPtr) > 0)
75         if(contacto.num != 0){
76             n++;
77             imprimeContacto(&contacto);
78         }
79     printf("Existen %d contactos registrados.\n", n);
80
81     return n;
82 }
83
84 void imprimeContacto(const CONTACTO *contacto){
85     printf("%d. ", contacto->num);
86     printf("%s", contacto->nombre);
87     printf("Tel. %s\n", contacto->telefono);
88 }
89
90 void leeContacto(CONTACTO *contacto){

```

```

91     printf("\tNombre?: ");
92     fgets(contacto -> nombre, TAMNOM, stdin);
93     printf("\tTelefono?: ");
94     fgets(contacto -> telefono, TAMTEL, stdin);
95     putchar('\n');
96 }
97
98 int menu(void){
99     int op;
100
101     printf("\n\t**** MENU ****\n");
102     printf("[1] Agregar contacto\n");
103     printf("[2] Eliminar contacto\n");
104     printf("[3] Modificar contacto\n");
105     printf("[4] Salir\n");
106     printf("Opcion?: ");
107     scanf("%d", &op);
108
109     return op;
110 }

```

Ejemplo 9.6: Acceso para la modificación de datos de un archivo binario de acceso aleatorio

Respecto a la función **main** sólo se harán dos observaciones, todo lo demás debe resultarle familiar:

1. La línea 25 muestra el modo de apertura **rb+**, lo cual le indica a la función **fopen** que abra el archivo binario *contactos.dat* para actualización (consulte la Tabla 9.1).
2. A diferencia de todos los ejemplos anteriores, el archivo *contactos.dat* es abierto al iniciar el programa, y permanece así hasta su terminación (línea 40). Observe cómo la variable *archivoPtr* es enviada como argumento a las funciones que trabajan con el archivo (líneas 28 y 31).

Teniendo en cuenta lo anterior, la atención del Ejemplo 9.6 se centrará entonces en dos funciones:

1. *leeDirectorio*: es la encargada de leer los datos del archivo por medio de la función **fread** (línea 74)⁹, la cual trabaja de manera análoga pero en el sentido contrario a la función **fwrite** explicada en el Ejemplo 9.5.

⁹Mientras **fread** pueda leer datos del archivo, regresa el número total de elementos exitosamente procesados, cuando se llegue al fin de archivo y la función no pueda leer más datos, la función regresará cero.

Si el elemento miembro *num* de algún contacto es distinto de cero¹⁰ (línea 75), se imprimen sus datos y se contabiliza.

2. *agregaContacto*: Esta función se desglosará en varias partes por orden de secuencia:

- a) Solicita y valida un número de contacto (líneas 49 - 52)¹¹.
- b) La función **getchar** (línea 53) absorbe el ‘\n’ introducido en la entrada estándar después del número leído en la línea 51, el número proporcionado se almacena en *num*, pero si el ‘\n’ no es procesado, será el elemento del que disponga la siguiente lectura de datos, y si ésta es la lectura de una cadena, dará la impresión de leer una cadena vacía.
- c) La función **fseek** (líneas 55 y 63) establece el indicador de posición del archivo *archivoPtr* en una determinada posición, la cual es proporcionada como segundo argumento en la forma de un desplazamiento en *bytes*. El tercer argumento le indica a la función la posición utilizada como referencia para el desplazamiento:
 - *SEEK_SET*: del inicio del archivo.
 - *SEEK_CUR*: de la posición actual del indicador de posición del archivo.

Note que la expresión de desplazamiento:

$$(\text{num} - 1) * \text{sizeof}(\text{CONTACTO})$$

realiza el cálculo del elemento específico dentro del archivo, lo cual, respecto a la analogía planteada con anterioridad, es equivalente al índice de un arreglo.

- d) La combinación de las funciones **fseek** y **fread** (líneas 55 y 56) y de **fseek** y **fwrite** (líneas 63 y 64) realizan la parte medular respecto a la inserción de un nuevo contacto en el archivo.

¹⁰El valor cero indica que es un contacto en blanco, dado que así se inicializó la estructura del archivo (Ejemplo 9.5), cualquier otro caso hace referencia a un número de contacto con datos ya almacenados.

¹¹Recuerde que en el Ejemplo 9.5 se generó una estructura en el archivo para almacenar cien contactos.

- e) Note la necesidad de volver a calcular la posición del elemento a insertar en el archivo (línea 63), debido a que la lectura del contacto (línea 56) deja el indicador de posición del archivo, en el elemento siguiente al de interés.

La salida del Ejemplo 9.6 puede ser bastante extensa, la invitación es hacia compilar el programa y probarlo, así como a entender la relación del funcionamiento de cada una de la sentencias que componen el programa, hasta obtener una total comprensión de dicho ejemplo.

9.3.2. Archivos con contenido especial

El último ejemplo del capítulo y del libro se muestra en el Ejemplo 9.7, el cual abre un archivo binario independientemente de su contenido, formato o estructura, lee su contenido, y presenta en la salida estándar los *bytes* que lo conforman.

La mayoría de los elementos del programa han sido ya analizados o mencionados, por lo que su comprensión debería ser sencilla, sin embargo, se enfatizarán algunos aspectos relevantes:

1. El programa procesa datos de la línea de comandos, es decir, recibe el archivo a procesar como argumento en la invocación del programa (líneas 8 y 11 - 16).
2. Note que la función **rewind** (línea 24) establece el indicador de posición nuevamente al inicio del archivo referido por *archivoPtr*, debido a que en la línea 21 se lee el archivo *byte* por *byte* para poder determinar su tamaño a través de la función **ftell** (línea 23)¹².
3. Observe cómo la función **fread** lee el archivo completo en la línea 33, y que el número de *bytes* leídos es almacenado en *tamano2*, para ser posteriormente comparado (línea 34) con el número de *bytes* que se le especificó a la función **fread** que leyera.
4. Finalmente, note en la línea 42 el uso del especificador de formato de salida “%x”, el cual le indica a la función **printf** que imprima el

¹²La función **ftell** regresa el indicador de posición actual del flujo asociado al archivo *archivoPtr*. Si el archivo al que se hace referencia es binario, dicho indicador corresponde al número de *bytes* respecto del inicio del archivo.

dato almacenado en `buffer[i]`, en formato hexadecimal (minúsculas); mientras que el especificador de formato de salida “%u” (línea 43) se utiliza para imprimir un entero sin signo.

```

1  /* Ejemplo de lectura de un archivo binario. Con este programa puede
2     verse la estructura de un archivo mp3 por ejemplo.
3     @autor: Ricardo Ruiz Rodriguez
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main(int argc, char *argv[]) {
9      FILE *archivoPtr;
10
11      if(argc != 2){
12          printf("Uso: %s archivo\n", argv[0]);
13          exit(0);
14      }
15      if((archivoPtr = fopen(argv[1], "rb")) == NULL)
16          printf("Error al abrir el archivo \"%s\"\n", argv[1]);
17      else{
18          unsigned char byte, *buffer;
19          size_t tamaniol, tamanio2;
20          printf("Leyendo el archivo...\n");
21          while(fread(&byte, 1, 1, archivoPtr))
22              ;
23          tamaniol = ftell(archivoPtr); /* Se obtiene el tamaño del archivo */
24          rewind(archivoPtr); /* Regresa el indicador de posición al inicio */
25
26          buffer = (unsigned char *) malloc(tamaniol * sizeof(unsigned char));
27          if(buffer == NULL){
28              printf("No hay memoria disponible para procesar el archivo\n");
29              fclose(archivoPtr);
30              exit(0);
31          }
32
33          tamanio2 = fread(buffer, 1, tamaniol, archivoPtr);
34          if(tamanio2 != tamaniol){
35              fputs("Error al leer el archivo\n", stderr);
36              free(buffer); fclose(archivoPtr);
37              exit(3);
38          }else{
39              unsigned long i;
40              printf("Bytes del archivo:\n");
41              for(i = 0; i < tamaniol; i++)
42                  printf("%x ", buffer[i]);
43              printf("\n%u bytes leídos\n", tamanio2);
44          }
45          fclose(archivoPtr);
46          free(buffer);
47      }
48      return 0;
49  }

```

Ejemplo 9.7: Lectura de un archivo binario

Pruebe el programa del Ejemplo 9.7 con distintos archivos binarios, puede probarlo con archivos ejecutables, de música, de video, de imágenes, e incluso de texto.

Por último, tome en cuenta que el Ejemplo 9.7 fue diseñado para contener en memoria todos los *bytes* del archivo procesado, por lo que si el archivo es muy grande, la solicitud de una gran cantidad de memoria podría ser rechazada. En la sección de ejercicios tendrá la oportunidad de corregir ésta deficiencia.

9.4. Ejercicios

1. La Tabla 9.1 muestra cuatro modos de apertura para archivos, independientemente de si son o no binarios. Investigue para qué sirven los modos de apertura **w+** y **a+**.
2. Modifique el Ejemplo 9.1 en las líneas 47 y 48. Substituya *archivoPtr* por *stdout*, y vea lo que sucede. ¿Se comprueba lo descrito en el texto?
3. Para el programa del Ejemplo 9.1 ¿qué sucede si en lugar de abrir el archivo en modo agregar se abre en modo leer, sin cambiar ninguna otra sentencia?
4. Para el programa del Ejemplo 9.1, ¿qué sucede si en lugar de abrir el archivo en modo agregar se abre en modo escribir, sin cambiar ninguna otra sentencia?
5. Modifique la función *llenaDirectorio* del Ejemplo 9.2, para que en lugar de la función **gets** use la función **fgets**.
6. Modifique el programa del Ejemplo 9.3 para que procese, y en consecuencia presente en la salida estándar, todos los archivos que se le proporcionen en la invocación.
7. Escriba un programa que compare dos archivos y determine si dichos archivos son o no iguales. Los archivos procesados serán proporcionados desde la invocación del programa. Así, si su programa se llama *compara*, la ejecución:

```
$/compara archivo1 archivo2
```

implica que se deben comparar *archivo1* con *archivo2*.

Si los archivos son iguales, se notifica al usuario de dicha situación, en caso contrario, se indicará la primera línea de cada archivo que no coincidió.

8. Para el programa del Ejemplo 9.4, pruebe lo siguiente:

- a) Seleccione un archivo de texto que contenga la letra de una canción por ejemplo. Busque de preferencia la letra de una canción en inglés, para omitir los detalles referentes a la letra ñ, los acentos, y la diéresis de nuestro lenguaje.
 - b) Cambie la vocal *a* del archivo fuente (*archivo1*), por la vocal *u* en el archivo destino (*archivo2*).
 - c) Use el archivo generado en el paso anterior (*archivo2*), como archivo fuente para una nueva ejecución, en la que cambiará ahora la vocal *u* por la vocal *a*, generándose un nuevo archivo destino: *archivo3*.
 - d) Si todo está bien, *archivo1* debe ser idéntico a *archivo3*. Compruebe que así sea. Puede apoyarse de programas que comparan archivos, como el comando **diff** de Unix y GNU/Linux, y el programa *compara* del Ejercicio 7.
9. Modifique el programa del Ejemplo 9.3 para que presente en la salida estándar, dos veces, una seguida de otra, el archivo procesado. La idea de este ejercicio es que pueda practicar el uso de la función **rewind** descrita brevemente en la Sección 9.2.3.
 10. Escriba un programa que procese un archivo de texto proporcionado en la invocación del programa, y presente ordenadas en la salida estándar, cada una de las líneas del archivo. Así, la ejecución de:

```
$./ordena archivo
```

presenta en la salida estándar cada una de las líneas de texto del archivo *archivo*, ordenadas alfabéticamente.

Consulte el funcionamiento del programa **sort** de Unix y GNU/Linux para tener una mejor idea acerca del funcionamiento de su programa.

11. Pruebe lo que sucede si elimina la función **getchar** (línea 53) del Ejemplo 9.6.
12. El programa del Ejemplo 9.6 está incompleto. Modifíquelo para completar las opciones de menú:

- a) Eliminar contacto.
- b) Modificar contacto.

Tome en consideración que la eliminación de un contacto requiere de la confirmación de la eliminación de dicho contacto antes de proceder, y que la modificación del contacto puede ser en varios sentidos:

- a) Conservar los datos del contacto pero moverlo de lugar, es decir, que sea ahora otro número de contacto, lo cual implica verificar que el nuevo número de contacto seleccionado no esté ya ocupado.
- b) Modificar sólo el nombre del contacto.
- c) Modificar únicamente el teléfono del contacto.
- d) Modificar tanto el nombre como el teléfono del contacto.

Para el caso de la modificación, no olvide también confirmar los cambios.

- 13. En base al Ejercicio 9 del Capítulo 8, complemente la funcionalidad planteada en el ejercicio, para que su programa sea capaz de leer figuras geométricas desde un archivo de texto especificado por el usuario.
- 14. Escriba un programa que en base a lo siguiente:

```
$ ./morse cadena archivo1 archivo2
```

permita leer el archivo de texto *archivo1*, de tal forma que si *cadena* es “a”, su programa:

- a) Procese únicamente símbolos alfanuméricos y espacios de *archivo1*, y los convierta a su correspondiente representación en código Morse en *archivo2*. Los espacios quedan sin cambio alguno. Cualquier otro símbolo es ignorado y descartado.
- b) Cada símbolo representado en código Morse irá separado por comas en *archivo2*. Los espacios quedan sin cambio alguno.

Por otro lado, si *cadena* es “de”, se deberá hacer el proceso inverso: convertir de Morse a texto¹³.

Sugerencia: para el caso de Morse a texto, investigue la función *strtok* de la biblioteca *string.h*.

15. En el Capítulo 8 se desarrolló una biblioteca de funciones que inicio con el Ejemplo 8.5. Posteriormente, en la sección de ejercicios, se fueron desarrollando nuevas funciones que fueron incorporadas a dicha biblioteca. Tomando en cuenta lo anterior, realice lo siguiente:
 - a) Genere una nueva biblioteca de funciones que incorpore gestión de archivos, a todas las operaciones con matrices realizadas:
 - 1) suma.
 - 2) resta.
 - 3) producto.
 - 4) transpuesta.
 - 5) esSimetrica.
 - 6) inversa.
 - b) Presente un menú de opciones con las operaciones que es posible realizar con matrices. Agregue al menú la opción de *Matriz Aleatoria*, la cual deberá realizar lo siguiente:
 - 1) Solicitar las dimensiones del matriz: m (renglones) y n (columnas).
 - 2) Solicitar un rango para el numerador (num) y un rango para el denominador (den).
 - 3) Generar dinámicamente una matriz de $m \times n$ de números racionales aleatorios, los cuales tendrán la forma:

$$\frac{p}{q}, \quad p \in \{0 \dots num - 1\}, \quad q \in \{0 \dots den - 1\}$$
 - c) Para cada operación, se deberá permitir introducir la o las matrices (según sea el caso), desde la entrada estándar o desde un archivo de texto especificado por el usuario.
 - d) Para cada operación, excepto *esSimetrica*, se deberá permitir, además de visualizar el resultado en la salida estándar, guardarlo en un archivo de texto especificado por el usuario.

¹³En efecto, se habrá perdido la redacción original pero la esencia del texto deberá ser la misma.

16. Modifique el Ejemplo 9.7 para que pueda procesar archivos de cualquier tamaño.
Sugerencia: defina un *buffer* de un tamaño fijo, digamos 4 KB, y lea bloques de 4 KB en cada acceso al archivo. Tome en cuenta que el último bloque leído podría tener menos de 4 KB¹⁴, por lo que deberá verificar el número de *bytes* leídos por la función **fread** en cada lectura.
17. Tome como referencia el Ejemplo 9.7, para hacer una implementación del comando **cp** (*copy*) de GNU/Linux.

¹⁴Si el archivo es muy pequeño dicha situación podría presentarse desde el primer bloque leído.

Apéndice A

Fundamentos de programación

Este apéndice introduce al lector en tres temas:

1. Algoritmos.
2. Pseudocódigo.
3. Diagramas de flujo.

y se presentarán también algunos conceptos relacionados con el paradigma de la programación estructurada, con el objetivo de proporcionar un marco de referencia propicio a dicho paradigma.

A.1. Panorama general

La resolución de un problema mediante una computadora, se refiere al *proceso* que consiste en partir de la descripción de un problema¹, desarrollar un *programa* de computadora que resuelva dicho problema. Éste proceso considera, *grosso modo*, las siguientes etapas:

1. Comprensión del problema.
2. Análisis del problema.
3. Diseño o desarrollo de un algoritmo.

¹Dicho problema está habitualmente expresado en lenguaje natural, y en los términos propios del dominio del problema.

4. Transformación del algoritmo en un programa (codificación).
5. Ejecución y validación del programa.

Aunque cada una de éstas etapas es de significativa importancia, la experiencia indica que las tres primeras etapas son en las el estudiante presenta mayor dificultad.

En este sentido, el orden en el que se presentan dichas etapas no es aleatorio, ya que cada una de ellas supone la adecuada terminación de la anterior, así por ejemplo, sería imposible tratar de analizar un problema que ni siquiera se ha comprendido en su contexto más general, y ni qué decir entonces de la elaboración de un algoritmo.

Probablemente la etapa más sencilla de todas sea la número cuatro: la codificación, ya que una vez que el problema se ha comprendido, analizado, y obtenido un algoritmo que lo resuelve, su transformación a un programa de computadora es una tarea de simple traducción, que si bien requiere del conocimiento de la sintaxis del lenguaje elegido para ello, la parte intelectual y de raciocinio, queda plasmada en el algoritmo.

La **meta de la programación estructurada** es la de hacer programas que sean fáciles de diseñar, depurar, y susceptibles de ser entendidos por personas diferentes a las que escribieron el programa [Ullman]. No se trata de escribir programas que solamente su creador entienda, se trata de algo más que eso.

Ahora bien, un programa puede ser visto como un *sistema* en el que sus componentes son programas más pequeños, y a su vez estos componentes pueden ser vistos como otros, compuestos por similares todavía más pequeños, y así sucesivamente hasta un nivel granular razonable, en donde los componentes finales están compuestos de unas cuantas sentencias de código. Los programas escritos de la forma descrita se denominan **programas estructurados** [Ullman].

En un programa estructurado, el flujo lógico de ejecución se gobierna por tres estructuras de control

1. Estructura secuencial.
2. Estructuras de selección.
3. Estructuras de repetición.

Dichas estructuras constituyen una de las partes medulares de la programación estructurada, por lo que se mencionan y describen con más detalle en secciones posteriores.

A.1.1. Teorema de Böhm y Jacopini

El teorema de Böhm y Jacopini establece, que un **programa propio** puede ser escrito utilizando únicamente tres tipos de estructuras de control: estructura secuencial, estructuras de selección y estructuras de repetición.

Para que un **programa sea estructurado**, el programa debe ser propio, y un programa se define como propio, si cumple las siguientes condiciones:

- Tiene un único punto de entrada, y un único punto de salida.
- Todas las sentencias del algoritmo son alcanzables, y existe al menos un camino que va desde el inicio hasta el fin del algoritmo.
- No tiene ciclos infinitos.

De lo anterior se deduce que, si los algoritmos se diseñan empleando las estructuras de control mencionadas, los algoritmos, y en consecuencia, los programas derivados de ellos, serán propios.

A.1.2. Estructuras de Control

Las estructuras secuencial, de selección y de repetición se denominan **estructuras de control** porque son las que **controlan** el modo o flujo de ejecución del algoritmo. Su importancia es tal que, una vez que se comprendan, se podrá tener el control respecto al flujo de los algoritmos.

Estructura secuencial

La estructura secuencial es la más simple de las tres, y se caracteriza porque una acción o sentencia se ejecuta detrás de otra, es decir, el flujo del algoritmo coincide con el orden físico en el que se han colocado las sentencias en el algoritmo.

Estructuras de selección

Este tipo de estructuras realizan una **selección** de las acciones o sentencias a ejecutar. La selección de qué sentencias se procesarán está en directa función o dependencia de una **condición**.

La condición puede ser tan elaborada como la naturaleza del problema lo requiera, pero se deberá asegurar de que dicha condición pueda ser evaluada de manera *booleana*, esto es, como **verdadera** o **falsa**: sin ambigüedad².

Existen tres tipos de estructuras de selección:

1. Estructura de selección simple.
2. Estructura de selección doble.
3. Estructura de selección múltiple.

Los flujos de ejecución que siguen, dependiendo de la evaluación de su condición, pueden visualizarse mejor al observar sus representaciones en diagrama de flujo o en pseudocódigo, y se describirán en las secciones A.3.1 y A.4.1 respectivamente.

Estructuras de repetición

En las estructuras de repetición, las acciones o sentencias del cuerpo del ciclo se repiten mientras se cumpla una determinada condición, misma que deberá seguir los mismos lineamientos descritos para las estructura de selección.

En este tipo de estructuras, es frecuente el uso de **contadores**³ o **centinelas**⁴ para controlar el número de repeticiones de un ciclo.

En las secciones A.3.1 y A.4.1, se muestran las notaciones algorítmicas, tanto para diagramas de flujo como para pseudocódigo respectivamente, utilizadas en las estructuras de repetición.

²La ambigüedad viola una de las cinco condiciones de un algoritmo (vea la sección A.2.3).

³Utilizados cuando se conoce de antemano el número de repeticiones.

⁴Utilizados cuando no se conoce con anticipación el número de repeticiones.

A.2. Algoritmos

El concepto de algoritmo⁵ forma parte esencial de los fundamentos de la computación. La matemática discreta y en particular la matemática constructivista, son tan antiguas como la propia matemática, y trata aquellos aspectos para los cuales existe una solución constructivista, esto es, no se conforma con demostrar la existencia de solución, sino que se pregunta cómo encontrar dicha solución [Abellanas].

A.2.1. Definición y conceptos

En términos generales se define un **algoritmo**, como el *método* para resolver un determinado problema, donde el ejecutor de las instrucciones que realiza la tarea correspondiente se denomina **procesador**.

Existen algoritmos que describen toda clase de procesos, por ejemplo:

- Las recetas de cocina.
- Las partituras musicales.
- Cambiar la llanta de un auto, etc.

El procesador realiza dicho proceso siguiendo o ejecutando el algoritmo correspondiente. La ejecución de un algoritmo requiere de la ejecución de cada uno de los pasos o instrucciones que lo conforman.

Por otro lado, un algoritmo debe estar expresado de tal forma que el procesador lo entienda para poder ejecutarlo. Se dice que el procesador es capaz de interpretar un algoritmo, si el procesador puede:

1. Entender lo que significa cada paso.
2. Llevar a cabo (procesar) la sentencia correspondiente.

⁵El término algoritmo es muy anterior a la era de la computación: proviene de Mohammed al-Khowârizmî (cuyo apellido se tradujo al latín en la palabra *algoritmus*), quien era un matemático persa del siglo IX, que enunció paso a paso las reglas para sumar, restar, multiplicar y dividir números decimales. En este mismo contexto también debe recordarse el algoritmo de Euclides obtenido 300 a.C.

Esto significa que para que un algoritmo pueda ser correctamente ejecutado, cada uno de sus pasos debe estar expresado de tal forma que el procesador sea capaz de entenderlos y ejecutarlos adecuadamente.

Ahora bien, el término de algoritmo puede tener diferentes connotaciones dependiendo del contexto del que se hable, y en nuestro caso el contexto es el de programación, esto es, el de utilizar a la computadora como herramienta para la resolución de problemas.

En este sentido, se definirá un **algoritmo** como un conjunto finito de instrucciones que especifican la secuencia ordenada de operaciones a realizar para resolver un problema.

En base a lo anterior, si el procesador del algoritmo es una computadora, el algoritmo debe estar expresado en forma de un **programa**, el cual se escribe en un **lenguaje de programación**⁶, y a la actividad de expresar un algoritmo en un lenguaje de programación determinado se le denomina **programar**.

Cada paso del algoritmo se expresa mediante una instrucción o sentencia en el programa, por lo que un **programa** consiste en una secuencia de instrucciones, en donde cada una de las cuales especifica ciertas operaciones a realizar por parte de la computadora.

A.2.2. Uso de la computadora en la resolución de problemas

En general, se escriben algoritmos para resolver problemas que no son tan fáciles de resolver a primera vista, y de los que se necesita especificar el conjunto de acciones que se llevarán a cabo para su resolución. Además, como lo que interesa es resolver problemas utilizando a la computadora como medio, los algoritmos tendrán como finalidad el ser traducidos en programas, razón por la cual es conveniente el mencionar el **proceso general de resolución de problemas**, mismo que abarca, desde que se dispone de un algoritmo, hasta que la computadora lo ejecuta.

El proceso general de resolución de problemas mostrado en la Tabla A.1, ha sido adaptado de [Abellanas].

Existen diferentes formas de traducir un algoritmo a un programa. En el proceso mostrado en la Tabla A.1, se ha escogido la representación en un

⁶Un lenguaje de programación es lenguaje artificial que se utiliza para expresar programas de computadora.

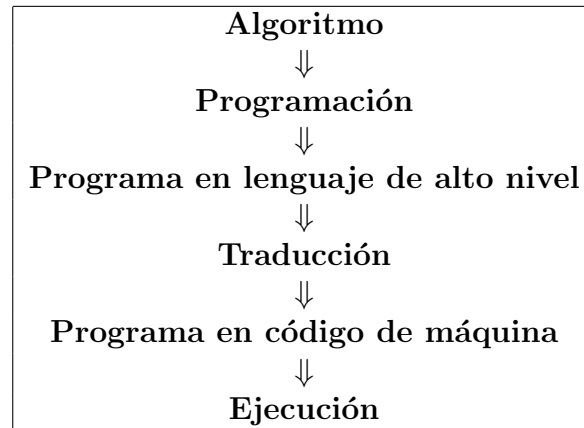


Tabla A.1: Proceso general de resolución de problemas

lenguaje de programación de alto nivel, debido a que los lenguajes de éste tipo, proporcionan un mayor nivel de abstracción y semejanza con el lenguaje natural (inglés).

Es importante recordar que las computadoras tienen su propio lenguaje (binario), por lo que es necesario un proceso de traducción (realizado por el compilador), para que se traduzca el conjunto de sentencias escritas en un lenguaje de programación de alto nivel (**código fuente**), a un conjunto de instrucciones que sean compresibles para la computadora (**código de máquina u objeto**).

Aunque lo que sucede en cada una de las etapas del proceso involucrado, es más complejo que lo que aquí se ha descrito, si todo está bien, el resultado de este proceso será la ejecución del programa basado en el algoritmo.

Teniendo en consideración lo anterior, debería ser claro que el papel del algoritmo es fundamental, ya que sin algoritmo no puede haber programas, y sin programas, no hay nada que ejecutar en la computadora.

Es importante mencionar y enfatizar también, un algoritmo es independiente tanto del lenguaje de programación en que se expresa, como de la computadora en la que se ejecuta, por lo que no deberían ser diseñados en función de ello.

Finalmente, otro aspecto muy importante dentro de los algoritmos, es el del análisis. El análisis y diseño formal de algoritmos es una actividad intelectual considerable y no trivial, ya que el diseño de buenos algoritmos requiere creatividad e ingenio, y no existen reglas para diseñar algoritmos

en general, es decir, todavía no existe un algoritmo que diseñe un algoritmo para un problema determinado. El análisis de algoritmos está fuera de los alcances de este libro.

A.2.3. Cinco importantes condiciones de un algoritmo

Los algoritmos, además de ser un conjunto finito de reglas que dan lugar a una secuencia de operaciones para resolver un tipo específico de problemas, deben cumplir con cinco importantes condiciones [Abellanas] mismas que son descritas en la Tabla A.2.

Las cinco condiciones mostradas en la Tabla A.2, reducen significativamente el espectro tan amplio que hasta ahora se ha manejado como algoritmo. Así por ejemplo, aunque una receta de cocina podría considerarse como un algoritmo, es común encontrar expresiones imprecisas en ella que dan lugar a ambigüedad (violando la condición 2), tales como “añádase una pizca de sal”, “batir suavemente”, etc., invalidando con ello la formalidad de un algoritmo dentro del contexto que nos compete.

A.2.4. Estructura de un algoritmo

Aunque no existe una única forma de representar un algoritmo⁷, la estructura general de un algoritmo debería ser como la mostrada en la Tabla A.3.

Cuando se escribe un algoritmo es recomendable que, aunado a la estructura general y partiendo de la descripción del problema y de un análisis inicial, se determine la entrada, la salida y el proceso general de solución del algoritmo.

Con la finalidad de mostrar dicho procedimiento, a continuación se describirá brevemente un ejemplo clave y clásico en los algoritmos: el algoritmo de Euclides.

Definición del problema

Dados dos números enteros positivos m y n , encontrar su máximo común divisor (MCD), es decir, el mayor entero positivo que divide a la vez a m y a n .

⁷Vea [Abellanas], [Deitel], [Joyanes], [Ullman], [Wirth], etc. cada uno tiene formas diferentes aunque esencialmente iguales de representar un algoritmo.

1. Finitud	Un algoritmo tiene que acabar siempre tras un número finito de pasos. (Un procedimiento que tiene todas las características de un algoritmo, salvo que posiblemente falla en su finitud, se conoce como método de cálculo.)
2. Definibilidad	Cada paso de un algoritmo debe definirse de modo preciso; las acciones a realizar deben estar especificadas para cada caso rigurosamente y sin ambigüedad.
3. Conjunto de entradas	Debe existir un conjunto específico de elementos, donde cada uno de ellos, constituye los datos iniciales de un caso particular del problema que resuelve el algoritmo, y a este conjunto se le denomina conjunto de entradas del algoritmo.
4. Conjunto de salidas	Debe existir un conjunto específico de elementos, donde cada uno de ellos, constituye la salida o respuesta que debe obtener el algoritmo para los diferentes casos particulares del problema. A este conjunto se le denomina conjunto de salidas del algoritmo, y para cada entrada del algoritmo, debe existir una correspondiente salida asociada, que constituye la solución al problema particular determinado por dicha entrada.
5. Efectividad	Un algoritmo debe ser efectivo. Todas las operaciones a realizar por el algoritmo deben ser lo bastante básicas para poder ser efectuadas de modo exacto, y en un lapso de tiempo finito por el procesador que ejecute el algoritmo.

Tabla A.2: Cinco condiciones que debe cumplir un algoritmo

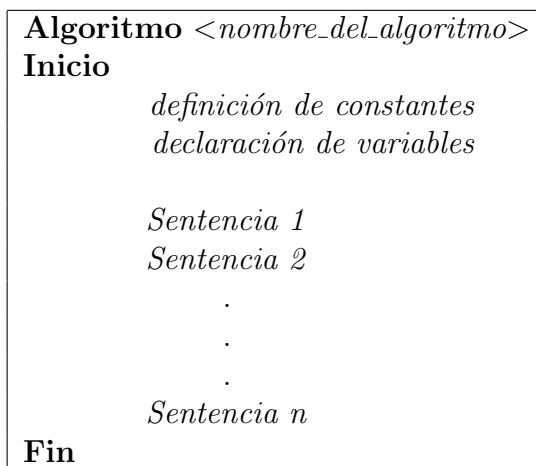


Tabla A.3: Estructura general de un algoritmo

- Entrada:** Dos números enteros positivos: m y n .
- Salida:** Un número que representa el MCD (Máximo Común Divisor) de m y n .
- Proceso:** La solución está basada en el residuo de la división de los operandos m y n . Si el residuo es 0 entonces hemos terminado, si no, habrá que hacer intercambio de valores y continuar con el proceso.

El ejemplo del algoritmo de Euclides se utilizará a lo largo del apéndice para mostrar el proceso de solución que se debería seguir para resolver un problema. A partir de éste análisis inicial, la idea será entonces ir particularizando la entrada, la salida y el proceso de solución en un refinamiento progresivo, de tal forma que eventualmente se genere un algoritmo más específico, definido, y funcional.

A.2.5. Pruebas de algoritmos

Una vez que se ha generado un algoritmo que parece correcto, una de las partes más importantes dentro de su diseño, es la referente a las pruebas. La fase de la validación de los datos de entrada para el algoritmo, es un aspecto sumamente importante de su diseño.

Una vez que se tiene una solución algorítmica de un problema, no se debería suponer que funcionará bien siempre. En el diseño del algoritmo se deben considerar al menos algunos casos de prueba. En éste sentido, es habitual que el dominio de trabajo de un algoritmo sea un conjunto de elementos, en cuyo caso sería deseable el saber por ejemplo ¿cómo se comporta el algoritmo en los límites del conjunto?, dado un mismo dato de entrada, ¿se obtiene siempre la salida esperada?, dado un mismo dato de entrada, ¿se obtiene siempre la misma salida?, etc.

La fase de prueba de los algoritmos es una parte fundamental dentro del diseño del mismo, y debe ser una práctica habitual de cualquier programador que se jacte de serlo, ya que es una importante técnica de programación, y un principio esencial de la Ingeniería de Software.

Las técnicas y métodos formales de pruebas están fuera de los alcances de este texto, pero a este nivel, es suficiente saber que es conveniente realizar pruebas sobre los algoritmos desarrollados. Un libro ampliamente recomendable en cuanto a técnicas básicas de pruebas y estilo de programación es [Kernighan1].

A.3. Diagramas de flujo

Un diagrama de flujo es una notación algorítmica de tipo gráfica.

Un **diagrama de flujo** es una herramienta gráfica de descripción de algoritmos, que se caracteriza por utilizar un conjunto de símbolos gráficos, para expresar de forma clara los flujos de control o el orden lógico en el que se realizan las acciones de un algoritmo.

Aunque existe en la literatura una amplia variedad de representaciones para los símbolos utilizados en los diagramas de flujo, en este texto se adoptarán sólo cinco, mismos que se presentan en la Figura A.1.

A.3.1. Estructuras de control

Esta sección muestra los diagramas de flujo de las estructuras de control, para más detalles respecto a las estructuras de control, refiérase por favor a la sección A.1.1.






Símbolo	Significado/Función
	Inicio/Fin/Conector
	Proceso
	Entrada/Salida
	Decisión
	Flujo de ejecución

Figura A.1: Símbolos y su significado en los diagramas de flujo

Estructura secuencial

La Figura A.2 muestra el diagrama de flujo que representa a la estructura de control secuencial.

Estructuras de selección

La Figura A.3 muestra los diagramas de flujo de las estructuras de selección.

Puede observarse en la Figura A.3, que en la **estructura de selección simple** se evalúa la *condición*, y si ésta es verdadera, se ejecuta un determinado grupo de *sentencias*; en caso contrario, las sentencias son ignoradas.

Por otro lado, en la **estructura de selección doble**, cuando la *condición* es verdadera, se ejecutará un determinado grupo de *sentencias*, y si es falsa se procesará otro grupo diferente de *sentencias*.

Por último, en la **estructura de selección múltiple** se ejecutarán unas sentencias u otras según sea el valor que se obtenga al evaluar una expresión representada por el *indicador*. Se considera que dicho resultado debe ser de tipo ordinal, es decir, de un tipo de datos en el que cada uno de los elementos que constituyen el tipo, excepto el primero y el último, tiene un

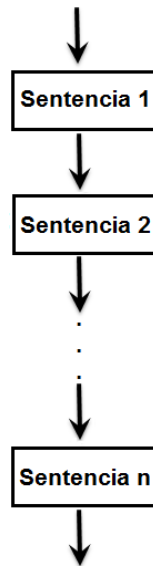


Figura A.2: Estructura secuencial en diagrama de flujo

único predecesor y un único sucesor.

Estructuras de repetición

La Figura A.4 muestra las estructuras de repetición básicas.

Lo que caracteriza a la estructura de repetición “mientras” (*while*), es que las *sentencias* del cuerpo del ciclo se procesan cuando la condición es verdadera, además de que la *condición* es verificada al principio, de donde se deduce que las sentencias se podrán ejecutar de 0 a n veces.

Por otro lado, en la estructura de repetición “hacer mientras” (*do-while*), las *sentencias* del cuerpo del ciclo se ejecutan al menos una vez, y continúan repitiéndose hasta que la condición sea falsa. La verificación de la *condición* se realiza al final del ciclo, por lo que se deduce que las sentencias se ejecutarán de 1 a n veces.

A.3.2. Diagrama de flujo del algoritmo de Euclides

La Figura A.5 muestra el diagrama de flujo para el problema propuesto en la sección A.2.4. La solución a dicho problema está determinada por el algoritmo de Euclides.

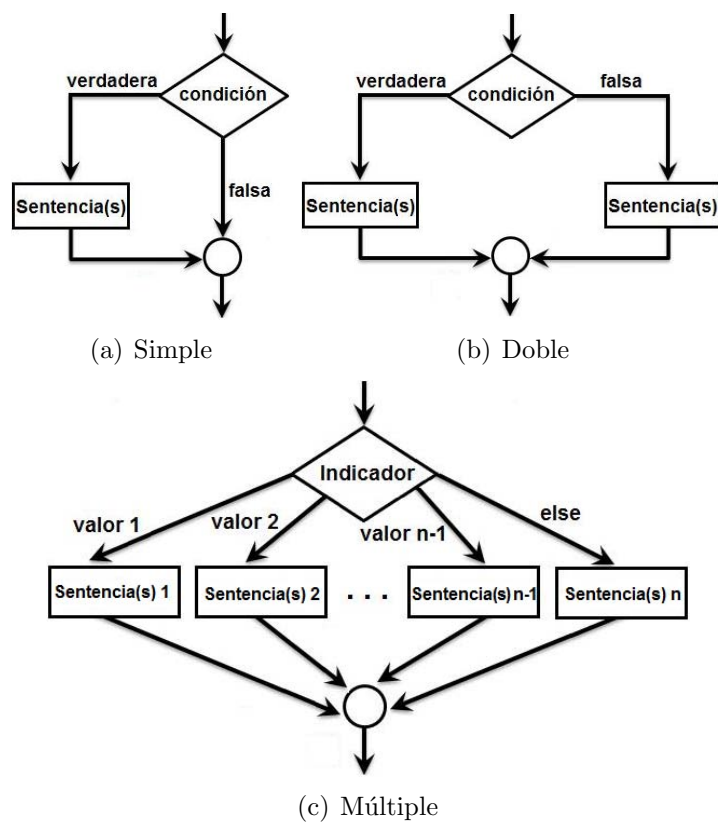


Figura A.3: Estructuras de selección en diagrama de flujo

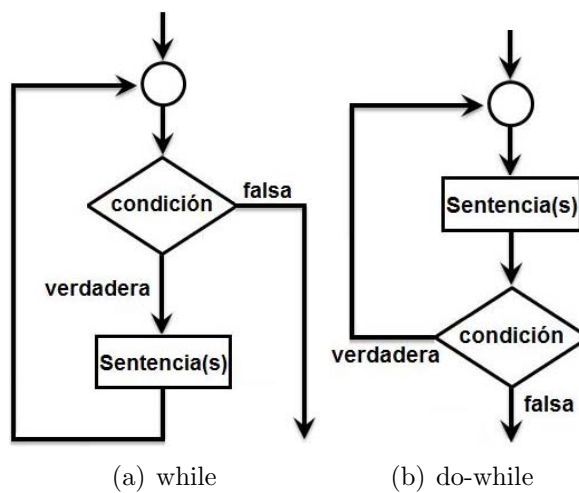


Figura A.4: Estructuras de repetición en diagrama de flujo

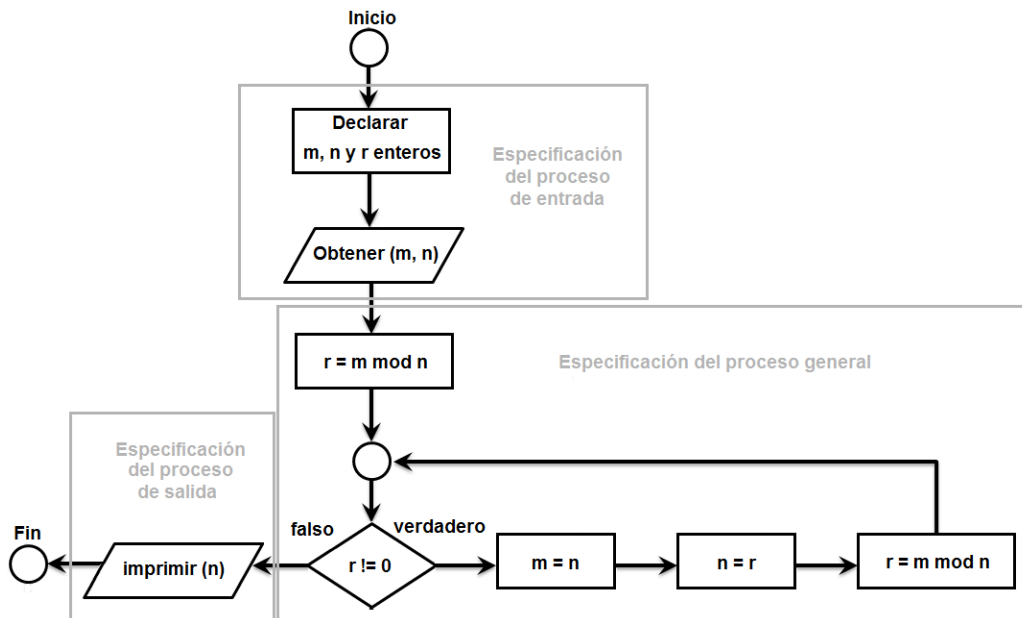


Figura A.5: Diagrama de flujo para el algoritmo de Euclides

Para ser congruentes con la propuesta de solución realizada en la sección A.2.4, además del algoritmo, se ha indicado en recuadros de menor intensidad, las especificaciones del proceso de entrada, del proceso de salida, y del proceso general de solución.

A.4. Pseudocódigo

El **pseudocódigo** es otro tipo de notación algorítmica textual, y es un lenguaje artificial de especificación de algoritmos, caracterizado por:

1. Mantener una indentación o sangría adecuada para la fácil identificación de los elementos que lo componen.
2. Permitir la declaración de los datos (constantes y/o variables) manipulados por el algoritmo.
3. Disponer de un conjunto pequeño y completo de palabras reservadas⁸.

⁸Las palabras reservadas son propias del lenguaje artificial, y no pueden ser utilizadas como identificadores de variables o módulos.

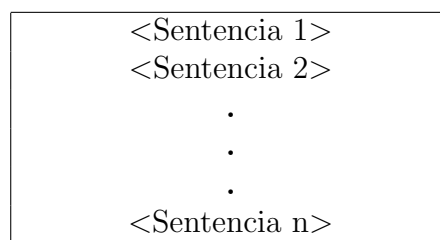


Tabla A.4: Estructura secuencial en pseudocódigo

El pseudocódigo se concibió para superar las dos principales desventajas del diagrama de flujo:

1. Lento de crear.
2. Difícil de modificar.

Sin embargo, los diagramas de flujo son una excelente herramienta para el seguimiento de la lógica de un algoritmo, así como para transformar con relativa facilidad dichos algoritmos en programas.

A.4.1. Estructuras de control

Esta sección describe brevemente en su representación en pseudocódigo, las estructuras de control. Todas las estructuras de control de esta sección siguen las mismas características descritas con anterioridad en las secciones A.1.2 y A.3.1.

Estructura secuencial

La Tabla A.4 ilustra la representación en pseudocódigo de la estructura secuencial.

Estructuras de selección

Las estructuras de selección simple, doble y múltiple, se muestran en las Tablas A.5, A.6 y A.7 respectivamente.

if (condición) <Sentencia(s)>

Tabla A.5: Estructura de selección simple en pseudocódigo

if (condición) <Sentencia(s) 1> else <Sentencia(s) 2>
--

Tabla A.6: Estructura de selección doble en pseudocódigo

if (condición 1) <Sentencia(s) 1> else if (condición 2) <Sentencia(s) 2> . . . else <Sentencia(s) n>	case (indicador) of <valor 1>:<Sentencia(s) 1> <valor 2>:<Sentencia(s) 2> . . . <valor n>:<Sentencia(s) n> < default >:<Sentencia(s) n+1> end case
---	--

Tabla A.7: Estructura de selección múltiple en pseudocódigo

while (condición) <Sentencia(s)> end while
--

Tabla A.8: Estructura de repetición hacer mientras (while)

do <Sentencia(s)> while (condición)

Tabla A.9: Estructura de repetición repetir mientras (do-while)

Estructuras de repetición

Finalmente, las estructuras de repetición “mientras” (*while*), y “hacer mientras” (*do-while*), son presentadas en la Tablas A.8 y A.9 respectivamente.

A.4.2. Pseudocódigo del algoritmo de Euclides

Esta sección muestra el pseudocódigo para el problema propuesto en la sección A.2.4.

Se deja como ejercicio al lector el comparar el algoritmo propuesto en la Tabla A.10, con el diagrama de flujo de la Figura A.5.

A.5. Pruebas de escritorio

La Figura A.5 y la Tabla A.10 muestran, en notaciones distintas, el mismo algoritmo: el algoritmo de Euclides o MCD.

Una **prueba de escritorio** es un tipo de prueba algorítmica, que consiste en la validación y verificación del algoritmo a través de la ejecución de las sentencias que lo componen (proceso), para determinar sus resultados (salida) a partir de un conjunto determinado de elementos (entrada).

Suponga que desea saber cual es el MCD de quince y cuatro. Una prueba de escritorio para el algoritmo MCD consistiría, en primer lugar, la identificación de las variables declaradas:

Algoritmo MCD
Inicio
Variables
m, n, r de tipo entero
Obtener (m, n)
r = m mod n
while (r \neq 0)
m = n
n = r
r = m mod n
end while
imprimir (n)
Fin

Tabla A.10: Algoritmo de Euclides en pseudocódigo

```

m =
n =
r =

```

La sentencia *Obtener*(m , n) definiría los valores de las variables m y n respectivamente, que en base a lo que se desea saber sería:

```

m = 15
n = 4
r =

```

mientras que r se define en base a la expresión $r = m \bmod n$, es decir:

```

m = 15
n = 4
r = 3

```

el siguiente paso, es verificar la condición del *while*, y como es verdadera, se tendría la siguiente secuencia de valores para las variables, la cual es generada dentro del ciclo:

```

m = 15, 4, 3
n = 4, 3, 1
r = 3, 1, 0

```

de donde puede verse, que el MCD de 15 y 4 es 1 (n).

Asegúrese de comprender el proceso descrito y repítalo. Utilice ahora los mismos valores pero intercambiados ¿qué sucede? ¿qué secuencia de valores se genera? ¿cuál es el MCD? ¿cuál es ahora el MCD de 15 y 10? ¿cuál es el MCD de 10 y 15?

Pruebe con otros valores hasta que comprenda el funcionamiento del algoritmo, y compruebe los resultados de sus pruebas de escritorio con los algoritmos de la Figura A.5 y de la Tabla A.10.

A.6. Diseño básico de programas estructurados

La esencia del análisis mostrado a continuación, está basado principalmente en las reglas para la formación de programas estructurados propuestas en [Deitel].

Al conectar de forma arbitraria y/o poco cuidadosa los símbolos de los diagramas de flujo, se puede incurrir en diagramas de flujo no estructurados como el de la Figura A.6.

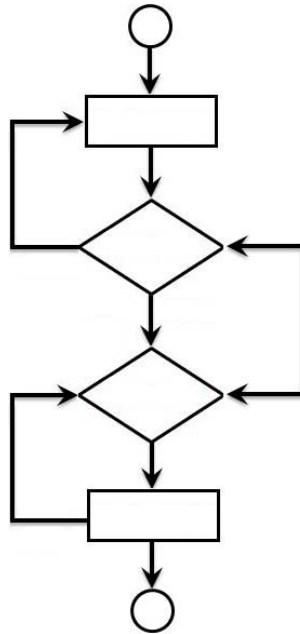
La generación de diagramas de flujo estructurados, requiere de utilizar estructuras de control con una única entrada, y una única salida⁹, de tal manera que sólo haya una forma de entrar y una de salir de la estructura de control.

A.6.1. Reglas para la formación de algoritmos estructurados

La Tabla A.11 muestra las reglas para la construcción de algoritmos estructurados, las cuales podrían denominarse como “*Algoritmo para la construcción de algoritmos estructurados*”.

En la Figura A.7 se muestra la regla 1 y la aplicación repetida de la regla 2. Observe que la regla 1 es perfectamente consistente con el proceso general de resolución de algoritmos propuesto en la sección A.2.4.

⁹Las estructuras mostradas en las secciones anteriores cumplen con esto, y antes de continuar se debería estar convencido de ello. Observe que, aunque la estructura se selección múltiple mostrada en la Figura A.3 c) parece tener más de una salida, sólo una de ellas es finalmente procesada, y que independientemente de la sentencia(s) que haya(n) sido

Figura A.6: Diagrama de flujo **no** estructurado

1. Empiece con el diagrama de flujo más simple.
2. Cualquier rectángulo (acción, sentencia, etc.) puede ser reemplazado por dos rectángulos de manera secuencial. Ésta es la regla de apilamiento.
3. Cualquier rectángulo puede ser reemplazado por cualquier estructura de control. Esta es la regla de anidamiento.
4. Aplicar de manera sucesiva las reglas 2 y 3.

Tabla A.11: Reglas para la formación de algoritmos estructurados

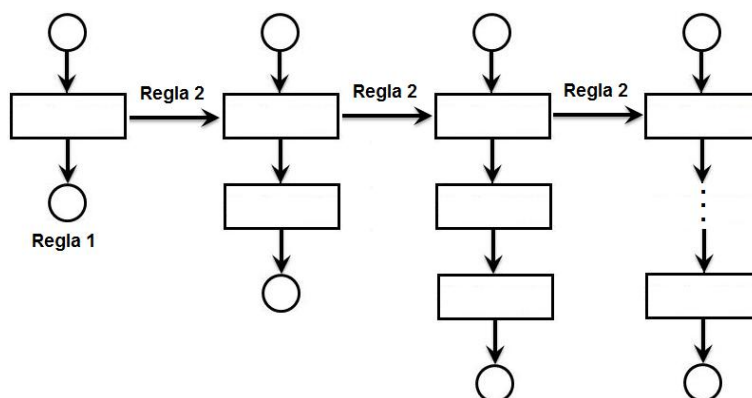


Figura A.7: Aplicación de las reglas 1 y 2

La aplicación de las reglas de la Tabla A.11, siempre resulta en un diagrama de flujo estructurado con una apariencia clara y de bloques constructivos [Deitel].

Por otro lado, la Figura A.8 muestra la aplicación de la regla de anidamiento al diagrama de flujo más simple que puede haber. Note que los bloques han sido substituidos por estructuras de selección doble y de repetición.

Por último, es importante resaltar que la aplicación de la regla 4 genera estructuras más grandes, más complejas y más profundamente anidadas. Adicionalmente, cabe mencionar también que los diagramas de flujo que resultan de aplicar las reglas de la Tabla A.11, constituyen el conjunto de todos los diagramas de flujo estructurados posibles, y por lo tanto, también el conjunto de todos los posibles programas estructurados [Deitel].

A.7. Consideraciones finales

La programación estructurada no debe confundirse con el conocimiento de un lenguaje de programación. La programación estructurada es un modelo de programación, y es independiente del lenguaje que se utilice para su implementación.

El estudio y el conocimiento de técnicas, así como el desarrollo de algoritmos previos a la fase de implementación, no sólo es una buena práctica de programación, sino un principio esencial de la Ingeniería de Software.

seleccionada(s), su salida converge a un único punto de salida representado por el conector.

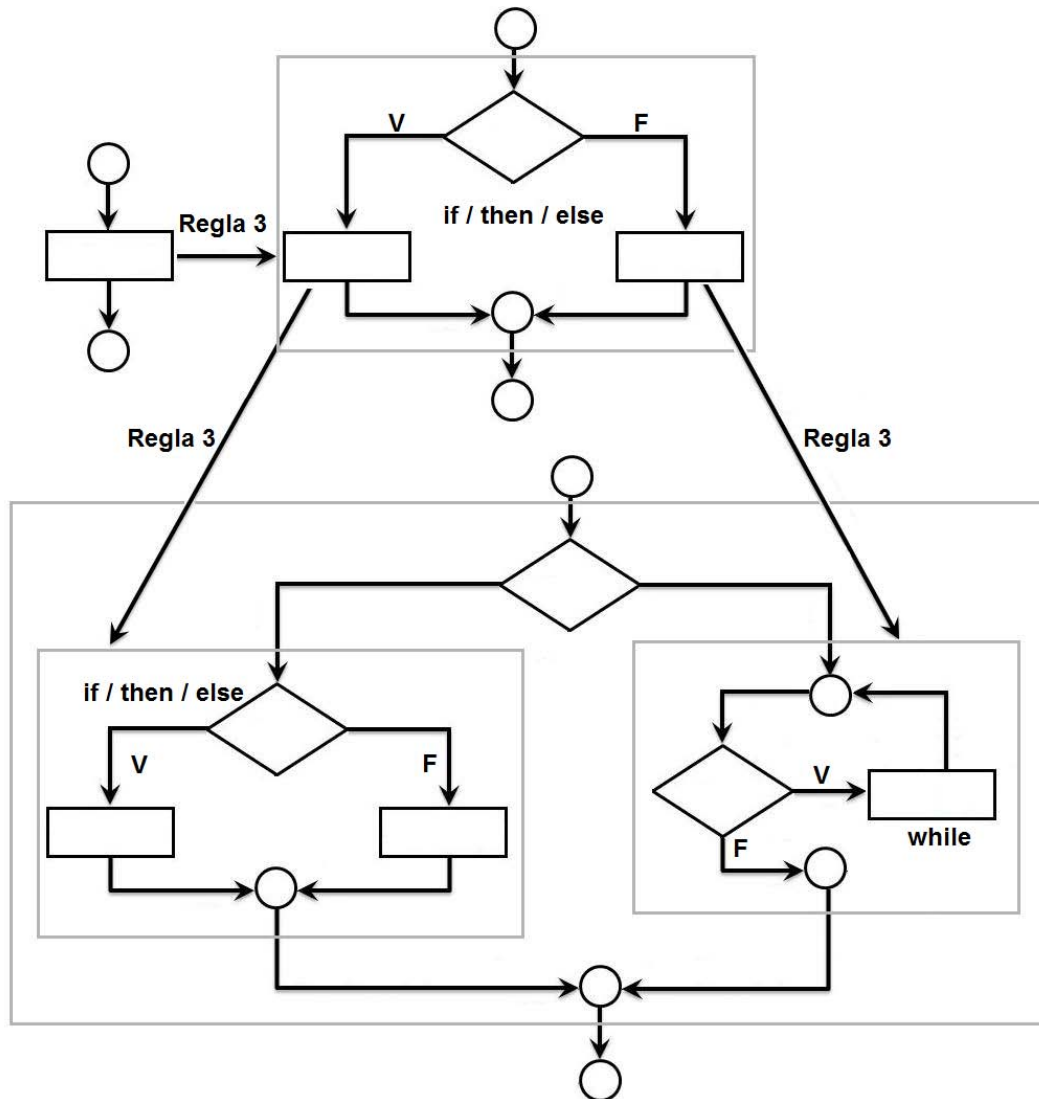


Figura A.8: Aplicación de la regla 3

Por todo lo anterior, si lo que se quiere es llevar a buen fin sus proyectos de programación y evitar dolores de cabeza innecesarios, debe tomar en consideración que, antes de enfrentarse en una lucha encarnizada, teniendo a la computadora como su adversario y al lenguaje de programación como arma, debería, antes de escribir una sola primera línea de código, hacer un análisis profundo del problema a resolver, elaborar un plan de solución general, especificar gradualmente dicho plan, y realizar un conjunto de pruebas representativas.

Apéndice B

Compilación de programas

Básicamente, existen dos tipos de compiladores para el lenguaje de programación C:

1. Propietarios: compiladores que requieren de licencia para su uso.
2. Código abierto o software libre (*open source*).

En este apéndice se describirán brevemente los elementos básicos para compilar, en línea de comandos, programas escritos en el lenguaje C utilizando el compilador de GNU¹ GCC. Sin embargo, dependiendo de la plataforma de software (sistema operativo) de su preferencia, se recomienda ampliamente el uso de un entorno de desarrollo (IDE), seleccione aquel que más se ajuste a sus necesidades, interés y presupuesto.

B.1. El compilador GCC

GCC es software libre, y lo distribuye la *Free Software Foundation* (FSF) bajo la licencia general pública GPL (por sus siglas en inglés). Originalmente GCC significaba *GNU C Compiler* (compilador GNU de C), porque sólo compilaba el lenguaje C, sin embargo, GCC fue posteriormente extendido para compilar también otros lenguajes de programación.

¹GNU es un acrónimo recursivo que significa GNU No es Unix (*GNU is Not Unix*), y es un proyecto iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre.

GCC es parte del proyecto GNU, y tiene como objetivo principal, mejorar el compilador usado en todos los sistemas GNU.

El desarrollo de GCC usa un entorno de desarrollo abierto, y soporta distintos tipos de plataformas, con la intención de asegurar que GCC y los sistemas GNU funcionen en diferentes arquitecturas y en diferentes entornos; pero sobre todo, para extender y mejorar las características de GCC.

B.2. Compilación de programas con GCC

Esta sección se centra en explicar los elementos básicos para una compilación en línea de comandos para sistemas de tipo GNU/Linux, sin entrar en detalles de los comandos necesarios para cambiarse de directorio, listar archivos, etc. El símbolo siguiente representa el símbolo del sistema al que se estará haciendo referencia de aquí en adelante:

\$

Ahora bien, dependiendo de la distribución de Linux que utilice, el símbolo del sistema podría ser distinto, pero estos son detalles irrelevantes a la compilación de programas en C, y no se detallarán tampoco aquí.

Tome en cuenta que las descripciones siguientes, asumen que tiene instalada alguna versión del compilador de GCC, los detalles de instalación y configuración de rutas (*paths*), quedan fuera de los propósitos de este apéndice.

Por otro lado, considere que para todos los ejemplos descritos, se supondrá que tanto el programa fuente como el ejecutable, se generarán en el directorio de trabajo, es decir, el directorio en donde se ejecuten los comandos.

B.2.1. Comprobando la instalación de GCC

Para compilar programas en C, hay que usar el compilador de C, y para usar el compilador de C, éste debe estar instalado.

La mayoría de los compiladores reportan sus mensajes en inglés, aunque existen algunos mecanismos e implementaciones que los reportan en castellano. El compilador que se utilizará reporta los mensajes en inglés, y así se

presentarán en los ejemplos. Para determinar si tiene instalado el compilador GCC en su sistema, escriba lo siguiente² en el símbolo del sistema:

```
$ gcc
gcc: no input files
```

Si la salida que obtiene no es semejante a la anterior, podría deberse básicamente a dos cosas:

1. No se tiene instalado el compilador GCC.
2. Tiene instalado el compilador GCC, pero las rutas de acceso a programas están mal, o no cuenta con los permisos requeridos para la ejecución del compilador.

En cualquiera de los dos casos anteriores, no se podrán compilar programas en C, y deberá corregir dicha situación antes de continuar.

B.2.2. Compilación básica

Una vez que se ha comprobado que se cuenta con el compilador GCC, lo siguiente es compilar un programa en C.

La sección 2.1 muestra, en el Ejemplo 2.1, un programa bastante simple de nombre *bienvenido.c*, mismo que se utilizará para el siguiente ejemplo de compilación:

```
$ gcc bienvenido.c
$
```

Aquí es importante mencionar algunas cosas:

1. Note que la extensión del archivo es “.c”. Esto es importante, debido a que le indica al compilador GCC el tipo de compilador a utilizar, en este caso: el compilador de C.
2. La impresión del símbolo del sistema (\$) sin ningún mensaje adicional, implica que la compilación fue exitosa, y que no existe ningún tipo de problema: éste es el mejor de los casos.

²El uso de minúsculas es importante. De hecho, asegúrese de escribir los ejemplos tal y como aparecen en el texto.

3. Si existen errores, estos se reportarán antes de la siguiente aparición del símbolo del sistema, precedidos del número de línea en donde se detectó el problema. Dichos errores tendrán que ser corregidos en algún editor de texto, se deberán guardar los cambios realizados, y se deberá repetir nuevamente la compilación del programa fuente, hasta que no aparezcan más errores (como se describe en el punto anterior).
4. La compilación exitosa (sin errores) de un programa en C, genera un archivo ejecutable de nombre “*a.out*”.

La ejecución del programa anterior, junto con la salida del programa, se muestra a continuación:

```
$ ./a.out
Bienvenido a C!
$
```

B.2.3. Ligado de bibliotecas

Algunos programas utilizan bibliotecas de funciones que tienen que ser explícitamente ligadas a dichos programas. Un caso muy común, es el de la biblioteca de funciones *math.h*.

Si su compilador no reporta nada, compile tal y como se ha descrito hasta ahora, sin embargo, si al compilar alguna función matemática, se reporta algo como lo siguiente:

```
$ gcc raices.c
/tmp/ccokd6n.o: In function 'main':
raices.c:(.text+0xfb): undefined reference to 'sqrt'
raices.c:(.text+0x137): undefined reference to 'sqrt'
collect2: ld returned 1 exit status
$
```

tendrá que escribir:

```
$ gcc raices.c -lm
$
```

Lo cual le indica al compilador GCC que ligue o asocie la biblioteca de funciones matemáticas (*link math (lm)*) al programa fuente *raices.c*.

B.2.4. Generación de ejecutables

Una vez que se ha escrito, compilado, corregido y depurado un programa en C, en muchas ocasiones resulta deseable tener una imagen ejecutable de dicho programa con un nombre más significativo y representativo que el de *a.out*. En dichas situaciones, utilice la opción **-o** del compilador GCC para generar la salida (*output*) en un archivo específico:

```
$ gcc bienvenido.c -o bienvenido
$
```

La sentencia anterior le indica la compilador GCC que compile el archivo *bienvenido.c* y genere, si todo está bien, el ejecutable en el archivo *bienvenido*, de tal forma que la ejecución del programa cambiaría de:

```
$ ./a.out
Bienvenido a C!
$
```

a:

```
$ ./bienvenido
Bienvenido a C!
$
```

B.3. Redireccionamiento

Los programas de una computadora, leen normalmente sus datos de la entrada estándar (habitualmente asociada al teclado), y escriben sus datos en la salida estándar (habitualmente asociada a la pantalla), sin embargo, en algunas ocasiones es deseable que los procesos obtengan sus datos desde archivos, o que la información y los datos que generan sean almacenados en un archivo.

Un mecanismo básico para lograr lo anterior es el de redireccionamiento. Por medio del redireccionamiento, es posible hacer que un programa procese los datos de entrada desde un archivo, y/o hacer que los datos o la información que genere sean enviados a un archivo, sin la necesidad de incluir instrucciones explícitas de gestión de archivos dentro del programa.

B.3.1. Redireccionamiento de la entrada estándar

Considere el Ejemplo 3.17 del Capítulo 3. Sus detalles no se analizarán aquí, sino que se utilizará como referencia para mostrar el redireccionamiento de la entrada estándar.

Asumiendo que el programa fuente se llama *cuentaVocales.c*, la siguiente secuencia de comandos muestra los detalles desde su compilación, hasta el redireccionamiento de la entrada estándar:

```
$ gcc cuentaVocales.c -o cuentaVocales
$ ./cuentaVocales < archivo.txt
```

El símbolo “<” le indica al sistema operativo, que la entrada estándar no será el teclado, sino el archivo *archivo.txt*, es decir, los datos no se procesarán desde el teclado sino desde el archivo.

Observe que tanto la información como los datos que genera el programa *cuentaVocales* seguirán siendo presentados en la salida estándar (pantalla), debido a que no se ha realizado ninguna modificación sobre ella; los detalles de su redireccionamiento se mencionan a continuación.

B.3.2. Redireccionamiento de la salida estándar

El redireccionamiento de la salida estándar es análogo al de la entrada estándar, con la diferencia del símbolo utilizado para el redireccionamiento: “>”.

Continuando con el Ejemplo 3.17, el redireccionamiento de su salida estándar sería:

```
$ ./cuentaVocales > salida.txt
$
```

donde el archivo *salida.txt* contendrá la salida del programa *cuentaVocales*, incluso los mensajes de tipo *prompt*³ que presente.

Note que el símbolo del sistema aparece de inmediato, debido a que no hay nada que reportar en la pantalla.

Finalmente, es importante mencionar que es posible re direccionar al mismo tiempo tanto la entrada como la salida estándar, de tal forma que los datos sean leídos desde un archivo (*archivo.txt*), y almacenados en otro archivo (*salida.txt*), tal y como se muestra a continuación:

³La línea 11 del Ejemplo 3.17 es de éste tipo.

```
$ ./cuentaVocales < archivo.txt > salida.txt  
$
```

Los mecanismos de redireccionamiento del sistema operativo, están estrechamente ligados con los descriptores de archivo que administra, sin embargo, no es la intención del apéndice profundizar en dichos detalles, que si bien son importantes, quedan fuera de los alcances y objetivos del libro.

Por el momento, basta con que comprenda el funcionamiento de los mecanismos de redireccionamiento brevemente descritos en esta sección.

B.4. Consideraciones finales

La compilación de programas en línea es una tarea interesante, pero tome en cuenta que también puede llegar a ser abrumadora, sobre todo para programadores con poca o nula experiencia.

Revise la ayuda en línea:

```
$ gcc --help
```

y el manual:

```
$ man gcc
```

para tener una idea de todas las opciones y modalidades que se tienen al compilar en línea con GCC.

La recomendación es nuevamente en éste sentido, apoyarse de algún entorno de desarrollo que facilite, tanto la tarea de compilación, como la de vinculación de bibliotecas, y la depuración de programas, de tal forma que, cuando se tenga un mejor nivel de experiencia, se puedan explorar la potencialidad y las características, no sólo del compilador GCC, sino de otras útiles herramientas de desarrollo como **gdb**, **make**, etc.

Bibliografía

- [Abellanas] Abellanas, M. y Lodaes D., “Análisis de Algoritmos y Teoría de Grafos”. Macrobit ra-ma.
- [Deitel] Deitel, H. M. y Deitel, P. J., “Cómo Programar en C/C++”, Prentice Hall.
- [Joyanes] Joyanes, Aguilar L., “Metodología de la Programación: Diagramas de Flujo, Algoritmos y Programación Estructurada”, Mc Graw Hill.
- [Kernighan1] Kernighan, B.W. y Pike R., “La Práctica de la Programación”, Prentice Hall.
- [Kernighan2] Kernighan, L., Brian, A.H., y Ritchie, K., Dennis, “El lenguaje de Programación C”, Prentice Hall.
- [Knuth] Knuth, Donald Ervin., “*The Art of Computer Programming*”, Vol. 1, 3rd. ed. Addison-Wesley.
- [Ullman] Ullman, Jeffrey. D., “*Fundamental Concepts of Programming Systems*”, Addison-Wesley Publishing Company Inc.
- [Wirth] Wirth, Niklaus, “Algoritmos y Estructuras de Datos”, Prentice Hall.

Índice Analítico

- bugs*, 7
- cast*, 161
- linker*, 6
- loader*, 7
- pointer*, 145
- prompt*, 17, 43
- streams*, 214
- Integrated Development Environment*,
1
- abstracción, 69, 70, 81, 112, 185, 188,
199
- algoritmo, 2, 245, 246
 - condiciones, 248
 - estructura, 248
 - Euclides, 245, 248
 - pruebas, 251
- análisis
 - léxico, 4
 - semántico, 5
 - sintáctico, 4
- ancho de campo, 20
- apuntador, 143
 - a función, 168
 - arreglos de, 171
 - aritmética, 149, 158
 - arreglo de, 154
 - void, 156
- archivo, 213
- archivos
 - acceso secuencial, 214
 - binario, 225
 - de acceso aleatorio, 226
 - estructura, 226
 - de texto, 220
 - indicador de posición, 224
 - modo de apertura, 216
 - texto, 214
- argumentos main, 163, 165
- arreglo, 108
 - con funciones, 112
 - declaración, 109
 - impresión, 110
 - recorrido, 110
- asignación de memoria, 156
- búsqueda
 - binaria, 119
 - lineal, 118
- bandera, 118
- bloque, 14, 15
- bloque de construcción, 14
- break, 35–37, 57–59
- burbuja, 115, 117
- código
 - de máquina/objeto, 247
 - fuelle, 247
- cadenas, 121
- cargador, 7
- case, 35, 36

- centinela, 55, 244
- char, 16
- ciclo infinito, 45
- comentarios, 12
- compilación, 4
- compiladores, 265
- const, 150
- contador, 53, 244
- continue, 57
- ctype.h, 151

- declaración de variables, 15
- default, 35, 37
- delimitadores de bloque, 13
- depuración, 7
 - break points*, 7
 - step into*, 7
 - step over*, 7
 - trace*, 7
 - watch*, 7
- diagrama de flujo, 251
- difftime, 99

- EOF, 50, 219
- error
 - en tiempo de ejecución, 5
 - fatal, 5
 - preventivo (*warning*), 5
- especificador de formato, 17, 19, 20
- estándar
 - entrada, 21
 - error, 21
 - salida, 21
- estructura, 186
- estructura de control
 - algoritmos, 242
 - anidada, 26, 30, 35
 - do-while, 41–44
 - for, 44, 45
 - if-else, 29–31, 33
 - repetición, 37, 244, 253, 258
 - secuencial, 11, 12, 25, 243, 252, 256
 - selección, 26, 244, 252, 256
 - switch, 35
 - while, 37, 38
- estructura de datos, 108
- estructura general de un programa, 8
- estructuras
 - arreglos de, 198
 - compuesta, 195
 - elemento miembro, 187
- Euclides
 - diagrama de flujo, 253
 - pseudocódigo, 258

- factorial
 - iterativa, 91
 - recursiva, 92
- fclose, 217
- fgets, 219
- fibonacci
 - iterativa, 95
 - recursiva, 97
- FILE, 214, 216, 224
- float, 16
- fopen, 215
- fprintf, 217, 224
- fread, 230
- fseek, 231
- ftell, 232
- función
 - argumentos, 73
 - definición, 71
 - estructura general, 71
 - invocación, 70

- llamado, 70
 - parámetros, 71
 - prototipo, 70
- fwrite, 227
- gcc, 163, 265, 266
- gdb, 24, 271
- getchar, 51, 231
- gets, 219
- GNU/Linux, 51, 69, 220, 266
- IDE, 1, 13, 14, 265
- identificador, 16
- if, 26
- incremento, 39
- int, 16
- invocación, 67, 68
- isalnum, 152
- isalpha, 152
- isdigit, 152
- islower, 152
- ispunct, 152
- isupper, 152
- isxdigit, 152
- lenguaje de programación, 246
- llamado de función, 67, 68
- long, 95
- máximo común divisor, 248
- make, 271
- malloc, 156
- math.h, 67, 268
- matrices, 124
- MCD, 248, 258
- número racional, 188
- números mágicos, 38
- notación compacta, 47
- operador
 - de decremento, 41
 - de des referencia, 146
 - de dirección, 145
 - de incremento, 40
 - flecha, 194
 - punto, 190
 - ternario, 33, 34
- operadores
 - aritméticos, 18
 - lógicos, 52
 - relacionales, 26
- ordenamiento, 114
- palíndromo, 138
- palabra reservada, 16
- parámetros
 - referencia, 147
 - valor, 146
- pos incremento, 40
- pre incremento, 40
- pre procesador, 4
- precedencia, 18
- precisión, 20
- printf, 13, 20
- proceso, 6, 7
- programa, 6, 241, 246
 - estructurado, 7, 243
 - propio, 243
- programación
 - estructurada, 242
 - modular, 69, 70
- programar, 246
- programas estructurados, 242
 - diseño básico, 260
- pruebas, 28
 - de escritorio, 3, 258
- pseudocódigo, 255

- putchar, 123
- rand, 48
- recursividad
 - overhead*, 89, 91
 - caso base, 88
 - criterio de paro, 88
 - definición, 88
- redireccionamiento, 21, 269
- referencia, 143, 146, 147, 168
- relación de recurrencia, 95
- repetición
 - controlada por centinela, 52, 55
 - controlada por contador, 52, 53
- rewind, 224, 232
- scanf, 17, 191
- secuencias de escape, 13
- sentencia compuesta, 14
- Sistema Operativo, 13
- size_t, 48
- sizeof, 23, 156
- sqrt, 67, 68
- srand, 48
- static, 81, 135
- streams, 21
- struct, 186, 187
- teorema de Böhm y Jacopini, 243
- time, 48
- time.h, 99
- tipo de dato, 16, 17
- token, 4, 16
- tolower, 152
- torres de Hanoi, 100
- toupper, 152
- typedef, 186, 188, 190
- union, 212
- Unix, 220
- unsigned, 104
- variable
 - ámbito, 78
 - ámbito interno, 79
 - alcance, 78
 - automática, 80
 - estática, 81
 - global, 78
 - local, 74, 77, 78, 80
 - visibilidad, 78
- variable de control, 39
- vectores, 108
- vinculador, 6
- void, 75

Agradecimientos

A *Thelma* por no permitirme quitar el dedo del renglón, y por su invaluable ayuda para la realización de imágenes, fórmulas, diagramas, y otros elementos que conforman este libro; pero sobre todo, por su loable paciencia y capacidad para tolerar mi carácter, sin todo ello, este libro seguiría siendo sólo un pendiente más...

A *Tomás Balderas Contreras* por escribir el prólogo y por sus sugerencias durante el desarrollo del libro.

A *Ricardo Pérez Aguila* porque sin saberlo ni proponérselo, me indujo a dar forma por fin a las ideas y método que conforman este libro. También le agradezco sus sugerencias y recomendaciones respecto a la publicación.

A todos mis estudiantes que me ayudaron implícitamente con sus preguntas, dudas y observaciones, ya que gracias a ello pude corregir y adecuar mejor los programas de ejemplo y robustecer el método.

Por último, pero no por ello menos importante, a mis padres *María Luisa* y *Rodolfo*, ya que gracias a la educación que recibí y a su guía, pude tener una formación profesional entre otras tantas cosas: siempre estaré en deuda con ustedes. . .

Acerca del Autor

Ricardo Ruiz Rodríguez nació en la ciudad de Puebla, Pue., México. Actualmente y desde el año 2002, es profesor investigador adscrito al Instituto de Computación en la Universidad Tecnológica de la Mixteca (UTM), en Huajuapán de León, Oaxaca, México, y cuenta con más de 14 años de experiencia como docente.

El maestro Ricardo además de la docencia, se ha desempeñado también como desarrollador de software en la industria.

Entre sus intereses actuales se encuentran los métodos de enseñanza de la disciplina computacional, la música, y la música por computadora, pero se ha desarrollado académicamente en áreas como la Ingeniería de Software, la Interacción Humano-Computadora, y el Paradigma Orientado a Objetos.

El autor tiene la Licenciatura en Ciencias de la Computación por la Benemérita Universidad Autónoma de Puebla (BUAP) y la Maestría en Ciencias con especialidad en Ingeniería en Sistemas Computacionales, por la Universidad de las Américas-Puebla (UDLA-P).

Información adicional del autor, así como el material relacionado con el libro, se puede consultar en su página personal:

<http://sites.google.com/site/ricardoruizrodriguez/>

