

Notas de Programación Estructurada

María Lizbeth Gallardo López y
Pedro Lara Velázquez

Agosto - 2012

Índice general

1. Arquitectura de computadoras	1
1.1. Modelo actual de la Computadora	1
1.1.1. Unidad central de procesamiento	2
1.1.2. Memoria	3
1.1.3. Dispositivos de entrada y salida de datos	5
1.2. Software de base	5
2. Algoritmos	9
2.1. Algoritmo	9
2.2. Programación	10
2.3. Programación estructurada	11
2.4. Diagramas de flujo	12
3. Lenguaje C	13
3.1. Lenguaje C	13
3.1.1. Tipos de datos	13
3.1.2. Variables y constantes	14
3.1.3. Operadores	15
4. Estructuras de decisión	19
4.1. Decisión simple	19
4.2. Decisión binaria	21
4.3. Decisión encadenada	23
4.4. Decisión anidada	26
4.5. Decisión múltiple	29
5. Funciones	33
5.1. Importancia de dividir un problema	33
5.2. Reglas básicas para definir funciones	34
5.2.1. Declarar la función	34

5.2.2.	Definir una función	34
5.2.3.	Llamar a una función	35
5.3.	Paso de parámetros por valor	37
5.4.	Paso de parámetros por referencia	38
6.	Estructuras repetitivas	41
6.1.	Estructura repetitiva “durante” (for)	41
6.2.	Estructura repetitiva “mientras” (while)	47
6.3.	Estructura repetitiva “haz - mientras” (do-while)	49
7.	Tipos de datos estructurados	55
7.1.	Arreglos unidimensionales	55
7.2.	Arreglos unidimensionales y funciones	58
7.3.	Arreglos bidimensionales	65
7.4.	Cadenas	69
8.	Registros	75
8.1.	Definición de variables de tipo registro.	75
8.2.	Definición de un registro empleando alias.	76
8.3.	Representación de los registros en la memoria	79
9.	Archivos	83
9.1.	Conceptos básicos	83
9.2.	Funciones usuales para el manejo de archivos	83
10.	Introducción al sistema operativo UNIX	99
10.1.	Acceder al sistema operativo	99
10.2.	Comandos básicos	101
10.2.1.	Mostrar el directorio actual de trabajo – pwd	102
10.2.2.	Listar el contenido de un directorio - ls	102
10.2.3.	Conocer quién está conectado al servidor - who	105
10.2.4.	Cambiar de directorio – cd	105
10.2.5.	Crear un nuevo directorio - mkdir	107
10.2.6.	Eliminar un archivo - rm	108
10.2.7.	Renombrar o mover archivos – mv	109
10.2.8.	Copiar archivos - cp	110
10.2.9.	Mostrar el manual de comandos de UNIX – man	112

Capítulo 1

Arquitectura de computadoras

1.1. Modelo actual de la Computadora

Una computadora está formada por: 1) Hardware es la parte física o tangible: Unidad central de procesamiento (CPU-Central Process Unit), monitor, disco duro (HD-Hard Disk), etc. 2) Software es la parte no tangible de la computadora: el software de base y los programas de aplicación.

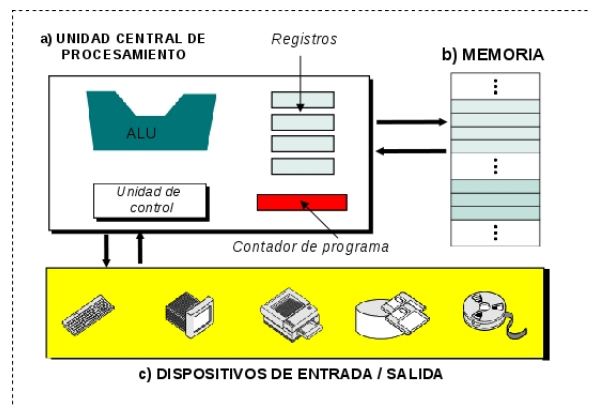


Figura 1.1 *Modelo actual de la computadora*

La arquitectura de hardware (ver figura 1.1) de las computadoras que empleamos hoy en día se atribuye a John von Neumann y otros investigadores. Los principales componentes de la computadora son: la unidad central de procesamiento, la memoria y los dispositivos de entrada y salida de datos.

1.1.1. Unidad central de procesamiento

La unidad central de procesamiento (CPU - Central Process Unit) se encarga de realizar las operaciones aritmético-lógicas con los datos, la forman: la unidad aritmético Lógica, la unidad de control y un conjunto de registros.

Unidad aritmético-lógica

La unidad aritmético-lógica (ALU - Arithmetic Logic Unit) está encargada de realizar las operaciones aritméticas y lógicas. Dentro de las operaciones aritméticas, distinguimos entre las unarias: incremento y decremento; y las binarias: $+$, $-$, $*$ y $/$. La forma de expresar las operaciones binarias es: $A < operador > B = C$. Las operaciones lógicas son: *NOT* (unaria), *AND*, *OR*, *XOR*. Con excepción de la operación de negación, el resto son binarias. Ejemplo: $p = \text{"llueve afuera"}$; $q = \text{"hay sol"}$ (ver tabla 1.1)

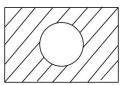
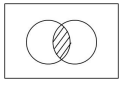
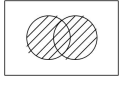
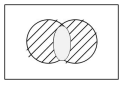
Diagrama	Operación lógica	Resultado
	NOT p	No llueve afuera
	p AND q	Llueve afuera y hay sol
	p OR q	Llueve afuera o hay sol
	p XOR q	Llueve afuera ó hay sol (uno de los dos)

Tabla 1.1 Operaciones lógicas

Unidad de control

La unidad de control (CU - Control Unit) es responsable de seleccionar las operaciones aritmético-lógicas. Esto se logra a partir de varias líneas de control que pueden estar activas o inactivas. Por ejemplo para una ALU simple con 10 operaciones diferentes se necesitan cuatro líneas de control, las cuales pueden definir 16 (2^4) situaciones diferentes, diez de las cuales pueden usarse para las operaciones

aritmético-lógicas y el resto para otros propósitos. La tabla 1.2 muestra algunos ejemplos sobre el código binario de algunas operaciones.

Código	Operación
0000	Neutro (no hay operación por hacer)
0001	Suma
0010	Resta
0011	Multipliación

Tabla 1.2 *Códigos de algunas operaciones*

Registros

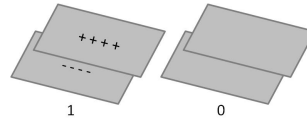
Los registros son celdas de almacenamiento, propias de la CPU, que guardan temporalmente tanto los datos de entrada que serán procesados por la ALU como el resultado del procesamiento. Distinguimos tres tipos: registros de entrada, registros de instrucción y el contador de programa.

- Registros de entrada. En el pasado las computadoras solo tenían un registro de entrada para alojar, por turnos, uno de los datos de entrada (el otro dato de entrada venía directamente de la memoria). Actualmente las computadoras utilizan docenas de registros para acelerar las operaciones, las cuales son cada vez más complejas y requieren de varios registros para mantener los resultados intermedios.
- Registros de instrucción. El CPU es el responsable de buscar las instrucciones en la memoria, una a una; luego debe almacenarlas en el registro de instrucción, interpretarlas y ejecutarlas.
- Contador de programa. Hace un seguimiento de la instrucción que se ejecuta actualmente; concluida la ejecución, el contador se incrementa para apuntar a la dirección (de memoria) de la siguiente instrucción o dato.

1.1.2. Memoria

Es una colección de celdas de almacenamiento. Cada celda tiene un identificador único conocido como “dirección“. Los datos se transfieren hacia y desde la memoria en grupos de *bits* llamados palabras.

Un *bit* es la unidad mínima de almacenamiento en una computadora; éste físicamente es un condensador (ver figura 1.2): si está cargado, entonces representa 1 (verdadero); si no está cargado, entonces representa 0 (falso).

**Figura 1.2** Condensador

Una palabra puede estar formada desde un patrón de 8 *bits* (ej. 10100100) hasta 64 *bits* en computadoras recientes. La figura 1.3 representa una memoria con 8 celdas; donde cada celda está formada por un patrón de 8 *bits*. Cada una tiene asignada una dirección expresada, de igual manera, en binario. Las direcciones están designadas por un número consecutivo que inicia con la dirección 0.

Contenido								Dirección	
1	0	1	1	0	0	1	0	↓	000
									001
									010
									011
									100
									101
									110
									111

Figura 1.3 Memoria con 8 celdas

La cantidad de celdas que contienen las memorias de las máquinas es variable: de cientos (un horno de microondas) a miles de millones (computadoras actuales). El tamaño la memoria principal suele medirse en las siguientes unidades: Kilobytes, Megabytes, Gigabytes y Terabytes; estas unidades corresponden a los valores de la tabla 1.3:

Potencia	Bytes	Equivalencia	Unidad
2^{10}	1,024	1,024 bytes	1 kilobyte (Kb)
2^{20}	1,048,576	1,024 Kb	1 Megabyte (Mb)
2^{30}	1,073,741,824	1,024 Mb	1 Gigaabyte (Gb)
2^{40}	$1,099511628 \times 10^{12}$	1,024 Gb	1 Terabyte (Tb)

Tabla 1.3 Unidades de medida para el tamaño de la memoria de una computadora

Por ejemplo una computadora cuya memoria tiene 4MB y el tamaño de palabra es igual a 16 *bits* tiene $4 * 1,048,576 = 4,192,304$ celdas, cada una con un tamaño

de 2 *byte*. Todas con su dirección correspondiente: 0, 1, 2, ..., 4, 192, 303 (pero expresado en binario). Existen varios tipos de memoria, de los cuales distinguiremos: 1) la memoria de acceso aleatorio y 2) la memoria de solo lectura.

- Memoria de acceso aleatorio (RAM - Random Access Memory). Es la memoria principal de una computadora. Un usuario puede leer y escribir en la RAM mientras la máquina esté encendida. La información es automáticamente borrada si deja de alimentarse de energía eléctrica a la computadora. Por esta característica, a la RAM también se le llama memoria volátil.
- Memoria de solo lectura (ROM - Read Only Memory). El contenido de esta memoria es grabada por el fabricante. De la información contenida en esta memoria podemos citar: datos del fabricante, modelo de la computadora, tipo de procesador, capacidad de almacenamiento, la fecha, etc. También contiene un programa llamado sistema básico de entrada y salida (BIOS - Basic Input-Output System) el cual “arranca o inicia” a la computadora.

1.1.3. Dispositivos de entrada y salida de datos

- Dispositivos de Entrada. Permiten ingresar datos a la computadora (ej. del usuario hacia la computadora)
- Dispositivos de Salida. Permiten recibir datos de la computadora (ej. de la computadora hacia el usuario)
- Dispositivos de almacenamiento secundario. Se clasifican como dispositivos de entrada y salida, a la vez. Salida: pueden almacenar grandes volúmenes de información. Entrada: esta información puede recuperarse en un momento posterior. Ejemplo: disco duro ¹, Memorias USB ², Discos compactos CD ³, disco óptico de almacenamiento de datos DVD ⁴, Memorias SD ⁵, entre otros.

1.2. Software de base

Al encender la computadora se ejecuta un programa llamado sistema básico de entrada y salida (BIOS - Basic Input-Output System), el cual está guardado en

¹HD - Hard Disc

²USB - Universal Serial Bus

³CD - Compact Disc

⁴DVD - Digital Versatile Disc

⁵Secure Digital

la ROM. El BIOS se encarga de revisar a todos los dispositivos conectados a la computadora y se encarga de buscar al sistema operativo en el disco duro, para luego cargarlo a la RAM.

El software de base esta formado por un conjunto de programas que nos permite comunicarnos con el hardware de la computadora. La figura 1.4 muestra como el software de base está entre el Hardware y los programas de aplicación.

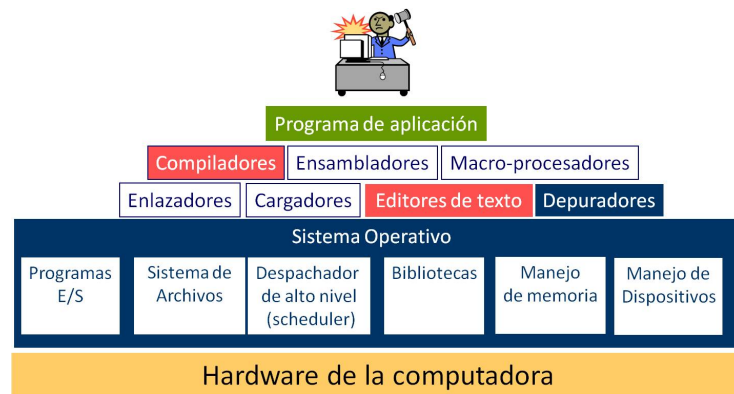


Figura 1.4 *Software de base*

En nuestro curso, construiremos distintos programas de aplicación; donde cada uno será escrito en un lenguaje de alto nivel (lenguaje C); luego, será traducido por un compilador a lenguaje máquina (lenguaje binario 0-apagado y 1-encendido); después, nuestro programa en lenguaje de máquina será ligado con otros programas propios del lenguaje C; y finalmente, este conjunto de programas será llevado a memoria principal (RAM) para que se inicie su ejecución, ver figura 1.5.

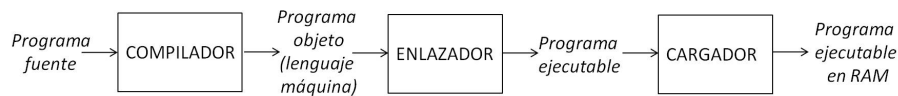


Figura 1.5 *Proceso para obtener un programa ejecutable*

- **Compilador:** Es un programa (o conjunto de programas) encargado de traducir de un programa fuente, escrito en lenguaje de alto nivel, a un programa objeto, escrito en código máquina (lenguaje binario). Un compilador realiza las siguientes operaciones: análisis léxico (lexical analysis), preprocesamiento (preprocessing), análisis sintáctico (parsing), análisis semántico (semantic analysis), generación de código (code generation) y optimización de código (code optimization).

- Un enlazador es un programa que toma los archivos: 1) del código objeto generados en el proceso de compilación y 2) el código objeto de las funciones empleadas por el programador en su programa de aplicación (las cuales forman parte de una biblioteca proporcionada por el lenguaje de alto nivel); finalmente, enlaza ambos tipos de códigos objeto produciendo un programa ejecutable.
- Un cargador es el responsable de llevar el contenido del archivo ejecutable (programa ejecutable) a la memoria principal (RAM) para, a solicitud del usuario, iniciar su ejecución.

Capítulo 2

Algoritmos

2.1. Algoritmo

Un algoritmo es un conjunto de pasos precisos y ordenados, que nos permiten alcanzar un resultado, o bien resolver un problema. Las características de un algoritmo son: Precisión, finitud y determinismo.

- Precisión. Los pasos a seguir en el algoritmo deben evitar, en la medida de lo posible, la ambigüedad.
- Finitud. Independientemente de la complejidad (dificultad) del algoritmo, el conjunto de pasos debe ser finito.
- Determinismo. Para un mismo conjunto de datos entrada, el algoritmo siempre debe arrojar el o los mismos resultados.

Las etapas principales de un algoritmo son: 1) la recepción de los datos de entrada, 2) el procesamiento de los datos y 3) la impresión de los resultados (ver figura 2.1). Ejemplo de un algoritmo para darse de alta en la aplicación “aula virtual”:

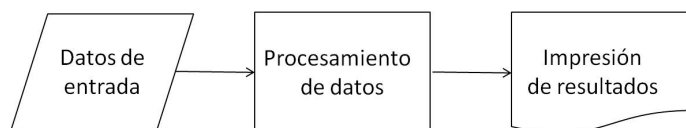


Figura 2.1 *Etapas principales de un algoritmo*

1. Acceder a la dirección de aula virtual

2. Acceder a la rúbrica "Ciencias Básicas e Ingeniería"
3. Seleccionar la UEA "Programación Estructurada" del profesor
4. Leer instrucciones para el registro
5. Realizar el registro
 - a) Escribir matrícula
 - b) Escribir fecha de nacimiento
 - c) Escribir datos generales
6. Acceder a aula virtual
7. Escribir matrícula y contraseña
8. Estás dentro!

Las etapas para resolver un problema proponiendo un algoritmo son: 1) análisis del problema, 2) construcción de un algoritmo y 3) verificación del algoritmo. El **análisis** del problema nos permite determinar su naturaleza; identificar motivos o causas de un problema específico; y, dividir el problema en subproblemas. Luego, hacemos un proceso de **síntesis** donde elaboramos escenarios, diseñamos una solución para cada subproblema, y luego las integramos para así resolver el problema inicial; este proceso también incluye mejorar esta solución. La **verificación** del algoritmo consiste en ejecutar (o realizar) cada paso señalado en el algoritmo, con datos de entrada para los cuales conocemos los resultados.

2.2. Programación

Programar es concebir un algoritmo que permita a una computadora resolver un problema. El algoritmo debe escribirse en un lenguaje claro, estricto y universal, a saber: pseudocódigo. Posteriormente, ese algoritmo expresado en pseudocódigo podrá ser traducido a cualquier lenguaje de programación, en nuestro caso a lenguaje C o lenguaje Basic.

- La **codificación** consiste en traducir el algoritmo a un lenguaje de programación, para así obtener un programa fuente.
- Un **programa**, concepto desarrollado por von Neumann en 1946, es un conjunto de instrucciones que sigue una computadora para alcanzar un resultado específico. El programa se escribe en un lenguaje de programación, a partir de un diagrama de flujo o de un algoritmo expresado en pseudocódigo.

- Un **lenguaje de programación** está constituido por un conjunto de reglas sintácticas y semánticas. Las reglas sintácticas especifican la correcta formación de palabras y oraciones en un lenguaje. Las reglas semánticas especifican el sentido o significado de un símbolo, una palabra o una oración.

2.3. Programación estructurada

La programación estructurada es una filosofía para la implementación de algoritmos a través de un conjunto finito de estructuras debidamente organizadas. Fué propuesto, en 1965, por Edgser Dijkstra, en la Universidad de Hainover. El Teorema de la estructura, propuesto por Bohn y Jacopini señala que son necesarias tres estructuras básicas para construir cualquier programa.

1. Una estructura de secuencia, donde las instrucciones se ejecutan una tras otra.
2. Una estructura de selección o decisión, donde un bloque de instrucciones se ejecuta sólo si se cumple una condición.
3. Una estructura repetitiva, donde un bloque de instrucciones se ejecuta varias veces en función de una condición.

La tabla 2.1 muestra gráficamente cada una de las estructuras básicas para construir un programa, a saber: estructura de secuencia, estructura de decisión y estructura de repetición. Estas estructuras tienen una sintaxis específica en el lenguaje de programación C; la definición y la sintaxis de cada una de ellas, las abordaremos en las secciones ??.

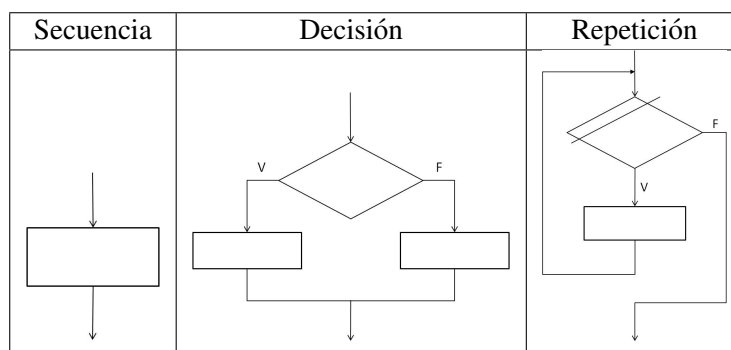


Tabla 2.1 Tres estructuras básicas para construir un programa

2.4. Diagramas de flujo

Un diagrama de flujo es la representación gráfica de un algoritmo; la construcción correcta de éste es muy importante porque nos permite escribir un programa en cualquier lenguaje de programación estructurada. La tabla ?? presenta los principales símbolos para representar a un algoritmo.

Algunas reglas para construir un diagrama de flujo correcto son:

1. Todo diagrama de flujo debe tener un inicio y un fin
2. Las líneas de dirección deben ser rectas
3. Las líneas de dirección deben estar conectadas a un símbolo que exprese lectura, proceso, decisión, escritura, conexión o fin del programa
4. Un diagrama de flujo debe construirse de arriba hacia abajo y de izquierda a derecha
5. La notación utilizada en el diagrama de flujo debe ser independiente del lenguaje de programación
6. Al realizar una tarea compleja, es conveniente poner comentarios que ayuden a entenderla
7. Si la construcción de un diagrama de flujo requiere más de una hoja, debemos emplear los conectores adecuados y enumerar las páginas.
8. No es posible que a un símbolo determinado llegue más de una línea de dirección.

Capítulo 3

Lenguaje C

3.1. Lenguaje C

C es un lenguaje de programación inventado por Dennis M. Ritchie entre 1969 y 1973 en los laboratorios Bell de New Jersey. C es un lenguaje estructurado, el cual ofrece algunas ventajas: 1) control de flujo, 2) manejo de variables simples y estructuradas, 3) simplificación en al expresar operaciones y 4) un conjunto de operadores. Originalmente C fue diseñado para el sistema operativo UNIX; sin embargo, el lenguaje C no esta ligado a ningún hardware o sistema operativo particular; podemos citar el ejemplo del sistema operativo Linux, desarrollado por Linus Torvals utilizando únicamente lenguaje C.

El lenguaje C es especialmente poderoso, pero es un lenguaje fuertemente orientado para ingenieros en Computación, porque es un lenguaje llamado nivel intermedio (se pueden hacer cosas muy apegadas a la arquitectura con la que se esté trabajando). Es un lenguaje con una curva de aprendizaje un poco más inclinada, pero como se mencionó anteriormente, esto permite hacer un uso más eficiente del hardware, siempre y cuando se esté dispuesto a pagar el precio en tiempo y esfuerzo que esto implica.

3.1.1. Tipos de datos

Los tipos de datos que procesa una computadora se clasifican en: básicos y estructurados.

- Los básicos, constituyen un solo valor que puede ser: entero, real, entero largo, real de doble precisión ó caracter.
- Los estructurados que hacen referencia a un grupo de valores. Los tipos de datos estructurados se clasifican en:

- Arreglos (vectores y matrices): Todos los valores del arreglo son de un solo tipo básico.
- Cadenas: Es un vector que guarda valores de tipo caracter
- Registros: Hacen referencia a un grupo de valores, donde cada uno de ellos puede ser de un tipo básico; inclusive puede incluir tipos de datos estructurados (arreglos, cadenas o registros)

3.1.2. Variables y constantes

Identificadores. Es el nombre que identifica a una o varias celdas de memoria, las cuales contendrán datos simples o estructurados. Un identificador se forma por medio de letras, dígitos y el caracter de subrayado (_). Un identificador siempre comienza con una letra. El lenguaje, C distingue entre minúsculas y mayúsculas, Ejemplo: Aux y aux son identificadores diferentes. La longitud máxima más común de un identificador, no excede los 7 caracteres. Existe un conjunto de palabras reservadas del lenguaje C que no deben emplearse como identificadores, ver tabla 3.1.

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

Tabla 3.1 Palabras reservadas.

Variables. Empleamos un identificador para nombrar a una variable, la cual designa un conjunto de celdas, que guardarán un valor temporalmente; es decir que el valor cambiará durante el tiempo de ejecución del programa. En el lenguaje C declaramos una variable, de los tipos mencionados en la sección 3.1.1. Ejemplos:

```
int valor1;
float valor2=5.4;
//se asigna un valor inicial a la variable
//El signo de igual (=) significa, en lenguaje C,
asignacion. El valor de la derecha del signo se asigna a
la variable o constante de la izquierda.
```

Constantes. Empleamos un identificador para nombrar a una constante, la cual designa un conjunto de celdas, que guardarán un valor que no cambia durante el tiempo de ejecución del programa.

```
#define PI 3.14169
//se declara enseguida de las bibliotecas

const float PI=3.14169;
//se declara en el cuerpo de una funcion
```

3.1.3. Operadores

Distinguiremos tres tipos de operadores: 1) aritméticos, 2) relacionales y 3) lógicos. También analizaremos los operadores simplificados, los operadores de incremento-decremento y el operador coma (,).

Operadores aritméticos. Suponga que declaramos dos variables (v y x) una de tipo real y otra de tipo entera: float v ; int x ; ver operaciones en la tabla 3.2

Símbolo	Descripción	Operación	Resultado
+	Suma	$x=4.5+3;$ $v=4.5+3;$	$x=7$ $v=7.5$
-	Resta	$x=4.5 - 3;$ $v=4.5 - 3;$	$x=1$ $v=1.5$
*	Multiplicación	$x=4.5 * 3;$ $v=4.5*3;$ $v=4*3;$	$x=12$ $v=13.5$ $v=12.0$
/	División	$x= 4 / 3;$ $x=4.0 / 3.0;$ $v=4 / 3;$ $v=4.0 / 3;$ $v=(float) 4 / 3;$ $v=((float) 5+3) / 6;$	$x=1$ $x=1$ $v=1.0$ $v=1.33$ $v=1.33$ $v=1.33$
%	Módulo ¹	$x=15 \% 2;$ $v=(15 \% 2)/2;$ $v=((float)(15 \% 2))/2;$	$x=1$ $v=0.0$ $v=0.5$

Tabla 3.2 Ejemplos de operaciones aritméticas asignadas a las variables v y x

Al evaluar expresiones aritméticas debemos respetar la jerarquía de los operadores y aplicarlos de izquierda a derecha (ver tabla 3.3).

Prioridad	Operadores
Mayor	$()$, $*$, $/$, $\%$
Menor	$+$, $-$

Tabla 3.3 Jerarquía de los operadores aritméticos

Operadores aritméticos simplificados. Nos permiten realizar la misma operación utilizando una notación más compacta (ver tabla 3.4).

Operador	Operación larga	Operación simplificada
$+=$	$x=x+5$	$x+=5$
$-=$	$y=y-2$	$y-=2$
$*=$	$x=x*y$	$x*=y$
$/=$	$x=x/y$	$x/=y$
$\%=$	$x=x \% y$	$x \% =y$

Tabla 3.4 Operadores aritméticos simplificados

Operadores de incremento y decremento. Se pueden utilizar antes o después de la variable, pero los resultados son diferentes. Suponga que declaramos las variables x y y como enteras: `int x,y; x=7;` ver resultados de operar con estas variables en la tabla 3.5

Operación	Resultado
$y=x++$	$y=7$ $x=8$
$y=++x$	$y=8$ $x=8$
$y=x--$	$y=7$ $x=6$
$y=--x$	$y=6$ $x=6$

Tabla 3.5 Operadores de incremento y decremento

Expresiones lógicas o booleanas. Están formadas por números, constantes, variables y operadores lógicos y relacionales. El valor que pueden tomar estas expresiones al ser evaluadas, es 1 (verdaderas) ó 0 (falso). Las expresiones lógicas se utilizan frecuentemente en estructuras de decisión y en estructuras repetitivas.

- Operadores relacionales. Se utilizan para comparar dos operandos, los cuales pueden ser: números, caracteres, cadenas, constantes o variables, ver tabla 3.6.
- Operadores lógicos. Permiten formular condiciones compuestas a partir de varias condiciones simples, ver tabla 3.7.

Operador relacional	Descripción	Ejemplo	Resultado
==	igual a	"h=="p"	0 (F)
!=	diferente de	"hi=="p"	1 (V)
<	menor que	7 < 5	0 (F)
>	mayor que	22 > 11	1 (V)
<=	Menor o igual que	15 <= 2	0 (F)
>=	Mayor o igual que	35 >= 20	1 (V)

Tabla 3.6 Operadores relacionales


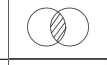

Operador lógico	Significado	Diagrama de Venn asociado
!	NOT	
&&	AND	
	OR	

Tabla 3.7 Operadores lógicos

La tabla de verdad 3.8 muestra el comportamiento de los operadores lógicos *not*, *and* y *or*

P	Q	$!P$	$!Q$	$P Q$	$P\&\&Q$
1	1	0	0	1	1
1	0	0	1	1	0
0	1	1	0	1	0
0	0	1	1	0	0

Tabla 3.8 Tabla de verdad para las sentencias P y Q

Operador “,” (coma). Sirve para encadenar expresiones. Dos ejemplos son:

```
int x, y, z, v;
x= (v=3, v*5);
```

Ejecutamos las operaciones de izquierda a derecha:

```
v = 3
x= v*5 = 3*5
x=15
```

Otro ejemplo:

```
int x, y, z, v;
x=(y=(15>10), z=(2>=y), y&&z);
```

Ejecutamos las operaciones de izquierda a derecha:

```
y=(15 > 10) entonces y = 1 (verdadero)
z=(2 >= y) entonces z = 1
x=(y && z) entonces x=1
```

Prioridades de los operadores. Las expresiones se evalúan de izquierda a derecha, pero los operadores se aplican según su prioridad, ver tabla 3.9.

Prioridad	Operadores
(+)	()
	!, ++, --
	*, /, %
	+, -
	==, !=, <, >, <=, >=
	&&,
	+=, -=, *=, /=, %=
(-)	,

Tabla 3.9 Prioridad en entre los operadores

Capítulo 4

Estructuras de decisión

Las estructuras de decisión se utilizan cuando en la solución de un problema, debemos elegir entre dos o más opciones, la cual determinará el flujo que seguirá el programa que estamos construyendo. La toma de una decisión se basa en la evaluación de una o más condiciones. Distinguimos cuatro tipos de estructuras de decisión:

1. Estructura de decisión simple (if)
2. Estructura de decisión binaria (if – else)
3. Estructura de decisión encadenada (if – else if - else)
4. Estructura de decisión anidada
5. Estructura de decisión múltiple (switch)

4.1. Decisión simple

Al llegar a una estructura de decisión, durante la ejecución de un programa, se evalúa la condición, si es verdadera entonces se ejecuta un bloque de instrucciones específico; pero si la condición es falsa entonces este bloque es ignorado y se continúa con el flujo normal del programa, ver figura 4.1.

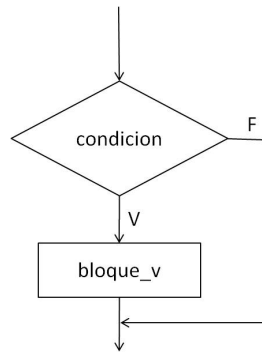


Figura 4.1 Estructura de decisión simple

```

if(condicion) {
  bloque_V;
}
:

```

- La condición puede estar formada por una o varias condiciones que al ser evaluadas nos proporcionan un valor lógico (v-verdadero ó f-falso)
- bloque_V denota un conjunto de instrucciones que se ejecuta únicamente cuando el resultado de la condición es verdadera
- Los dos puntos significan el flujo secuencial del programa

Programa 4.1: Descuento en dulces para los niños menores de 12 años En la tienda de Don Pascual, siempre ávido de tener utilidades a costa de una mayor cantidad de diabéticos infantiles, tiene una promoción que consiste en otorgar el 20% de descuento solo para los niños menores de 12 años, cuya compra sea mayor o igual a 100 pesos. Realice un programa para que de manera automática, Don Pascual no tenga que utilizar su calculadora cada que atienda a un niño con esas características.

Especificación El programa debe dar como entrada la cantidad a pagar (valor real) y un valor entero: 0 si el niño que compra es mayor o igual a 12 años, ó 1 si el niño que compra es menor de 12 años. El programa calcula el precio de la compra (valor real), con únicamente dos decimales. La tabla 4.1 muestra dos ejemplos.

Entrada	Salida
123.40 1	98.72
833.17 0	833.17

Tabla 4.1 Precio de dulces

```

#include <stdio.h>
#define DESC 0.80

int main(void) {
    float venta=0.0;
    int esMenor=0;
    printf("Cual es la cantidad de la venta: ");
    scanf("%f",&venta);
    printf("Es menor de edad (1-si 0-no): ");
    scanf("%d",&esMenor);
    if(venta>=100 && esMenor==1) {
        venta=venta*DESC;
    }
    printf("A pagar: %8.2f",venta);
    return 0;
}

```

Código 4.1: *dulces.c***NOTA:**

1. En este programa es importante que el usuario introduzca los valores con el formato correcto.
2. Recordemos que en el lenguaje C, el operador relacional “igual que” es “==”; en contraste con el operador asignación, que es “=”.
3. No olvidemos que los corchetes delimitan el inicio y fin de cualquier estructura de control, en este caso la estructura if.
4. El código de impresión esta limitado a 128 caracteres, en los programas en C no se pueden imprimir acentos ni eñes.

4.2. Decisión binaria

En la estructura de decisión binaria tenemos dos bloques alternativos de instrucciones, los cuales son independientes uno de otro; en el punto de la toma de una decisión solo se ejecutará uno de ellos, dependiendo del valor lógico que resulte al evaluar la condición, ver figura 4.2.

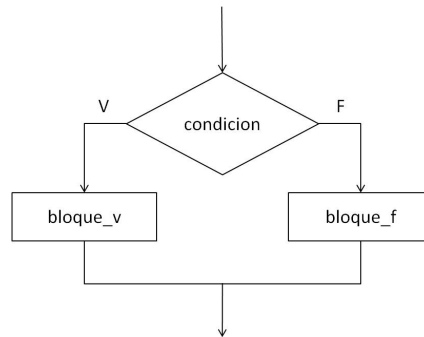


Figura 4.2 Estructura de decisión binaria

```

if (condicion) {
    bloque_V;
}
else {
    bloque_F;
}
  
```

- La condición puede estar compuesta por una o varias condiciones que al ser evaluadas nos proporcionan un valor lógico (V-verdadero ó F-falso).
- El bloque_V denota un conjunto de instrucciones que se ejecutará únicamente cuando el resultado de la condición es verdadero
- El bloque_F denota un conjunto de instrucciones que se ejecutará únicamente cuando la condición es falsa

Programa 4.2: Raíces de un polinomio de segundo grado Hacer un algoritmo que encuentre las raíces de un polinomio de segundo grado. El algoritmo debe proporcionar las raíces imaginarias, si ese fuera el caso, ver la ecuación 4.1

$$x = \frac{-b + -\sqrt{b^2 - 4ac}}{2a} \quad (4.1)$$

Especificación: Las variables de entrada son tres valores reales, y como salida pueden ser dos números reales, ó dos números imaginarios. Cuando el determinante es cero, la raíz está degenerada y se imprime dos veces. La tabla 4.2 proporciona algunos datos de entrada y salida esperados.

Entrada	Salida
1 2.5 3	-1.25 + 1.198958 i -1.25 - 1.198958 i
-1.5 2 3.5	-1 0.666667

Tabla 4.2 Ecuación cuadrática

```

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

int main(void) {
    float a, b, c, det, aux;
    printf("Dame los 3 coeficientes de la ecuaci de segundo grado:");
    printf("ax2+bx+c=0 :");
    scanf("%f %f %f", &a, &b, &c);
    det=(b*b-4*a*c);
    aux=-b/(2*a);
    if(det < 0){
        printf("x1 = %f + %f i \n",aux,pow(-det,.5)/(2*a));
        printf("x2 = %f - %f i",aux,pow(-det,.5)/(2*a));
    }
    else {
        printf("x1 = %f \nx2 = %f",aux+pow(det,.5)/(2*a),aux-pow(det,.5)/(2*
a));
    }
    return 0;
}

```

Código 4.2: *ecuacion.c*

4.3. Decisión encadenada

Cuando una condición que es falsa nos lleva a evaluar una nueva condición, decimos que las estructuras de decisión están encadenadas. Cada condición se evalúa siguiendo el orden, el bloque de instrucciones del otro caso *else* solo se ejecutará cuando ninguna de las condiciones anteriores se ha cumplido. Observe que las estructuras encadenadas mantienen independencia una de otra, por lo tanto es más sencillo manipularlas durante la solución de un problema, ver figura 4.3.

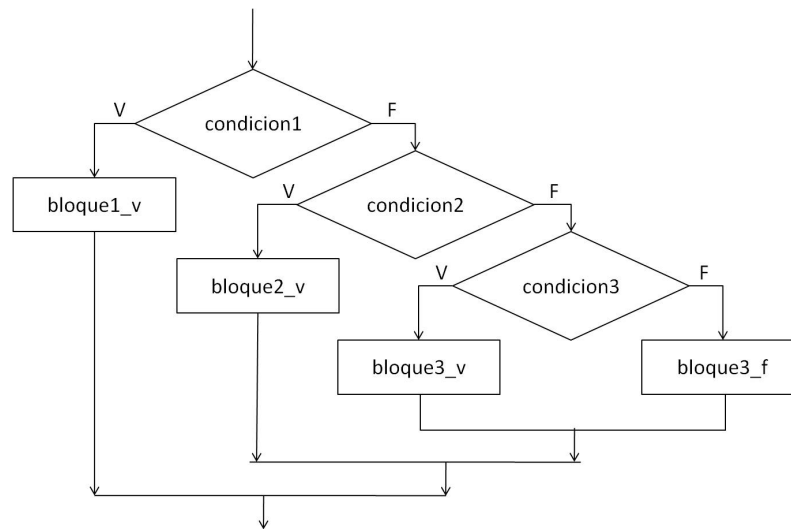


Figura 4.3 Estructura de decisión encadenada

```

if(condicion1) {
    bloque1_V;
}
else if(condicion2) {
    bloque2_V;
}
else if(condicion3) {
    bloque3_V;
}
else {
    bloque3_F;
}
  
```

- La condición puede estar compuesta por una o varias condiciones que al ser evaluadas nos proporcionan un valor lógico (V-verdadero ó F-falso).
- El bloque_V denota un conjunto de instrucciones que se ejecutará únicamente cuando el resultado de la condición es verdadero
- El bloque_F denota un conjunto de instrucciones que se ejecutará únicamente cuando ninguna de las condiciones previas se hayan cumplido

Programa 4.4: Pirinola/perinola/peonza digital. Construir un programa que simule el juego de la pirinola, también llamada perinola o peonza. Hay siete valores que se pueden asociar a un número aleatorio entre 0 y 6; o bien, en un intervalo (depende del generador de números aleatorios del lenguaje de programación).

Especificación: En este problema no hay una entrada por parte del usuario; el programa debe generar un valor aleatorio, correspondiente a los lados de la pirinola, para luego indicar la acción que los jugadores deben realizar. La tabla 4.3

determina el valor y la acción asociada.

Valor	Acción
0	Pon uno
1	Pon dos
2	Toma uno
3	Toma dos
4	Toma todo
5	Todos ponen
6	Pierdes todo

Tabla 4.3 Valores de la pirinola digital

Entrada	Salida
Sin entrada	toma todo
Sin entrada	Pon uno
Sin entrada	Todos ponen

Tabla 4.4 Pirinola

```
// Programa que simula el juego con una pirinola
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main(void) {
    double num=0.0;
    srand(time(NULL));
    printf("Pirinola/perinola/peonza digital \n");
    num=rand() %7+1;
    if(num == 1) printf("Pon uno \n");
    else if(num == 2) printf("Pon dos \n");
    else if(num == 3) printf("Toma uno \n");
    else if(num == 4) printf("Toma dos \n");
    else if(num == 5) printf("Toma todo \n");
    else if(num == 6) printf("Todos ponen \n");
    else printf("Pierdes todo");
    return 0;
}
```

Código 4.3: pirinola.c

Programa 4.5: Función definida por intervalos Construya un algoritmo que calcule el resultado de la siguiente función:

$$\begin{array}{lll}
 3y + 36 & \text{si} & 0 < y \leq 11 \\
 y^2 - 10 & \text{si} & 11 < y \leq 33 \\
 y^3 + y^2 - 1 & \text{si} & 33 < y \leq 64 \\
 0 & \text{en otro caso} &
 \end{array}$$

Especificación: Los valores de entrada y salida son flotantes.

Entrada	Salida
3	45.0
44.4	89498.75

Tabla 4.5 Función matemática

```

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

int main(void) {
    float x=0.0, y=0.0;
    printf("Introduce el valor de y: ");
    scanf("%f", &y);
    if(y>0.0 && y<=11.0) printf("x = %.2f", 3*y+36);
    else if(y>11.0 && y<=33.0) printf("x = %.2f", pow(y,2)+10);
    else if (y>33.0 && y<=64.0) printf("x=%.2f", pow(y,3)+pow(y,2)-1);
    else printf ("x=%.2f", x);
    return 0;
}

```

Código 4.4: xyFuncion.c

4.4. Decisión anidada

En términos generales, una estructura de decisión puede contener a otras estructuras, por ejemplo: una estructura de proceso, una estructura de repetición e incluso una nueva estructura de decisión. En particular, cuando una condición que es verdadera nos lleva a evaluar una nueva condición, decimos que las estructuras selectivas están anidadas. En una estructura anidada se debe tener cuidado de no olvidar cerrar las llaves correspondientes a cada una de las estructuras; y también, se debe tener cuidado de cerrarlas en el lugar adecuado (ver figura 4.4).

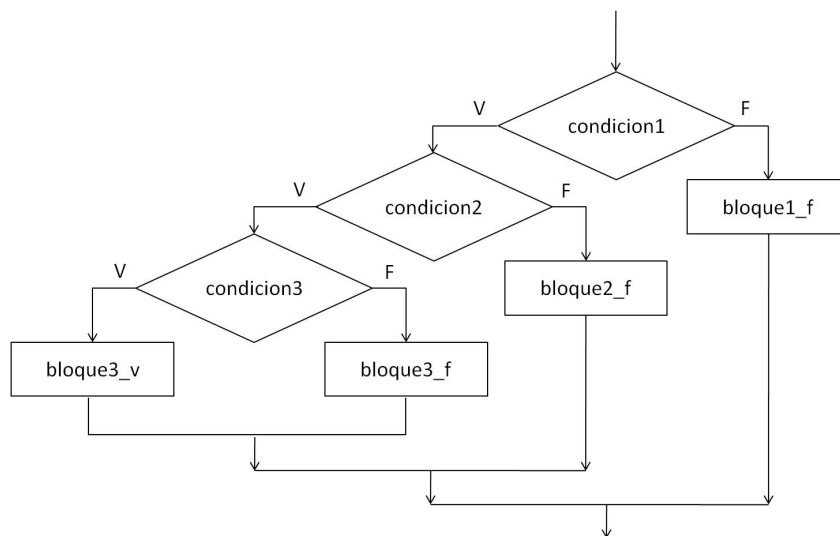


Figura 4.4 Estructura de decisión encadenada

```

if(condicion1) {
    if(condicion2) {
        if(condicion3) {
            bloque3_v;
        }
        else{
            bloque3_f;
        }
    }
    else{
        bloque2_f;
    }
}
else{
    bloque1_f;
}

```

- La condición puede estar compuesta por una o varias condiciones que al ser evaluadas nos proporcionan un valor lógico (V-verdadero ó F-falso).
- El bloque3_V solo se ejecutará cuando las condiciones 1,2 y 3 sean verdaderas
- Uno de los bloque_F se ejecutará únicamente cuando la condición respectiva no se haya cumplido

Programa 4.3: Juego de piedra-papel-tijera El programa simula ser un oponente que elige al azar una de las tres opciones: piedra, papel o tijera, y la compara con la que elige el usuario.

Especificación: Como entrada el usuario proporciona un de los siguientes números, donde cada uno tendrá un significado distinto: 1 significa piedra, 2 significa papel y 3 significa tijera. Como salida el programa deberá indicar el ganador

del juego. Es importante señalar que para simular la tirada de la computadora, en el programa deberá seleccionarse uno de los tres valores de manera aleatoria, empleando la función que genere un número aleatorio (*random* en inglés) del lenguaje de programación en que se desee codificar el algoritmo. La tabla 4.6 permite identificar los casos de gane y pierde para ambos jugadores (la computadora y el usuario). Las expresiones “Gané”, “Ganaste” y “Empate” provienen del jugador oponente simulado por la computadora. Algunos ejemplos de entrada salida se observan en la tabla 4.7

		Computadora	
Usuario	1-piedra	2-Papel	3-Tijera
1-Piedra	Empate	Gané	Ganaste
2-Papel	Gané	Empate	Gané
3-Tijera	Ganaste	Ganaste	Empate

Tabla 4.6 Opciones para piedra-papel ó tijera

Entrada	Salida
1	Tu opción fue piedra, yo escogí piedra – Empatamos!
2	Tu opción fue papel, yo escogí tijera – Gané!

Tabla 4.7 Piedra-papel-tijera

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main(void){
    int num=0, opc=0;
    srand(time(NULL));
    num=(rand() %3)+1;
    printf("1-Piedra 2-Papel 3-Tijera \n");
    printf("Escoge una de las opciones anteriores: ");
    scanf(" %d",&opc);
    if(opc==1){
        if(num==1)
            printf("Tu opcion fue Piedra, yo escogi Piedra - Empatamos!");
        else if(num==2)
            printf("Tu opcion fue Piedra, yo escogi Papel - Gane!");
        else if(num==3)
            printf("Tu opcion fue Piedra, yo escogi Tijera - Ganaste!");
    }
    else if(opc==2){
        if(num==1)
            printf("Tu opcion fue Papel, yo escogi Piedra - Ganaste!");
        else if(num==2)
            printf("Tu opcion fue Papel, yo escogi Papel - Empatamos!");
    }
}
```

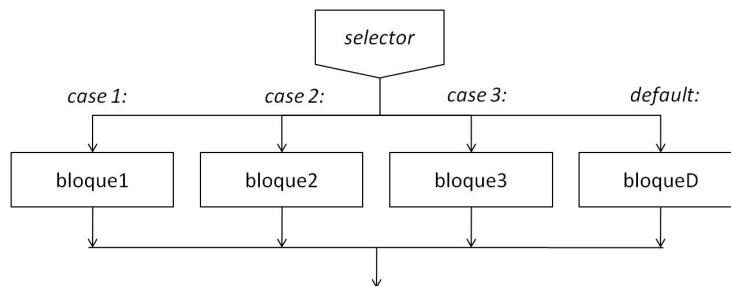


```
    else if (num==3)
        printf("Tu opcion fue Papel, yo escogi Tijera - Gane!");
    }
    else if (opc==3) {
        if (num==1)
            printf("Tu opcion fue Tijera, yo escogi Piedra - Gane!");
        else if (num==2)
            printf("Tu opcion fue Tijera, yo escogi Papel - Ganaste!");
        else if (num==3)
            printf("Tu opcion fue Tijera, yo escogi Tijera -Empatamos!");
    }
    return 0;
}
```

Código 4.5: *pipati.c*

4.5. Decisión múltiple

En problemas que involucran tomar una decisión entre múltiples opciones conviene emplear la estructura `switch` (ver figura 4.5), en lugar de emplear estructuras condicionales encadenada. La estructura de decisión múltiple proporciona un código más claro para quienes lo leen; su uso más común es la construcción de un menú de opciones para el usuario del programa.

**Figura 4.5** Estructura *switch*

```

switch(selector) {
  case 1:
    bloque1;
    break;
  case 2:
    bloque2;
    break;
  case 3:
    bloque3;
    break;
  default:
    bloqueD;
    break;
}

```

- case: denota un conjunto de instrucciones que deben ejecutarse en una opción particular.
- selector: solo puede tomar valores de tipo enteros o de tipo carácter
- opcionN: denota un número entero ó un carácter, de acuerdo al selector
- bloqueD: denota un conjunto de instrucciones que deben ejecutarse en un caso particular
- break: indica el fin de las instrucciones un caso
- default: el conjunto de instrucciones que se ejecutará cuando el valor del selector no corresponda a ninguno de los casos anteriores.

Programa 4.6: Cálculo de sueldo Dado el sueldo, la categoría y el costo por una hora extra de trabajo, calcular el sueldo total de un empleado en base a la tabla 4.8. El programa debe considerar que si el empleado trabaja más de 30 horas extras, solo se le pagarán las primeras 30.

Especificación: El usuario introduce la clave de la categoría, las horas extras y el sueldo; como resultado, el programa proporciona el sueldo neto del empleado; la tabla 4.9 muestra dos ejemplos

Categoría	Precio por hora extra
1	30
2	38
3	50
4	70

Tabla 4.8 Categoría y precio por hora extra

Entrada	Salida
2 457.9 13	951.90
4 5219.9 45	7319.90

Tabla 4.9 Sueldo de un empleado

```

#include <stdio.h>

int main(void) {
    int categoria=0,hExtra=0;
    float sueldo=0.0,PHE=0.0;
    printf("CAT \t Precio HE \n");
    printf(" 1 \t 30 \n 2 \t 38 \n 3 \t 50 \n 4 \t 70 \n");
    printf("Categoria, sueldo, hrs extra: ");
    scanf("%d %f %d", &categoria,&sueldo,&hExtra);
    if(hExtra > 30) hExtra=30;
    switch(categoria){
        case 1:
            PHE = 30;
            break;
        case 2:
            PHE = 38;
            break;
        case 3:
            PHE = 50;
            break;
        case 4:
            PHE=70;
            break;
        default:
            PHE = 0;
            break;
    }
    printf("El sueldo total es de : %.2f", sueldo+PHE*hExtra);
    return 0;
}

```

Código 4.6: *sueldoSwitch.c*

Programa 4.7: Pirinola con switch Ahora vamos a retomar el problema de la pirinola, pero esta vez emplearemos la función switch. Recordemos que para el juego tenemos siete posibilidades (ver tabla 4.10).

Valor	Acción
0	Pon uno
1	Pon dos
2	Toma uno
3	Toma dos
4	Toma todo
5	Todos ponen
6	Pierdes todo

Tabla 4.10 *Valores de la prinola digital*

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

int main(void) {
    int num=0;
    srand(time(NULL));
    printf("Pirinola/perinola/peonza digital \n\n");
    num=rand() %7;
    switch(num) {
        case 0: printf("Pon uno \n"); break;
        case 1: printf("Pon dos \n"); break;
        case 2: printf("Toma uno \n"); break;
        case 3: printf("Toma dos \n"); break;
        case 4: printf("Toma todo \n"); break;
        case 5: printf("Todos ponen \n"); break;
        default: printf("Pierdes todo"); break;
    }
    return 0;
}
```

Código 4.7: *pirinolaSwitch.c*

Capítulo 5

Funciones

Para resolver un problema de gran tamaño es conveniente descomponerlo en subproblemas, de tal manera que cada uno de ellos pueda ser resuelto de una manera más sencilla. El lenguaje C emplea el término “función” para denotar el código que resuelve un subproblema. Algunas ventajas de emplear funciones son:

- Facilita la escritura y lectura de los programas
- Permite el trabajo en paralelo
- Facilita la asignación de responsabilidades
- El código de la función se escribe una sola vez y se utiliza tantas veces como sea necesario.
- Facilita el mantenimiento de los programas.

5.1. Importancia de dividir un problema

Sea $L(p)$ la longitud de un programa cualquiera y sean $l(p_1)$ y $l(p_2)$ las longitudes de dos segmentos mutuamente excluyentes y complementarios del mismo programa p ; de tal modo que la longitud del programa será:

$$L(p) = l(p_1) + l(p_2) \quad (5.1)$$

Donde p designa al programa completo y p_i denota los subprogramas (códigos) que conforman a p . Sean además $E(p)$, $e(p_1)$ y $e(p_2)$ los esfuerzos necesarios para obtener $L(p)$, $l(p_1)$ y $l(p_2)$ respectivamente; en general se tendrá que:

$$E(p) > e(p_1) + e(p_2) \quad (5.2)$$

Es decir, el esfuerzo para construir al programa p es estrictamente mayor al esfuerzo requerido para construir de forma independiente los programas p_1 y p_2 .

5.2. Reglas básicas para definir funciones

1. Declarar la función
2. Definir de la función
3. Llamar a una función

5.2.1. Declarar la función

Es indicar al compilador que una función forma parte del programa; a esta acción también se le conoce como especificación del prototipo de una función. Generalmente, el prototipo de una función se escribe después de incluir las bibliotecas y antes de iniciar la función `main()`.

```
tipo_dato nombre_funcion(lista_parametros);
```

Donde:

- Tipo_dato indica el tipo de dato que devolverá la función al concluir su ejecución, por ejemplo: `int`, `float`, `char`
- Lista_parametros corresponde al conjunto de valores que recibirá de la función que la llamó. Los valores están contenidos en variables, por lo tanto deberán recibirse estos valores en variables locales (propias de la función receptora)
- Una función no necesariamente devuelve un valor y no necesariamente recibe parámetros, en este caso se indica con la palabra reservada `void`

5.2.2. Definir una función

Significa especificar el conjunto de instrucciones que realizarán una tarea determinada dentro del programa.

```
tipo_dato nombre_funcion(lista_parametros) {  
    declarar_variables;  
    bloque_de_instrucciones;  
    return <valor_de_regreso>;  
}
```

Donde:

- Declarar_variables corresponden a variables locales que requerirá la función para realizar su tarea
- Bloque de instrucciones que determinan el comportamiento de la función
- return marca el fin de la función y devuelve, cuando así se requiera, un resultado cuyo tipo corresponde al definido en el prototipo de la función
- Es importante que el prototipo de la función coincida exactamente con la cabecera de la función, en su definición

5.2.3. Llamar a una función

Se realiza desde la función principal ó desde cualquier otra función. Durante la ejecución del programa, el llamado a una función implica un "salto" hacia ella para ejecutar su conjunto de instrucciones; al terminar, se devuelve el control de la ejecución a la función que la llamó; en este caso a la función `main()`. Al hacer el llamado debe especificarse:

- El nombre de la función a la que se desea invocar
- Las variables cuyos valores serán enviados a la otra función. Estas variables se escriben dentro del paréntesis, siguiendo el orden y respetando el tipo de dato declarado en el prototipo.

```
int main(void) {  
    :  
    nombre_funcion(lista_de_valores);  
    :  
    return 0;  
}
```

Programa 5.1: Encontrar el mayor de tres números. Dados tres números enteros a , b y c , encontrar al mayor de entre ellos.

Especificación: La función recibe como parámetros tres valores de tipo entero, y como salida muestra al mayor de entre ellos; la tabla 5.1 muestra dos ejemplos.

Entrada	Salida
82 1 7	82
342 867 950	867

Tabla 5.1 Mayor de tres números

```
#include <stdio.h>

int leeValor(void);
int encuentraMax (int, int, int);
void imprime(int);

int main(void) {
    int a,b,c,max;
    a=leeValor();
    b=leeValor();
    c=leeValor();
    max = encuentraMax(a,b,c);
    imprime(max);
    return 0;
}

int leeValor(void) {
    int val;
    printf("Dame un valor entero: ");
    scanf("%d", &val);
    return (val);
}

int encuentraMax(int a, int b, int c) {
    int max;
    if(a>b && a>c) {
        max=a;
    }
    else if (b>a && b>c) {
        max=b;
    }
    else {
        max=c;
    }
    return (max);
}

void imprime(int max) {
    printf("El mayor de los numeros es: %d",max);
    return;
}
```

Código 5.1: *maxNum.c*

5.3. Paso de parámetros por valor

Una manera de realizar la comunicación entre las funciones es a través del paso de parámetros por valor, donde la función receptora recibe como parámetros una copia de los valores contenidos en las variables originales (propias de la función que hace el llamado); por lo tanto, si una variable local (propia de la función receptora) sufre una alteración no afectará a la variable original, porque la variable local solo tiene una copia del valor.

Programa 5.2: Piedra-Papel-Tijera con función. Retomando el juego de piedra-papel-tijera, vamos a reformular el código, empleando una función llamada “referi”, la cual estará encargada de determinar al ganador de la jugada.

Especificación: La función recibe como parámetros: el valor de la tirada del usuario y el valor de la tirada de la computadora; la función envía el resultado a la pantalla de la computadora. La tabla 5.2 muestra dos ejemplos.

Entrada	Salida
3	Tu opción fue Tijera, yo escogí Piedra – Ganaste!
2	Tu opción fue Papel, yo escogí Papel – Empatamos!

Tabla 5.2 Piedra-Papel-Tijera

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

void referi(int mine, int yours);

int main(void) {
    int mine=0, yours=0;
    printf("Juguemos a piedra papel o tijera \n");
    printf(" 1. Piedra \n 2. Papel \n 3. Tijera \n");
    printf("Cual es tu tirada: ");
    scanf("%d", &yours);
    printf("Mi tirada es: ");
    srand(time(NULL));
    mine=rand() %3+1;
    printf("%d \n",mine);
    referi(mine,yours);
    return 0;
}

void referi(int mine, int yours){
    if(mine==yours) printf("Empate entre tu y yo \n");
    else{
```

```
switch(yours){
    case 1:
        if(mine==2) printf("Gane !!!! \n");
        else printf("Ganaste!!! \n");
        break;
    case 2:
        if(mine==1) printf("Ganaste!!! \n");
        else printf("Gane!!! \n");
        break;
    case 3:
        if(mine==1) printf("Gane!!! \n");
        else printf("Ganaste!!! \n");
        break;
    default:
        printf("ERROR en tu tiro!!! \n");
        break;
}

return;
}
```

Código 5.2: *pipatiSwitch.c*

5.4. Paso de parámetros por referencia

El paso de parámetros por referencia es otra forma de comunicarnos con las funciones. Aquí los parámetros no son copias de los valores, sino las direcciones de las variables originales; por lo tanto, si un parámetro sufre una alteración en la función que lo recibe, la variable original también se ve afectada.

En el lenguaje C, el paso de parámetros por referencia se realiza mediante el concepto de "apuntador". Un apuntador es una dirección de memoria (por ejemplo 00X00X101010). Dado que debemos manipular una dirección de memoria, se vuelve necesario declarar variables que puedan guardar direcciones; las cuales no corresponden a ninguno de los tipos de datos básicos vistos hasta ahora (int, float, char, etc.). Por lo tanto, emplearemos una notación especial que nos permita denotar una variable de tipo apuntador; a saber, los operadores de: dirección (&), declaración de variable tipo apuntador (*) e indirección (*). En efecto el operador * tiene un doble propósito (ver código 5.3).

```
#include<stdio.h>

int main(void) {
    int x=0,z=0;
    int *y; //declaro a y como una variable de tipo apuntador

    x=3;
```

```

y=&x; // a y le asigno la dirección de x, por lo tanto y apunta a x
z=*y; // a z le asigno el valor de la variable a la cual apunta y

printf("z= %d ",z);
return 0;
}

```

Código 5.3: *apuntador.c*

En tiempo de ejecución, el manejo de la memoria sería como se muestra en la figura ??

Programa 5.3: Intercambio del valor de dos variables Escribir un programa que realice el intercambio de valores entre dos variables, empleando paso de parámetros por referencia.

Especificación: Este programa debe estar formado por una función que permuté los valores de dos variables *a*, *b*; esta función no devuelve ningún valor y debe recibir las direcciones de las tres variables; la tabla 5.3 muestra dos ejemplos.

Entrada	Salida
a=84 b=10	a=10 b=84
a=2 b=22	a=22 b=2

Tabla 5.3 *Intercambio de variables*

```

#include<stdio.h>

void intercambia(int *x, int *y);

int main() {
    int a=10, b=5;
    printf("\n \n Los valores de las variables originales A y B son: %d y %d", a, b);
    intercambia(&a, &b);
    printf("\n \n Los valores de A y B han sido intercambiados, ahora son: %d y %d", a, b);
    return 0;
}

void intercambia(int *x, int *y) {
    int aux;
    aux=*x;
    *x=*y;
    *y=aux;
    return;
}

```

Código 5.4: *intercambia.c*

Programa 5.4: De segundos a horas, minutos y segundos Construya un programa en C que reciba un valor entero correspondiente a una cantidad en segundos; y luego, debe mostrar en pantalla su equivalente en horas, minutos y segundos restantes.

Especificación: Escriba una función denominada “convertir” que reciba como parámetro, por valor, la cantidad de segundos; y los parámetros, por referencia *hora min* y *seg*. Las entradas y salidas son de tipo entero; la tabla 5.1 muestra dos ejemplos.

Entrada	Salida
123456a	34h 17m 36s
67758	18h 49m 18s

Tabla 5.4 Horas, minutos y segundos

```
#include <stdio.h>

void convertir (int,int*,int*,int*);

int main(void) {
    int segundos, hora, min, seg;
    printf("Ingresa los segundos: ");
    scanf("%d", &segundos);
    convertir(segundos, &hora, &min, &seg);
    printf("%d h %d min %d seg", hora, min, seg);
    return 0;
}

void convertir (int segundos, int* hora, int* min, int* seg) {
    *hora = segundos / 3600;
    *min = (segundos % 3600) / 60;
    *seg = (segundos % 3600) % 60;
    return;
}
```

Código 5.5: *horminsec.c*

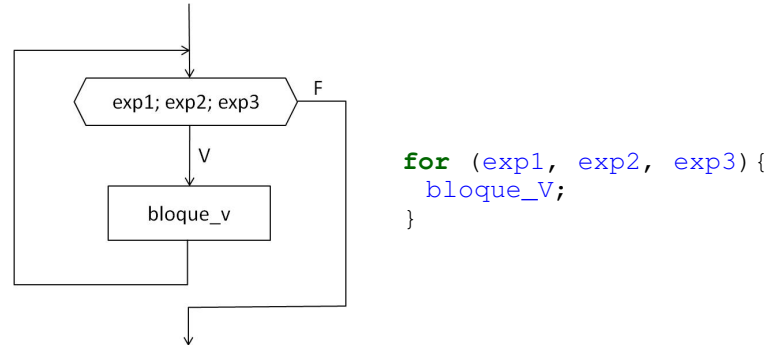
Capítulo 6

Estructuras repetitivas

En la solución de algunos problemas es común requerir que un conjunto de operaciones (instrucciones) sea ejecutado varias veces. Al conjunto de instrucciones que se ejecuta repetidamente se le llama ciclo; si bien, las instrucciones son las mismas, los datos pueden variar. En una estructura repetitiva debe existir una condición que detenga la ejecución del ciclo (suele llamarse condición de paro). En cada iteración esta condición de paro es evaluada para decidir si el ciclo continúa ó si debe detenerse.

6.1. Estructura repetitiva “durante” (for)

Generalmente, la estructura `for` (ver figura 6.1) se utiliza en soluciones donde se conoce el número de veces que deberá repetirse el ciclo. Por lo tanto, se establece un contador inicial; este contador se va incrementando ó decrementando en cada iteración, hasta alcanzar un valor máximo ó mínimo; una vez alcanzado este valor, el ciclo termina y se continúa con el flujo normal del programa.

**Figura 6.1** Estructura *for*

- **exp1**: establece un valor inicial a un contador.
- **exp2**: correspondiente a la condición de paro donde se determina el valor máximo o mínimo que podrá alcanzar el contador.
- **exp3**: corresponde al incremento ó decremento para el contador.
- **bloque_V**: conjunto de instrucciones que se repetirá hasta que la condición sea falsa. Cuando la condición es falsa.

Programa 6.1: Números perfectos Se dice que un número entero es perfecto si la suma de sus divisores excepto él mismo es igual al propio número. Por ejemplo, seis es número perfecto porque $1 * 2 * 3 = 6$ y $1 + 2 + 3 = 6$. Escriba un programa que obtenga los números perfectos comprendidos entre 1 y 10000.

Especificación: El programa no recibe ningún valor por parte del usuario; en lugar de eso, genera una lista de números en un rango de $[1., 10000]$ y para cada uno de ellos, determina si es un número perfecto. La tabla 6.1 muestra los cuatro números perfectos comprendidos en el rango antes mencionado.

Salida
6 28 496 8128

Tabla 6.1 Números perfectos

```

#include <stdio.h>
#define MAX 1e4

```

```

int main(void) {
    int num=0, sum=0, i=0;
    printf("Numeros perfectos \n\n");
    for(num=1; num<=MAX; num++) {
        sum=0;
        for(i=1; i<=(num/2); i++)
            if(num%i==0) sum = sum+i;
        if(sum==num) printf ("%5d es numero perfecto \n", num);
    }
    return 0;
}

```

Código 6.1: *4perfectos.c*

Programa 6.2: Adivina el número Construir un programa que simule el juego de adivinar un número comprendido entre 1 y 100. El usuario tiene 7 oportunidades para lograrlo. El programa debe generar aleatoriamente el número y luego solicitar al usuario que lo adivine. Si el número del usuario es menor que el número aleatorio, se le indica “mi número es mayor”; pero si el número del usuario es mayor que el número aleatorio, se le indica “mi número es menor”; si el número del usuario es igual que el número aleatorio, se le indica ”Acertaste”; si el usuario agota sus oportunidades simplemente termina el programa.

Especificación: Los números de entrada son enteros y la salida es una cadena de texto que se imprime en pantalla. La tabla 6.2 proporciona un ejemplo.

Entrada	Salida
50	Es menor
25	Es menor
12	Es mayor
17	Es mayor
21	Acertaste!
20	Es mayor
40	Es mayor
60	Es mayor
80	Es mayor
90	Es menor
88	Es menor
86	Se acabaron tus intentos!

Tabla 6.2 *Adivina un número*

```
// Programa que simula el juego de adivina el numero que estoy pensando

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

int main(void) {
    int num=0, tuNum=0, i=0;
    srand(time(NULL));
    printf("Programa adivina un nmero en 7 oportunidades o menos\n");
    printf("\n Estoy pensando un numero, entre 1 y 100");
    num=(rand() %100)+1;
    for(i=1; i<=7; i++){
        printf("\n Intenta adivinar el numero: ");
        scanf("%d", &tuNum);
        if(num < tuNum) printf("Mi numero es menor \n");
        else if(num > tuNum) printf("Mi numero es mayor \n");
        else{
            printf("Acertaste!!! \n");
            exit(1);
        }
    }
    printf("\n Lo siento, agotaste tus oportunidades");
    return 0;
}
```

Código 6.2: *adivina.c*

Programa 6.3: Números primos menores a cincuenta. Construir un programa que determine los números primos menores a cincuenta. Un número es primo, si sus únicos divisores son él mismo y la unidad

Especificación: No hay entrada por parte del usuario; como salida se en listan los números primos con la siguiente leyenda: x es primo. La tabla 6.3 muestra un ejemplo.

```
#include <stdio.h>
#define LIM 1e3

int main() {
    int num=0, i=0, band=0, cont=0;
    printf("Numeros primos comprendidos entre 2 y %.0f\n", LIM);
    for(num=2; num<=LIM; num++){
        band=0;
        for(i=2; i<=num/2; i++){
            if(num%i==0) band=1;
        }
        if(band==0) {
            printf("%d es primo \n", num);
            cont++;
        }
    }
}
```


Salida
2 es primo
3 es primo
5 es primo
7 es primo
11 es primo
13 es primo
17 es primo
19 es primo
23 es primo
29 es primo
31 es primo
37 es primo
41 es primo
43 es primo
47 es primo

Tabla 6.3 *Números primos*

```

    }
}
printf("\n%d numeros primos en el rango de 1 a %.0f\n", cont, LIM);
return 0;
}

```

Código 6.3: *primos.c*

Programa 6.4: Factorial de un entero Construir un algoritmo que calcule el factorial de un número.

$$n! = n * (n - 1) * (n - 2) * \dots * 2 * 1 \quad (6.1)$$

Especificación: Solicitar al usuario un número entero para obtener su factorial. Para la solución debe definir una función secundaria. La tabla 6.4 muestra dos ejemplos.

Entrada	Salida
0	1
10	3628800

Tabla 6.4 *Factorial de n*

```
#include <stdio.h>

int fact(int);
void serie(int);
void limite(int *n);

int main(void) {
    int n;
    limite(&n);
    serie(n);
    return 0;
}

void limite(int *n) {
    do{
        printf("introduce un numero natural: ");
        scanf("%d", n);
    }while(( *n) <= 0);
    return;
}

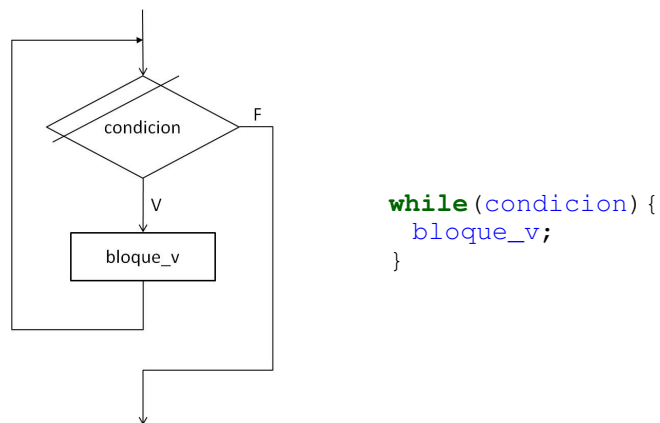
void serie(int n) {
    float termino=0.0, total=0.0;
    int i=0;
    for(i=1; i<=n; i++) {
        termino = (float) (2*i) / fact(2*i);
        printf("%f ", termino);
        total+=termino;
    }
    printf("\n %f ", total);
    return;
}

int fact(int n) {
    int aux=1, x=0;
    for(x=1; x<=n; x++) {
        aux=aux*x;
    }
    return aux;
}
```

Código 6.4: *factorial.c*

6.2. Estructura repetitiva “mientras” (while)

La estructura `while` evalúa primero una condición, si ésta resulta verdadera entonces se ejecuta el bloque de instrucciones; de lo contrario, se continúa con el flujo normal del programa. Generalmente, la estructura `while` se emplea en soluciones donde no se conoce previamente la cantidad de iteraciones que deben realizarse para obtener un resultado. Sin embargo, una estructura `while` puede emplearse en lugar de una estructura `for` en soluciones donde se conoce el número de iteraciones. Es importante señalar que una estructura `while`, podría no ejecutarse, si la condición nunca se cumple.



```
while(condicion) {  
    bloque_v;  
}
```

Figura 6.2 Estructura *while*

- **condicion:** mientras la condición sea verdadera, el ciclo continúa en ejecución. La condición de paro puede implicar varias condiciones unidas por los operadores lógicos `&&` y `||`.
- **bloque_v:** es el conjunto de instrucciones que conformarán el ciclo.

Programa 6.5: Serie alternante. Escriba un programa que empezando en el número 2, le suma cinco, al siguiente término le suma 3; y así alternadamente, hasta alcanzar el número 2500.

Especificación: El programa no tiene una entrada específica del usuario; y la salida es la serie de números que resultan de ir sumando alternadamente 5 y 3; la tabla 6.5 muestra un ejemplo.

Salida
2 7 10 15 18 ... 2500

Tabla 6.5 Serie 5 – 3

```
#include <stdio.h>

int main(void) {
    int serie=2, cambio=1;
    while(serie<=2500) {
        printf("%d\t", serie);
        if(cambio==1) {
            serie+=5;
            cambio=0;
        }
        else {
            serie+=3;
            cambio=1;
        }
    }
    return 0;
}
```

Código 6.5: *serie3-5.c*

Programa 6.5: Suma y promedio de calificaciones. Escriba un programa que solicite calificaciones a un usuario. El programa debe aceptar calificaciones hasta que se introduce un valor fuera del rango. El programa debe mostrar la suma y el promedio de las calificaciones.

Especificación: El usuario introduce las calificaciones en un rango de $[0, 10]$; y como salida se proporciona el promedio. Ambas variables deben ser declaradas como reales; la tabla 6.6 muestra dos ejemplos.

Entrada	Salida
3 5 10	6.0
5.5 8.5 9.5	7.83333

Tabla 6.6 Promedio de calificaciones

```
#include <stdio.h>

int main(void) {
    float califica=0.0, total=0.0, contador=0.0;
    printf("Calificacion: ");
```

```

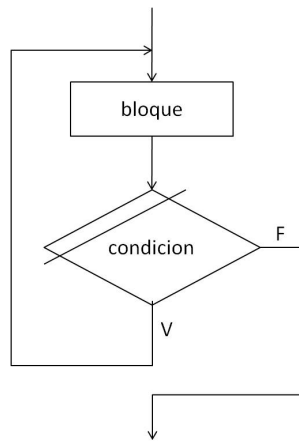
scanf("%f",&califica);
while(califica>=0 && califica<=10){
    total+=califica;
    contador++;
    printf("Calificacion: ");
    scanf("%f",&califica);
}
if(total!=0){
    printf("La suma es: %.2f \n",total);
    printf("El promedio es: %.2f \n",total/contador);
}
return 0;
}

```

Código 6.6: *calificacion.c*

6.3. Estructura repetitiva “haz - mientras” (do-while)

A diferencia de las estructuras `for` y `while`, que analizan la condición de paro al principio del ciclo, la estructura `do-while` comprueba la condición al final (ver figura 6.3). Esto significa que el bloque de instrucciones se ejecutará al menos una vez. Generalmente, esta estructura se emplea para validar datos que se reciben desde el teclado; sin embargo, puede emplearse en cualquier otro tipo de solución que involucre iteraciones.



```

do{
    bloque_V;
}while(condicion);

```

Figura 6.3 Estructura *do-while*

- `bloque_V`: es el conjunto de instrucciones que se va a repetir mientras la condición sea verdadera.

- condicion: una ó varias condiciones relacionadas con operadores lógicos.

Programa 6.6: Promedio de pares y promedio impares. Escribir un algoritmo que obtenga el promedio de los números pares ingresados por el usuario y el promedio de los impares. El programa se detiene cuando el usuario introduce cero o un valor negativo.

Especificación: El usuario introduce uno a uno valores enteros, cuando se introduce un valor menor o igual que 0, entonces se le informa por un lado sobre el promedio de los valores pares y por otro lado, sobre el promedio de los impares. Los valores de entrada son enteros y los de salida son reales. La tabla 6.7 muestra dos ejemplos.

Entrada	Salida
1 2 3 4 5 6 0	4.0 3.0
9 9 8 8 7 6	7.3333 8.33333

Tabla 6.7 Promedio de números pares e impares

```
#include <stdio.h>

int main(void) {
    int num=0;
    float tpar=0.0, timp=0.0, npar=0.0, nimp=0.0;
    do{
        printf("Introduce un entero positivo: ");
        scanf("%d", &num);
        if(num>0){
            if(num%2==0){
                tpar+=num;
                npar+=1;
            }
            else{
                timp+=num;
                nimp+=1;
            }
        }
    }while(num>0);
    if(npar==0) printf("\nNo hay promedio par");
    else printf("\nPromedio par = %f \n", tpar/npar);
    if(nimp==0) printf("\nNo hay promedio impar");
    else printf("\nPromedio impar = %f \n", timp/nimp);
    return 0;
}
```

Código 6.7: sumaPar-Impar.c

Problema 6.8: Cortina de números. Escriba un programa que genere una cortina de números en la pantalla.

Especificación: El usuario introduce un número natural para definir el tamaño de la cortina (recomendar un tamaño máximo de 12 para que se muestre la cortina correctamente en la pantalla). El programa debe validar el dato de entrada; todo valor menor o igual que 0 terminará el programa. Proponer una solución al problema empleando funciones. La tabla 6.8 muestra dos ejemplos.

Entrada	Salida
3	123321 12—21 1——1
5	1234554321 1234—4321 123——321 12————21 1—————1

Tabla 6.8 Cortinas de 3 y 5 números

```
#include <stdio.h>

int pideDato(void);
void figura(int);

int main(void) {
    int num=0;
    do{
        num=pideDato();
    }while(num<0);
    if(num>0) figura(num);
    else printf("\n Escogiste salir!!");
    return 0;
}

int pideDato(void) {
    int num=0;
    printf("\n\n Cortina de numeros. Para salir introduce un 0, o bien");
    printf("\n introduce un numero entero positivo menor a 14");
    printf("\n\n N = ");
    scanf("%d", &num);
    return (num);
}

void figura(int num) {
    int i=0, j=0, k=0, aux=num;
```

```

for(i=1; i<=num; i++){ //define numero de renglones
    for(j=1; j<=aux; j++){ //primera serie de numeros
        printf("%3d", j);
    }
    for(k=1; k<=(2*i-1); k++){ //serie de espacios
        printf(" ");
    }
    for(j=aux; j>0; j--){ // serie en orden inverso
        printf("%3d", j);
    }
    printf("\n");
    aux--; // aux determina el limite de cada rengln
}
return;
}

```

Código 6.8: *cortina.c*

Problema 6.8: Números perfectos menores o iguales a n . Escriba un programa que muestre en pantalla todos los números perfectos comprendidos en el rango de 1 a n , donde n es un número dado por el usuario.

Especificación: El programa debe incluir una función. Si $n = 0$ el programa termina. Si $n < 0$ insiste con el usuario en solicitar un valor para n , el límite de la serie. No se recomienda la búsqueda para valores más grandes a 10,000 por lo tardado que resulta encontrar el siguiente número perfecto, a saber: 33,550,336. Proponer una solución al problema empleando funciones; la tabla 6.9 muestra dos ejemplos.

Entrada	Salida
100	6 28
10000	6 28 496 8128

Tabla 6.9 Números perfectos

```

#include <stdio.h>

void numPerfecto(int);

int main(void) {
    int n=0;
    do{
        printf("Limite de la serie para encontrar numeros perfectos: ");
        scanf("%d", &n);
    }while(n<0);
    if (n!=0) {

```



```
        numPerfecto(n);
    }
    return 0;
}

void numPerfecto(int n) {
    int i=0, sum=0, j=0;
    for (i=1; i<n; i++) {
        sum=0;
        for (j=1; j<= (i/2); j++) {
            if (i%j==0) sum=sum+j;
        }
        if (sum==i) printf("%d es numero perfecto \n", i);
    }
    return;
}
```

Código 6.9: *serie-perfectos.c*

Capítulo 7

Tipos de datos estructurados

Un arreglo es una colección de variables del mismo tipo, que se referencian por un nombre común. En el lenguaje C, las celdas de memoria correspondientes a las variables de un arreglo son contiguas; por lo tanto, la dirección de memoria más baja corresponde al primer elemento del arreglo y la dirección de memoria más alta corresponde al último elemento. Dado que un arreglo está formado por celdas de memoria contigua, para acceder a un elemento es suficiente con emplear un índice, el cual denota la posición de un elemento en el arreglo.

7.1. Arreglos unidimensionales

Declarar un arreglo unidimensional es similar a declarar una variable, pero indicando su tamaño, por ejemplo:

```
int vector[10];  
//corresponde a un arreglo llamado vector de tamaño 10 y  
cuyos elementos deberán ser de tipo entero  
  
float num[40];  
//corresponde a un arreglo llamado num de tamaño 40 y cuyos  
elementos deberán ser de tipo real
```

Cuando declaramos un arreglo, es posible darle valores iniciales, por ejemplo:

```
int vector[8]={2,4,6,7,5,1,9,3};  
  
float num[3]={2.45,4.56,1.23};  
  
float calificaciones[7]={0,0};  
//todos elementos del arreglo son iguales a 0.0  
  
int datos[]={36,21,23};
```

//cuando se omite el tamaño de un arreglo, debe inicializarse con valores, y su tamaño correspondera al numero de valores de inicio

Se asigna un valor a cada celda. La asignación se hace en el orden en que aparecen los valores. En tiempo de ejecución, el arreglo vector se vería, en la memoria principal, como se muestra en la figura 7.1:

índice	0	1	2	3	4	5	6	7
vector	2.3	4.4	6.5	7.2	5.5	1.7	5.8	4.6

Tabla 7.1 Ejemplo de un arreglo unidimensional

Todos los arreglos comienzan en con una posición 0 (cero); por lo tanto, el tamaño de vector es 8, pero su última posición es 7. En general, para un arreglo de tamaño n , su última posición será $n - 1$.

Para acceder a cualquier elemento, nos referimos a él a través de su posición, por ejemplo:

En el ejemplo 7.1 para referirnos al elemento en la posición 2, escribimos `vector[2]` es 6,5; para referirnos al elemento en la posición 6, escribimos `vector[6]` es 5,8; el elemento del `vector[2 + 3]` es igual al `vector[5]`, es decir 1,7.

Problema 7.1 Imprimir en pantalla los valores de un vector. Construir un programa encargado de crear un vector con cinco elementos, el cual deberá imprimirse en la pantalla.

Especificación: Definir un arreglo de 5 elementos enteros al que llamaremos vector. El programa recibe los valores enteros desde el teclado; cuando el arreglo está completo deberá imprimirse cada elemento en la pantalla pero en el orden inverso en que fueron ingresados.

Entrada	Salida
3 4 1 9 5	5 9 1 4 3
6 28 49 68 12	12 68 49 28 6

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define MAX 5

int main(void) {
    int vector[MAX]={0}, i=0;
```

```

printf("\nProporciona 5 valores enteros para un vector\n");
for (i=0;i<MAX;i++){
    printf("\nvector[%d] = ", i);
    scanf("%d",&vector[i]);
}
printf("\nLos 5 valores del vector son, en el orden inverso son: \n\n");
;
for (i=MAX-1;i>=0;i--){
    printf("%d ", vector[i]);
}
printf("\n");
return 0;
}

```

Código 7.1: *leerEscribirVector.c*

Problema 7.2. Imprimir en pantalla los valores aleatorios de un vector Construir un programa que genere valores aleatorios que se guardarán en un arreglo unidimensional; y posteriormente, el arreglo se imprimirá en la pantalla.

Especificación: Definir un arreglo de 20 elementos al que llamaremos enteros. El programa debe generar los elementos del arreglo de forma aleatoria, en un rango de [0.,9]; el programa deberá imprimir el vector en la pantalla, la tabla 7.2 muestra dos ejemplos de arreglos generados aleatoriamente.

Salida
5 9 1 4 3 1 2 6 8 4 9 2 8 6 7 3 5 9 7 1
6 2 8 4 9 6 8 1 2 5 9 1 4 3 1 2 6 3 2 2

Tabla 7.2 *Elementos de un vector*

```

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define MAX 20

int main(void){
    int enteros[MAX]={0}, i=0;
    printf("\nGenerando valores aleatorios, en un rango de [0..9]");
    printf("\npara un vector de 20 elementos llamado enteros \n\n");
    for (i=0;i<MAX;i++){
        enteros[i]=rand()%10;
    }
    printf("\nLos 20 valores del vector enteros son: \n\n");
    for (i=0;i<MAX;i++){
        printf("%d ", enteros[i]);
    }
}

```

```
printf("\n");  
return 0;  
}
```

Código 7.2: *leerVectorAleatorio.c*

7.2. Arreglos unidimensionales y funciones

Cuando una función requiere pasar un arreglo (como parámetro) a otra función, éste pasa únicamente por referencia. Por lo tanto, cualquier modificación que se realice sobre el arreglo, en la función que lo recibe, afectará al arreglo original que se localiza en la función que realizó el llamado. La siguiente tabla muestra una comparación entre las dos notaciones que podemos emplear en el lenguaje C para manejar un arreglo que pasa como parámetro por referencia. El resultado de ambos es el mismo.

La diferencia entre los códigos 7.3 y 7.4, es que el segundo muestra el paso de parámetro de un arreglo empleando apuntadores. Sin embargo, en ambos casos, el arreglo pasa a la función *calcula* por referencia.

```
#include<stdio.h>  
  
// Prototipo  
void calcular(int datos[], int n);  
void imprimir(int datos[], int n);  
  
int main(void) {  
    int datos[]={1,2,3,4,5};  
    imprimir(datos,5);  
    calcular(datos, 5); //Llamado a la funcion  
    imprimir(datos,5);  
    return 0;  
}  
  
//Definicion de la funcion calcular  
void calcular(int datos[], int n) {  
    int i=0;  
    for(i=0; i<n; i++)  
        datos[i]= datos[i]-2;  
    return;  
}  
  
//Definicion de la funcion imprimir  
void imprimir(int datos[], int n) {  
    int i=0;  
    for(i=0; i<n; i++)  
        printf("%d ",datos[i]);  
    printf("\n\n");  
    return;  
}
```

```
}
```

Código 7.3: *vectorFuncion.c*

```
#include<stdio.h>

// Prototipo
void calcular(int* datos, int n);
void imprimir(int* datos, int n);

int main(void){
    int datos[]={5,6,7,8,9};
    imprimir(datos,5);
    calcular(datos, 5); //Llamado a la funcion
    imprimir(datos,5);
    return 0;
}

//Definicion de la funcion calcular
void calcular(int* datos, int n){
    int i=0;
    for(i=0; i<n; i++)
        *(datos+i) = *(datos+i)-2;
    return;
}

//Definicion de la funcion imprimir
void imprimir(int datos[], int n){
    int i=0;
    for(i=0; i<n; i++)
        printf("%d ", *(datos+i));
    printf("\n\n");
    return;
}
```

Código 7.4: *vectorFuncionApuntadores.c*

Problema 7.3: Suma de dos vectores enteros Construir un programa que calcule la suma de 2 arreglos unidimensionales de tipo entero. El resultado debe almacenarse en un tercer arreglo.

Especificación: El programa debe tener tres funciones: una que lea los datos, otra que haga la suma y finalmente otra que imprima los resultados. La tabla 7.3 muestra, como primer valor, el tamaño de los arreglos; luego, dos vectores dados como entrada y el vector correspondiente con la suma de los valores de ambos.

Entrada	Salida
3 1 2 3 4 6 8	5 8 11
4 3 4 7 9 9 8 4 5	12 12 11 14

Tabla 7.3 Suma de dos vectores de tipo enteros

```
#include <stdio.h>
#define MAX 100

int tamVEC(void);
void leeVEC(int X[], char nom, int n);
void sumVEC(int A[], int B[], int C[], int n);
void impVEC(int X[], char nom, int n);

int main(void) {
    int A[MAX]={0};
    int B[MAX]={0};
    int C[MAX]={0};
    int tam=0;
    tam=tamVEC();
    leeVEC(A, 'A', tam);
    leeVEC(B, 'B', tam);
    sumVEC(A, B, C, tam);
    impVEC(A, 'A', tam);
    impVEC(B, 'B', tam);
    impVEC(C, 'C', tam);
    return 0;
}

int tamVEC(void) {
    int n=0;
    do{
        printf("Dame el tama de los arreglos (1-100)");
        scanf("%d", &n);
    }while(n<=0 || n>=MAX);
    return (n);
}

void leeVEC(int X[], char nom, int n) {
    int i=0;
    printf("\nLectura del vector %c:\n", nom);
    for (i=0; i<n; i++) {
        printf("\n%c[%d]= ", nom, i);
        scanf("%d", &X[i]);
    }
    return;
}

void sumVEC(int A[], int B[], int C[], int n) {
```



```
int i=0;
for (i=0; i<n; i++){
    C[i]=A[i]+B[i];
}
return;
}

void impVEC(int X[], char nom, int n){
    int i=0;
    printf("\n Vector %c: ", nom);
    for (i=0; i<n; i++){
        printf("%d ", X[i]);
    }
    return;
}
```

Código 7.5: *sumaVectores.c*

Programa 7.4: Lanza treinta millones de dados Construir un programa que simule el lanzar 3 dados, $1E7$ veces; para luego, obtener la frecuencia con la que se presentó cada suma de caras en los lanzamientos.

Especificación: No hay datos de entrada, la salida es un entero que indique la suma y la frecuencia es un porcentaje (flotante menor que uno). La función debe tener tres funciones, una que haga la simulación de tirar los dados, otra que imprima la frecuencia. Opcionalmente, mediante valores *ascii* se puede incluir una especie de histograma de frecuencias, tal y como se muestra en el ejemplo de la tabla 7.4

Entrada	Salida
3	0.004597
4	0.013925 *
5	0.027767 **
6	0.046234 ****
7	0.069336 *****
8	0.097385 *****
9	0.115706 *****
10	0.125015 *****
11	0.125067 *****
12	0.115598 *****
13	0.097294 *****
14	0.069457 *****
15	0.046314 ****
16	0.027788 **
17	0.013925 *
18	0.004593
3	0.004658
4	0.013884 *
5	0.027817 **
6	0.046329 ****
7	0.069530 *****
8	0.097008 *****
9	0.115721 *****
10	0.124956 *****
11	0.125053 *****
12	0.115827 *****
13	0.097042 *****
14	0.069504 *****
15	0.046352 ****
16	0.027833 **
17	0.013837 *
18	0.004649

Tabla 7.4 *Histograma de frecuencias del lanzamiento de los dados*

```

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

#define NDADOS 3

```

```

#define ITERA 1.0e7

void tiraDados(int []);
void imprimeRes(int []);
void imprimeCad(int lim, char car);

int main(void) {
    int cara[100]={0};
    time_t tiempo1, tiempo2;
    srand(time(NULL));
    time(&tiempo1);
    tiraDados(cara);
    imprimeRes(cara);
    time(&tiempo2);
    printf("\nEl tiempo de ejecucion es: %f \n", (float)(tiempo2-tiempo1));
    return(0);
}

void tiraDados(int cara[]) {
    int i=0, j=0, suma=0;
    for(i=0; i<ITERA; i++) {
        suma=0;
        for(j=1; j<=NDADOS; j++) {
            suma+=(rand() %6)+1;
        }
        cara[suma]++;
    }
    return;
}

void imprimeRes(int cara[]) {
    int i=0;
    printf("Probabilidades para la suma de caras de %d dados son: \n", NDADOS);
    ;
    for(i=NDADOS; i<=NDADOS*6; i++) {
        printf("%d \t %f \t ", i, cara[i]/ITERA);
        imprimeCad((int)((100*cara[i])/ITERA), '*');
        printf("\n");
    }
    return;
}

void imprimeCad(int lim, char car) {
    int i=0;
    for(i=1; i<=lim; i++)
        printf("%c", car);
    return;
}

```

Código 7.6: *dados.c*

Problema 7.5: Convierte un entero de base 10 a cualquier otra base (base 10 incluida) Construir un programa que permita recibir dos valores: 1) un valor en-

7.3. Arreglos bidimensionales

Un arreglo bidimensional (también llamado tabla ó matriz) consta de renglones y columnas. Para acceder a un elemento debemos emplear dos índices, el primero para denotar el renglón y el segundo para denotar la columna donde se encuentra el elemento.

Declarar un arreglo bidimensional es similar a declarar un arreglo unidimensional, pero indicando, además del número de renglones, el número de columnas, por ejemplo:

```
int matriz[3][4];
//corresponde a un arreglo llamado matriz de 3 renglones y 4
//columnas; por lo tanto su tamaño es de 3x4=12 elementos,
//todos de tipo entero

float num[5][5];
//corresponde a un arreglo llamado valores de 5 renglones y
//5 columnas; por lo tanto es una matriz cuadrada de 25
//elementos de tipo real
```

Cuando declaramos un arreglo bidimensional, es posible darle valores iniciales, por ejemplo:

```
int matriz[3][4]={12,24,46,17,25,31,49,63, 42,21,89,96};
// Se asignan los valores en el orden de aparicion,
// cubriendo cada renglon

int matriz[3][4]={{12,24,46,17},{25,31,49,63},{
42,21,89,96}};
// Esta forma proporciona claridad a quien lee el codigo

int matriz[3][4]={0}; //incorrecto
// Una arreglo bidimensional no puede inicializarse con 0 (
//cero) de esta manera
```

En tiempo de ejecución, el arreglo matriz podría representarse como se muestra en la tabla 7.6:

	c_0	c_1	c_2	c_3
r_0	12	24	46	17
r_1	25	31	49	63
r_2	42	21	89	96

Tabla 7.6 Ejemplo de un arreglo bidimensional

Observe que también los arreglos bidimensionales comienzan en con una posición 0 (cero) tanto en los renglones como en las columnas; por lo tanto, el número

de renglones matriz es 3, pero su última posición es 2; mientras que el número de columnas es de 4 pero su última posición es 3, y el tamaño de la matriz es de 3×4 . En general, el tamaño de una matriz está determinado por el producto $(n_{\text{renglones}}) \times (m_{\text{columnas}})$ correspondiente al número de elementos que puede albergar el arreglo bidimensional.

Para acceder a cualquier elemento, nos referimos a él a través de su posición, por ejemplo:

En el ejemplo 7.6 para referirnos al elemento en la posición 2, 3, escribimos `matriz[2][3]` es 96; para referirnos al elemento en la posición 0, 1, escribimos `matriz[0][1]` es 24.

Problema 7.6: Traspuesta de una matriz. Construir un programa que dada una matriz, obtenga la matriz traspuesta.

Especificación: Definir dos arreglos bidimensionales (matrices) de tamaño $n \times m$, donde el tamaño máximo es de 50×50 elementos de tipo real, para cada matriz. El programa recibe, desde el teclado, el tamaño en renglones (n), el tamaño en columnas (m) y los valores de cada matriz; el programa deberá obtener su traspuesta y guardarla en una nueva matriz. Finalmente, el programa deberá imprimir en la pantalla cada una de las matrices. Para este ejercicio debemos emplear funciones. La tabla 7.7 muestra dos ejemplos.

Entrada	Salida
3 4	1 5 4
1 2 3 4	2 7 7
5 7 9 2	3 9 6
4 7 6 7	4 2 7
2 3	1.2 7.8
1.2 3.4 5.6	3.4 8.9
7.8 8.9 0.1	5.6 0.1

Tabla 7.7 Matriz traspuesta

```
#include <stdio.h>
#define MAX 50

void leeMatriz(float MAT[][MAX], int* n, int* m);
void traspuesta(float MAT1[][MAX], float MAT2[][MAX], int n, int m);
void imprMatriz(float MAT[][MAX], int n, int m);

int main(void) {
    float MAT1[MAX][MAX], MAT2[MAX][MAX];
```

```

    int n=0, m=0;
    leeMatriz (MAT1, &n, &m);
    traspuesta (MAT1, MAT2, n, m);
    imprMatriz (MAT1, n, m);
    imprMatriz (MAT2, m, n);
    return 0;
}

void leeMatriz(float MAT[][MAX], int *n, int *m){
    int i=0, j=0;
    printf("Dimension de la matriz m x n (MAXIMO 50): ");
    scanf("%d %d", n, m);
    for(i=0; i<(*n); i++){
        for (j=0; j<(*m); j++){
            printf("MAT[%d,%d]= ", i, j);
            scanf("%f", &MAT[i][j]);
        }
        printf("\n");
    }
    return;
}

void traspuesta(float MAT1[][MAX], float MAT2[][MAX], int n, int m){
    int i=0, j=0;
    for (i=0; i<n; i++){
        for (j=0; j<m; j++){
            MAT2[j][i]=MAT1[i][j];
        }
    }
    return;
}

void imprMatriz(float MAT[][MAX], int n, int m){
    int i=0, j=0;
    printf("\n");
    for (i=0; i<n; i++){
        for (j=0; j<m; j++){
            printf("%.2f ", MAT[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    return;
}

```

Código 7.8: *traspuesta.c*

Problema 7.7: ¿La matriz es simétrica? Escriba un programa que al recibir como dato un arreglo bidimensional (matriz) cuadrado, determine si el mismo es simétrico.

Especificación: Definir un arreglo bidimensional (matriz) cuadrado de tamaño máximo 50x50 elementos de tipo entero. El programa recibe, desde el teclado, el tamaño de renglones y columnas (n) y los valores de la matriz; el programa deberá determinar si esta matriz es simétrica ó no. Finalmente, el programa deberá imprimir en pantalla tanto la matriz y como la traspuesta. Para este ejercicio debemos emplear funciones. La tabla 7.8 muestra dos ejemplos.

Entrada	Salida
3 1 2 3 4 5 6 7 8 9	NO
4 3 0 4 0 0 1 2 4 4 2 5 9 0 4 9 3	SI

Tabla 7.8 *Matriz simétrica*

```
#include <stdio.h>
#define MAX 50

int leeMatriz(float[][MAX]);
int simetrica(float[][MAX],int);

int main(void) {
    float MAT[MAX][MAX];
    int n=0, resp=0;
    n=leeMatriz(MAT);
    resp=simetrica(MAT,n);
    if (resp==0) printf("\nLa matriz es simetrica\n");
    else
        printf("\nLa matriz no es simetrica\n");
    return 0;
}

int leeMatriz(float MAT[][MAX]) {
    int i=0, j=0, n=0;
    printf("Cual es el tamaño de la matriz cuadrada: ");
    scanf("%d",&n);
    for(i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            printf("MAT[%d,%d]= ",i,j);
            scanf("%f",&MAT[i][j]);
        }
        printf("\n");
    }
}
```



```

    return n;
}

int simetrica(float MAT[][MAX], int n) {
    int i=0, j=0, flag=0;
    for (i=0; i<n; i++) {
        for (j=i+1; j<n; j++) {
            if (MAT[i][j] != MAT[j][i])
                flag=1;
            j=n; i=n;
        }
    }
    return flag;
}

```

Código 7.9: *simetrica.c*

7.4. Cadenas

Una cadena es un arreglo unidimensional empleado para guardar elementos de tipo caracter *char*. Una cadena se declara igual que los arreglos, por ejemplo:

```
char cad[10];
```

//corresponde a una cadena llamada cad de tamaño 10 y cuyos elementos deberán ser de tipo caracter

En el lenguaje C, todas las cadenas terminan con un caracter *nulo* denotado por una diagonal inversa y el número cero 0; por lo tanto, al momento de definir el tamaño de la cadena tenemos que contar la celda que ocupará este caracter. El caracter nulo 0 sirve para marcar el fin de una cadena.

Cuando declaramos una cadena, es posible darle valores iniciales, por ejemplo:

```
char cadena[5]={h,o,l,a,\0};
```

//Los valores se asignan a cadena en el orden en que aparecen

```
char cad[5]=hola;
```

// otra forma valida de inicializar a cad

```
char cad[5]= ;
```

//Todos elementos del arreglo tienen como valor inicial al caracter espacio

```
char cad[]= hola;
```

//El tamaño del arreglo queda definido como 5

En tiempo de ejecución, la cadena se vería, en la memoria principal, como sigue:

índice	0	1	2	3	4
cadena	h	o	l	a	X

Tabla 7.9 Ejemplo de una cadena

Observe que si bien el tamaño de la palabra *hola* es 4, la cadena es declarada como de tamaño 5, y el caracter nulo X ocupa la última posición en el arreglo, es decir la posición 4. Al manipular una cadena se debe cuidar de no sustituir al caracter nulo X con algún otro valor.

Los operadores de asignación (=) y los relacionales (==, !=) no se emplean en el manejo de cadenas, salvo la asignación al declarar e inicializar una cadena; tal y como vimos en los ejemplos anteriores. Para manipular a las cadenas en un programa, debemos emplear un conjunto de funciones que encontramos en las bibliotecas: *string.h*, *stdlib.h* y *ctype.h*.

```
//Dos formas de ingresar caracteres
scanf ("%c",&character);

getc(character);

//Dos formas de imprimir caracteres en la pantalla
printf ("%c,character);

putc(character);

//Dos formas de ingresar cadenas:
scanf ("%s",&cadena);

gets(cadena);

//Dos formas de imprimir cadenas en la pantalla
printf ("%s,cadena);

puts(cadena);
```

Cuando se desea leer una nueva cadena, debemos emplear la función *fflush(stdin)* y *fflush(stdout)*, para limpiar el buffer de entrada y salida, donde se guarda una cadena temporalmente. La tabla 7.10 muestra una serie de comandos básicos para el manejo de cadenas en lenguaje C.

Comando	Precauciones
<i>strcpy(destino, origen)</i> . Copia el valor de cadena origen hacia la cadena destino.	No verifica que la cadena destino sea lo bastante grande como para almacenar el valor de la cadena origen.
<i>strncpy(destino, origen, limite)</i> . Igual que la función <i>strcpy</i> de dos argumentos, solo que se copian cuando mucho el límite caracteres.	Si <i>limite</i> se elige con cuidado, esta función es más segura que la versión <i>strcpy</i> .
<i>strcat(destino, origen)</i> . Concatena el valor de cadena origen con el final de la cadena destino.	No verifica que la cadena destino sea lo bastante grande como para almacenar el resultado de la concatenación.
<i>strncat(destino, origen, limite)</i> . Igual que la función <i>strcat</i> de dos argumentos, solo que se anexan cuando mucho límite caracteres.	Si <i>limite</i> se elige con cuidado, esta función es más segura que la versión <i>srtcat</i> .
<i>strlen(origen)</i> . Devuelve un entero igual a la longitud de la cadena origen; el caracter nulo X no se cuenta en la longitud.	
<i>strcmp(cadena₁, cadena₂)</i> . Devuelve 0 si <i>cadena₁</i> y <i>cadena₂</i> son iguales. Devuelve un valor menor a 0 si <i>cadena₁</i> es menor que <i>cadena₂</i> . Devuelve un valor mayor a 0 si <i>cadena₁</i> es mayor que <i>cadena₂</i> . Por lo tanto, devuelve un valor distinto de cero si <i>cadena₁</i> y <i>cadena₂</i> son distintas. Cada caracter tiene un valor entero, según la tabla de <i>ascii</i> .	Si <i>cadena₁</i> es igual a <i>cadena₂</i> esta función devuelve 0, lo que la convierte en falsa cuando se evalúa. Esto es lo inverso de lo que podríamos esperar dado que las cadenas son iguales. Tener cuidado con la lógica que sigue esta función.
<i>strncmp(cadena₁, cadena₂, limite)</i> . Igual que la función <i>strcmp</i> de dos argumentos, solo que se comparan cuando mucho límite caracteres.	Si <i>limite</i> se elige con cuidado, esta función es más segura que la función <i>strcmp</i> de dos argumentos

Tabla 7.10 Funciones para el manejo de cadenas

Programa 7.9: ¿La cadena es Palíndromo? Construir un programa que dada una cadena, determine si ésta es una cadena palíndromo. Se dice que una cadena es palíndromo si al invertir el orden de los caracteres (de derecha a izquierda) se lee igual.

Especificación: Se debe emplear una función la cual devuelve uno si la cadena es palíndromo y cero si no es palíndromo. Considere una cadena sin espacios

entre las palabras; la tabla 7.11

Entrada	Salida
AnitaLavaLatinA	Si
DammItImmaD	Si
LaminaElizaBeT	No

Tabla 7.11 ¿La cadena es palíndromo?

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

#define MAX 50

int palindrome(char cad[]);

int main(void) {
    char cad[MAX]={'\0'};
    int resp=0;
    printf("Ingresa la cadena a ser evaluada como palindrome,");
    printf("\npero SIN ESPACIOS ENTRE LAS LETRAS:\n");
    scanf("%s",cad);
    printf("la cadena mide %i y contiene %s", strlen(cad), cad);
    resp=palindrome(cad);
    if (resp==0) printf("\nLa cadena no es palindrome\n");
    else printf("\nLa cadena es palidrome\n");
    return 0;
}

int palindrome(char cad[]) {
    int i=0, j=0, flag=1;
    for (i=0, j=strlen(cad)-1; i<strlen(cad)/2 && j>=strlen(cad)/2; i++, j
--){
        printf("\n %c %c", cad[i],cad[j]);
        if(cad[i]!=cad[j]){
            flag=0;
            j=-1; i=strlen(cad)+1;
        }
    }
    return flag;
}
```

Código 7.10: *palindrome.c*

Programa 7.9: Anagramas. Construir un programa que reciba desde el teclado el nombre del usuario; el programa debe iterar n veces con el propósito de desordenar la cadena en cada iteración. El programa debe imprimir la cadena obtenida en cada iteración.

Especificación: El programa debe usar una función que reciba la cadena original y su tamaño; esta función deberá desordenar la cadena n veces; donde n es el tamaño de la cadena que ingresó el usuario. Esta misma función se encargará de imprimir la cadena resultante de cada iteración; la tabla 7.12 muestra dos ejemplos de anagramas para sus respectivas cadenas.

Entrada	Salida
William Shakespeare	ep haileSreaismWla ai pleerSWsikaalmh aihl lsmeaeWaSepirk ikaiaWrseepalheS m i am a Weakish Speller ishpeWISer ilamkaearSemeaeahs apilkWi mpilraS aaeikeelsWh Sepimli ehakWeslara salaieSWleepkhiarm eis ieklhWalrSpmaae eapWkaeiSishlalrem epmiashkerWleilaaS elmkhWe eSraasipali eapmi lelshiSekaaWr iSpmhe sklileearaWa leeea ialWrikmhpaSs imeh laelakareSWpis lieeilaSsmapWharek
lofos	sfool soofl lofso fools olfso

Tabla 7.12 *Anagramas*

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define MAX 50

void desordena(char nom[]);

int main(void) {

```

```
char nom[MAX]={'\0'};
srand(time(NULL));
printf("Escribe una cadena: ");
gets(nom);
desordena(nom);
return 0;
}

void desordena(char nom[]) {
    int i=0, j=0, esc=0;
    char aux='\0';
    for(j=1; j<=strlen(nom); j++) {
        for(i=0; i<strlen(nom); i++) {
            esc=rand()%(strlen(nom)-i)+i;
            aux=nom[i]; nom[i]=nom[esc]; nom[esc]=aux;
        }
        printf("\n");
        printf(" %s ", nom);
    }
    return;
}
```

Código 7.11: *anagramas.c*

Capítulo 8

Registros

Es una colección finita y heterogénea de elementos. A cada uno de los elementos se les denomina campo. Un campo debe ser declarado señalando su tipo de dato; sea básico: entero, real, caracter; o estructurado: arreglos, cadenas o registros. No es necesario establecer un orden entre los campos. En el lenguaje C, un registro se define como se muestra en el pseudocódigo 8.1:

```
struct registro{  
    tipo_dato campo_1;  
    tipo_dato campo_2;  
    tipo_dato campo_3;  
    tipo_dato campo_4;  
};
```

Código 8.1: Definición de un registro

Un registro define un nuevo tipo de dato; es decir, podemos crear variables que tienen la forma del registro recién definido. Todo registro debe ser definido antes de la función principal *main()* y después de incluir las bibliotecas que emplearemos en el programa.

8.1. Definición de variables de tipo registro.

Cuando declaramos variables de tipo registro, la palabra *struct* debe preceder al nombre del registro (ver pseudocódigo 8.2). Observe que una vez definido el registro, declaramos variables locales, pertenecientes a la función *main()*

```
struct registro{  
    tipo_dato campo_1;  
    tipo_dato campo_2;  
};
```

```
    tipo_dato campo_3;
    tipo_dato campo_4;
};

int main(void) {
    struct registro reg_1, reg_2;
    :
    :
    return 0;
}
```

Código 8.2: Declaración de variables locales de tipo registro

En el pseudocódigo 8.3 se declaran variables globales que siguen a la definición del registro. Sin embargo, en el curso emplearemos variables locales solamente.

```
struct registro{
    tipo_dato campo_1
    tipo_dato campo_2
    tipo_dato campo_3
    tipo_dato campo_4
} reg_1, reg_2;
```

Código 8.3: Declaración de variables globales de tipo registro

8.2. Definición de un registro empleando alias.

La instrucción *typedef* permite definir alias o sinónimos para los tipos de datos simples o estructurados. En el caso de los registros, resulta práctico porque elimina la necesidad de escribir reiteradamente la palabra *struct*. Tenemos dos alternativas para emplear un alias:

La primera alternativa, consiste en anteponer la palabra *typedef* a la palabra *struct* y colocar el nombre del registro antes del punto y coma (ver pseudocódigo 8.4).

```
typedef struct{
    tipo_dato campo_1;
    tipo_dato campo_2;
    tipo_dato campo_3;
    tipo_dato campo_4;
}registro;

int main(void) {
    registro reg_1, reg_2;
    :
```



```

:
return 0;
}

```

Código 8.4: Definición de un registro empleando un alias

La segunda alternativa, consiste en definir el registro y después del punto y coma, definir su alias, tal y como se muestra en el pseudocódigo 8.5.

```

struct registro{
    tipo_dato campo_1;
    tipo_dato campo_2;
    tipo_dato campo_3;
    tipo_dato campo_4;
};

typedef struct registro ejemplar;

int main(void){
    ejemplar reg_1, reg_2;
    :
    :
return 0;
}

```

Código 8.5: Definición de un registro empleando un alias

Para acceder a los campos de un registro empleamos dos operadores:

Operador	Símbolo	Descripción	Ejemplo
Punto	.	Lo empleamos cuando una variable es declarada del tipo registro	var_registro.campo_1
Flecha	→	Lo empleamos cuando una variable es declarada como de tipo apuntador a registro	var_aptr_registro → campo_1 (*var_aptr_registro).campo_1

Tabla 8.1 Dos formas de acceder a los campos de un registro

Programa 7.9: Registro de información personal. Construir un programa que permita guardar los datos personales de dos personas en registro llamado persona, el cual incluya: nombre, edad, peso y fecha de nacimiento. La fecha de nacimiento

de una persona deberá guardarse en un registro que incluya los campos de: año, mes y día.

Especificación: El programa debe tener dos estructuras: *fecha*, que incluye los campos *año*, *mes* y *día*, todos de tipo entero; y una estructura *persona* que incluye los campos: *nombre*, *edad*, *peso* y *nace*.

Entrada	Salida
Nombre: Susana Peso:66.50 Dia de nacimiento: 13 Mes de nacimiento: 6 Año de nacimiento: 1966	Susana tiene un peso: 66.50 kg; su fecha de nacimiento es 13/6/1966
Nombre: Daniel Peso:77.80 Dia de nacimiento: 19 Mes de nacimiento: 10 Año de nacimiento: 1974	Daniel tiene un peso: 77.80 kg; su fecha de nacimiento es 19/10/1974

Tabla 8.2 Registro de personas

```
#include<stdio.h>

typedef struct{
    int año;
    int mes;
    int día;
}fecha;

typedef struct{
    char nombre[30];
    float peso;
    fecha nace;
}persona;

void leerPersona(persona* una_persona);
void imprPersona(persona una_persona);

int main(void) {
    persona p1, p2;
    printf("Lectura de datos de dos personas \n");
    leerPersona(&p1);
    leerPersona(&p2);
    imprPersona(p1);
    imprPersona(p2);
    return 0;
}

void leerPersona(persona* una_persona) {
```

```

printf("-----");
printf("\nNombre: ");
scanf("%s", una_persona->nombre);
fflush(stdin);
printf("\nPeso: ");
scanf("%f", &una_persona->peso);
printf("\nDia de su nacimiento: ");
scanf("%d", &una_persona->nace.dia);
printf("\nMes de su nacimiento: ");
scanf("%d", &una_persona->nace.mes);
printf("\nAño de su nacimiento: ");
scanf("%d", &una_persona->nace.anio);
return;
}

void imprPersona(persona una_persona) {
printf("-----");
printf("\n%s ", una_persona.nombre);
printf("tiene un peso: %4.2f Kg;", una_persona.peso);
printf("\nsu fecha de nacimiento: %d/", una_persona.nace.dia);
printf("%d/", una_persona.nace.mes);
printf("%d", una_persona.nace.anio);
printf("\n\n");
return;
}

```

Código 8.6: *persona.c*

8.3. Representación de los registros en la memoria

En la memoria de la computadora el registro persona se vería como lo muestra la figura 8.1, donde nace es una variable de tipo fecha, la cual a su vez es un registro.

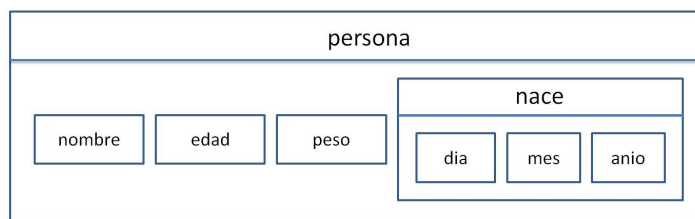


Figura 8.1 Registro persona en la memoria principal

Tal y como lo muestra el ejercicio anterior (definición del registro persona) un registro puede incluir campos de tipo: 1) arreglo (como nombre), 2) registro (como

nace); sin embargo, también es posible definir arreglos de registros, lo cual resulta útil cuando se desea manipular numerosos registros de un mismo tipo.

Observe que el paso de parámetros por valor y por referencia también se aplica en el manejo de registros. En el paso de parámetros por referencia, empleamos los operadores: (*) para declarar una variable de tipo apuntador a registro; (&) para referirnos a la dirección de una variable tipo registro; y (*) para referirnos al valor al cual apunta la variable apuntador a registro. Ver el capítulo ?? para detalles del manejo de apuntadores.

La figura 8.2 muestra las combinaciones que podemos realizar entre arreglos y registros.



Figura 8.2 *Combinaciones entre registros y arreglos*

- Registros con arreglos; es decir, al definir un registro uno más campos pueden ser de tipo arreglo unidimensional, arreglo bidimensional o cadena.
- Registros anidados; es decir, al definir un registro uno o más de sus campos pueden ser de tipo registro.
- Arreglos de registros; es decir, podemos declarar una o varias variables que son arreglos de tipo registro, tal y como lo haríamos para definir un arreglo de enteros, reales ó caracteres.

Problema 7.10: Base de datos de alumnos. Construir un programa donde se defina un arreglo unidimensional de personas, el programa debe leer los datos de las personas desde el teclado; posteriormente, deberá imprimir en pantalla el contenido de ese arreglo.

Especificación: Definir un arreglo de personas con 3 elementos. El programa deberá emplear dos funciones: una para leer el arreglo y la segunda para imprimir el contenido del arreglo en la pantalla; la tabla 8.3 muestra algunos ejemplos de los registros contenidos en un arreglo.

Entrada	Salida
Nombre: Sergio Peso:56.50 Dia de nacimiento: 23 Mes de nacimiento: 9 Anio de nacimiento: 1986	Sergio tiene un peso de 56.50 kg; la fecha de su nacimiento es 23/9/1986
Nombre: Arturo Peso:87.80 Dia de nacimiento: 14 Mes de nacimiento: 8 Anio de nacimiento: 1976	Arturo tiene un peso de 87.80 kg; la fecha de su nacimiento es 14/8/1976
Nombre: Mario Peso:98.30 Dia de nacimiento: 3 Mes de nacimiento: 6 Anio de nacimiento: 1973	Mario tiene un peso de 98.30 Kg; la fecha de su nacimiento es 3/6/1973

Tabla 8.3 Arreglo de personas

```
#include<stdio.h>
#define MAX 3

typedef struct{
    int anio;
    int mes;
    int dia;
}fecha;

typedef struct{
    char nombre[30];
    float peso;
    fecha nace;
}persona;

void leerPersona(persona* alumnos);
void imprPersona(persona* alumnos);

int main(void){
    persona alumnos[MAX];
    printf("\nLectura de datos de un grupo de alumnos \n");
    leerPersona(alumnos);
    printf("\n\nLos datos de los alumnos recién ingresados son: \n");
    imprPersona(alumnos);
    return 0;
}

void leerPersona(persona* alumnos){
    int i=0;
    for(i=0; i<MAX; i++){
        printf("-----");
        printf("\nNombre: ");
```

```
scanf("%s", alumnos[i].nombre);
fflush(stdin);
printf("\nPeso: ");
scanf("%f", &alumnos[i].peso);
printf("\nDia de nacimiento: ");
scanf("%d", &alumnos[i].nace.dia);
printf("\nMes de nacimiento: ");
scanf("%d", &alumnos[i].nace.mes);
printf("\nAño de nacimiento: ");
scanf("%d", &alumnos[i].nace.anio);
}
return;
}

void imprPersona(persona* alumnos) {
    int i=0;
    for(i=0; i<MAX; i++) {
        printf("-----");
        printf("\n%s ", alumnos[i].nombre);
        printf("tiene un peso de %4.2f Kg; ", alumnos[i].peso);
        printf("la fecha de su nacimiento es %d/", alumnos[i].nace.dia);
        printf("%d/%d", alumnos[i].nace.mes, alumnos[i].nace.anio);
        printf("\n\n");
    }
    return;
}
```

Código 8.7: *personaArreglo.c*

En el programa 8.7 hemos definido un arreglo de personas llamada alumnos. Observe también que el acceso a los datos se realiza a través del operador punto (.) porque, todos los arreglos se manejan, por defecto, mediante memoria dinámica y no es necesario expresarlo a través de la notación de apuntadores.

Capítulo 9

Archivos

Las cosas se simplifican cuando separamos el programa de los datos con los que opera, si los datos se mantienen en archivos aparte y los problemas vuelven a ser cortos y simples de leer. Tenemos varios tipos de archivos, aquellos que usaremos para consulta, otros que usaremos para poner la información de una ejecución en el mismo, y también archivos ya existentes donde se agregarán los datos recién procesados.

9.1. Conceptos básicos

Un archivo es un conjunto de datos de distintos tipos: *int*, *float*, *double*, *char*, etc., que se guardan juntos, bajo un nombre común, en un dispositivo secundario, a saber: disco duro, memoria USB, etc. Por ejemplo ‘texto.txt’ sería un archivo de datos.

Los nombres de los archivos son reconocidos por el sistema operativo a través de los comandos *ls* (unix) y *dir* (dos). Actualmente Unix y Windows (desde la versión 95) aceptan 255 caracteres para nombrar un archivo. El contenido de un archivo perdura hasta que una persona o programa lo modifica.

9.2. Funciones usuales para el manejo de archivos

Para manipular archivos, dentro de un programa escrito en lenguaje C, empleamos un conjunto de funciones, la cuales están contenidas en la biblioteca *stdio.h*; la tabla 9.1 presenta una breve descripción de estas funciones.

Función	Descripción
fopen()	Abrir archivo
fclose()	Cerrar un archivo
fputc()	Escribir un carácter en el archivo
fgetc()	Lee un carácter de un archivo
fputs()	Escribir una cadena en un archivo
fprintf()	Imprimir a un archivo
fscanf()	Leer sobre un archivo
feof()	Devuelve cierto cuando llega al fin del archivo
ferror()	Devuelve cierto si se ha producido un error
rewind()	Colocar el indicador de posición de un archivo al principio del mismo
remove()	Eliminar un archivo
fflush()	Vaciar el buffer que sirve para leer o escribir en el archivo

Tabla 9.1 *Funciones básicas de archivos*

En el lenguaje C es necesario indicar la acción que vamos a realizar sobre un archivo; a esto se le conoce como “modo de apertura de un archivo”; la tabla 9.2 describe cada uno de estos modos de apertura.

Modo	Significado	Descripción
r	read	Abrir un archivo de texto para lectura
w	write	Abrir un archivo de texto para escribir en él
a	append	Abrir un archivo para agregar texto en él
r+	read and write	Abrir un archivo para lectura y escritura

Tabla 9.2 *Modos de apertura de un archivo*

exit(valor-entero) es una función que termina de inmediato la ejecución de un programa y forma parte de la biblioteca *stdlib.h*. Se puede usar cualquier valor entero, pero por convención se emplea: 1 – para indicar que el programa terminará porque durante su ejecución se incurrió en algún error, previsto por el programador. 0 – para indicar que el programa terminará sin que esto lo cause algún error.

Programa 8.1: Escribir un conjunto de caracteres en un archivo Construir un programa que permita escribir un conjunto de caracteres en un archivo.

Especificación: Los caracteres deberán ser ingresados desde el teclado por el usuario. Al dar *enter* se guardará el conjunto de caracteres en el archivo llamado

texto.txt; posteriormente, deberá cerrarse este archivo. La tabla 9.3 muestra dos ejemplos.

Entrada del teclado	Salida al archivo <i>texto.txt</i>
King's Lead Hat = Talking Heads	King's Lead Hat = Talking Heads
Mr. Mojo Risin is Jim Morrison	Mr. Mojo Risin is Jim Morrison

Tabla 9.3 Escribir una cadena de caracteres en un archivo de texto

```
#include <stdio.h>
int main(void) {
    char car;
    FILE *ap;
    printf("Escriba una cadena para ser guardada en un archivo");
    printf("\n El fin de la cadena lo marca un <ENTER>\n");
    ap=fopen("texto.txt", "w");
    if (ap!=NULL) {
        while ((car=getchar()) != '\n') {
            fputc(car, ap);
        }
        fclose(ap);
    }
    else printf("No se puede abrir el archivo");
    return 0;
}
```

Código 9.1: *escribirArchivo.c*

Programa 8.2: Leer el contenido de un archivo Construir un programa que permita leer el contenido de un archivo. El contenido del archivo deberá desplegarse en la pantalla.

Especificación: El programa debe abrir el archivo llamado *texto.txt*; luego, debe leer carácter a carácter y desplegar cada uno en pantalla del usuario; la tabla 9.4 muestra dos ejemplos.

Archivo de entrada <i>texto.txt</i>	Salida en la pantalla
Because the wind is high it blows my mind	Because the wind is high it blows my mind
Well you should see Polythene Pam, She's so good-looking but she looks like a man.	Well you should see Polythene Pam, She's so good-looking but she looks like a man.

Tabla 9.4 Leer caracteres de un archivo de texto

```

#include <stdio.h>
int main(void) {
    char car;
    FILE *ap;
    if ((ap=fopen("prueba.txt", "r")) != NULL) {
        while (!feof(ap)) {
            car=fgetc(ap);
            putchar(car);
        }
        fclose (ap);
    }
    else printf("No se puede abrir el archivo");
    printf("\n");
    return 0;
}

```

Código 9.2: leerArchivo.c

Programa 8.3: Codificar el contenido de un archivo de texto Construir un programa que lea el contenido de un archivo de texto fuente y lo codifique. Todos los caracteres codificados deberán guardarse en un nuevo archivo.

Especificación: El criterio para codificar cada caracter del archivo consiste en sumarle 2 al caracter ascii correspondiente. El programa debe recibir desde el teclado, los nombres de los archivos fuente y destino. la tabla 9.1 muestra dos ejemplos de texto codificados.

Archivo de entrada	Archivo de salida
Because the sky is blue it makes me cry	Dgecwug"vjg"um{ "ku"dnwg"kv"ocmgu"og"et {
Because the world is round it turns me on	Dgecwug"vjg"yqtnf"ku"tqwpf"kv"vwtpu"og"qp

Tabla 9.5 Texto original y texto codificado

```

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

int main(int argc, char *argv[3]) {
    char mytext[80], encrtext[80];
    FILE *ap1, *ap2;

```

```
int i=0, desplaza=2;
if(argc!=3) {
    printf("el numero de parametros es erroneo");
    exit(1); //Termina el programa
}
else if((ap1=fopen(argv[1], "r")!=NULL && (ap2=fopen(argv[2], "w")!=
NULL) {
    while(!feof(ap1)) {
        fflush(stdin);
        i=0;
        fgets(mytext, 79, ap1);
        while (mytext[i]!='\0') {
            encrtext[i]=mytext[i]+desplaza;
            i++;
        }
        encrtext[i]='\0';
        fputs(encrtext, ap2);
    }
    fclose(ap1);
    fclose(ap2);
}
else printf("No es posible abrir el archivo");
return 0;
}
```

Código 9.3: *codificaTexto.c*

Programa 8.4: Decodificar el contenido de un archivo de texto Construir un programa que lea el contenido de un archivo de texto codificado y lo decodifique. Todos los caracteres decodificados deberán guardarse en un nuevo archivo.

Especificación: El criterio para decodificar cada caracter del archivo consiste en restarle 2 al caracter ascii correspondiente. El programa debe recibir desde el teclado, los nombres de los archivos fuente y destino. La tabla 9.6 dos ejemplos de texto decodificado.

Archivo de entrada	Archivo de salida
Ujqwnf"hkxg"rgtegpv"crrgct"vqq"uocnn. "Dg"jcpmhwn"K"fqp)v"vcmg"kv"cnn	Should five percent appear too small, Be hankful I don't take it all
[qw"cum"og"hqt"c"eqpvtkdwvkqp000"Yg) tg"fqkpi"yjc"yg"ecp	You ask me for a contribu- tion... We're doing what we can

Tabla 9.6 Texto original y texto decodificado

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

int main(int argc, char *argv[3]) {
    char mychar, yourchar;
    FILE *ap1, *ap2;
    int i=0, desplaza=-2;
    if(argc!=3){
        printf("el numero de parametros es erroneo");
        exit(1); //Termina el programa
    }
    else if((ap1 = fopen(argv[1], "r"))!=NULL && (ap2 = fopen(argv[2], "w
"))!=NULL) {
        while(!feof(ap1)) {
            fscanf(ap1, "%c", &mychar);
            yourchar=mychar+desplaza;
            fprintf(ap2, "%c", yourchar);
        }
        fprintf(ap2, "\n");
        fclose(ap1);
        fclose(ap2);
    }
    return 0;
}
```

Código 9.4: *decodificaTexto.c*

Programa 8.5: Simular el juego de bola 8 mágica Construir un programa que simule una “bola 8 mágica”, la cual adivina el futuro del usuario a través de una serie de preguntas por parte del usuario. Se trata de un juguete muy parecido a una bola de billar; la cual se agita y en la parte de atrás, se puede leer una de 20 respuestas. La bola tiene un icosaedro sumergido en tinta negra con una respuesta en cada cara).

Especificación: El programa debe contar con dos funciones aparte de la función “main”. Una de ellas debe encargarse de solicitar al usuario preguntas o la palabra fin que pone término al juego. La otra función debe proporcionar una respuesta en la pantalla. La respuesta debe ser tomada al azar de un conjunto de respuestas previamente definidas, considerando: 10 respuestas afirmativas, 5 ambiguas y 5 negativas, tal y como se muestra en el archivo 9.6. La tabla 9.7 muestra dos ejemplos.

Entrada	Salida
Pasaré el curso sin tener que estudiar?	Lo dudo mucho
En verdad adivinas el futuro?	Mejor no te digo ahora

Tabla 9.7 Respuestas de la bola 8 mágica

```
// Programa que simula una bola 8 magica

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<time.h>

void leeArchivo(char resp[][50]);
void generaPreg(char preg[]);
void generaResp(char resp[][50]);

int main(void) {
    char preg[100]={ }, resp[20][50];
    srand(time(NULL));
    printf("\n\n Este programa es magico, respondera a tus preguntas \n");
    leeArchivo(resp);
    do{
        generaPreg(preg);
        if(strcmp(preg,"fin") && strcmp(preg,"FIN"))
            generaResp(resp);
    }while(strcmp(preg,"fin") && strcmp(preg,"FIN"));
    return 0;
}

void generaPreg(char preg[]){
    fflush(stdin);
    printf("\n Haz tu pregunta o la palabra \"fin\" para terminar: \n");
    gets(preg);
    return;
}

void leeArchivo(char resp[][50]){
    int i=0;
    FILE *arch;
    arch=fopen("magica.txt", "r");
    if(arch!=NULL){
        while(!feof(arch)){
```

```
        fflush(stdin);
        fgets(resp[i], 49, arch);
        i++;
    }
}
return;
}

void generaResp(char resp[][50]){
    int seleccion=0;
    seleccion=(rand()%20)+1;
    printf("%s", resp[seleccion]);
    return;
}
```

Código 9.5: *magica8.c*

```
como lo veo... si
Es seguro
Decididamente si
Es lo mas probable
Positivamente
Parece ser que si
Sin lugar a dudas
Si
Si - definitivo
Puedes confiar en eso
Respuesta ambigua... intente de nuevo
Preguntame despues
Mejor no te digo ahora
No puedo predecirlo ahora
Concentrate y vuelve a preguntar
No cuentes con ello
Mi respuesta es no
Mis fuentes dicen que no
No parece que pase
Lo dudo mucho
```

Código 9.6: *magica8.txt*

Programa 8.6: Precios con IVA Una papelería necesita contar con un programa que le permita obtener la lista de sus precios con el Impuesto al Valor Agregado (IVA) correspondiente. El usuario deberá proporcionar el nombre del archivo donde se encuentran los precios originales en forma de lista; el programa debe calcular los nuevos precios con IVA y colocarlos en el archivo que le indique el usuario.

Especificación: El programa tiene cuatro funciones: una de ellas pregunta el nombre del archivo, otra recupera la información del archivo *A* en un arreglo, una tercera calcula el IVA y una cuarta guarda los nuevos valores con el IVA incluido en el archivo *B*. La tabla 9.8 muestra una lista con los precios originales y los precios con el IVA incluido.

Entrada	Salida
123	141.45
482.1	554.415
8.85	10.1775
9.56	10.994
4758	5471.7

Tabla 9.8 Precios originales y precios con IVA

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

#define MAX 50
#define LON 15

void nombreArchivo(char[], char[]);
int cargaDatos(float[]);
float calculaTotal(float[],int);
void guardaDatos(float[], float, int);

int main(void) {
    float precios[MAX]={0.0}, total=0.0;
    int tam=0;

    tam = cargaDatos(precios);
    total = calculaTotal(precios, tam-1);
    guardaDatos(precios, total, tam-1);
    return(0);
}

void nombreArchivo(char tipo[], char nom[]) {
    printf("Cual es el nombre del archivo de %s: \n", tipo);
    gets(nom);
    return;
}

int cargaDatos(float precios[]) {
    int n=0;
    char nomArch[LON];
    char aux1[LON]="entrada";
    FILE *ap;
    nombreArchivo(aux1,nomArch);
    ap=fopen(nomArch, "r");
    if(ap!=NULL) {
        while(!feof(ap)) {
```

```

        fscanf(ap, "%f", &precios[n]);
        n++;
    }
    fclose(ap);
    return(n);
}
else{
    printf("\n no puede abrir el archivo");
    exit(1);
}
}

float calculaTotal(float precios[],int tam){
    float total=0.0;
    int i=0;
    for(i=0; i<tam; i++){
        precios[i]*= 1.15;
        total+=precios[i];
    }
    return(total);
}

void guardaDatos(float precios[], float total, int tam){
    int i=0;
    char nomArch[LON];
    char aux1[LON]="salida";
    FILE *ap;
    nombreArchivo(aux1,nomArch);
    ap=fopen(nomArch, "a+");
    if(ap!=NULL){
        fprintf(ap, "\n");
        for(i=0; i<tam; i++){
            fprintf(ap, "\n%.2f", precios[i]);
        }
        fprintf(ap, "\nSuma total de los precios con iva: %f\n", total);
        fclose(ap);
        return;
    }
    else{
        printf("No puede abrir el archivo");
        exit(1);
    }
}
}

```

Código 9.7: *preciosTienda.c*

Programa 8.7: Guardar registros en un archivo Construir un programa que permita agregar, consultar y modificar los registros de varios alumnos, los cuales se encuentran almacenados en un archivo.

Especificación: El programa debe estar formado por: 1) una función que permita consultar los registros que están guardados en un archivo; 2) una función que permita modificar alguno de los registros; y 3) una función que permita agregar un nuevo registro. El nombre del archivo lo proporciona el usuario. La tabla 9.9 muestra el archivo *alumnos.txt* y el menú que se espera proporcione el programa al usuario; la tabla 9.10 muestra dos ejemplos para las operaciones de "agregar registro" y "consultar registro".

Nombre archivo	Menú
alumnos.txt	1. Agregar registro 2. Consultar registro 3. Modificar registro

Tabla 9.9 Archivo de entrada y menú de operaciones sobre la información del archivo

Opción	Salida
1 (Agregar registro)	Nombre: Luis Robles Pérez Peso: 45.67 Dia nacimiento:4 Mes nacimiento:6 Anio nacimiento:1967
2 (Consultar registros)	Ramiro Felix Reyes tiene un peso de 56.76 kg; su fecha de nacimiento es 7/12/1965 Luis Robles Pérez tiene un peso de 45.67 kg; su fecha de nacimiento es 4/6/1967

Tabla 9.10 Operación de agregar un registro y operación de consultar un registro

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

#define MAX 50
#define LON 15
#define TRUE 1
```

```

#define FALSE 0

typedef struct{
    int anio;
    int mes;
    int dia;
} fecha;

typedef struct{
    char nombre[LON];
    char paterno[LON];
    char materno[LON];
    float peso;
    fecha nace;
} persona;

char menu(void);
void consultar(char* nom);
void agregar(char* nom);
void modificar(char* nom);
int cargaDatos(persona* alumnos, char* nom);
void imprPersona(persona* alumnos, int tam);
void leerPersona(persona* alumnos, int i);
void guardaDatos(persona* alumnos, int tam, char* nom, int agrega);
void nombreArchivo(char* nom);

int main(void){
    char opcion=' ', nom[LON];
    nombreArchivo(nom);
    while((opcion= menu()) != 'Z' && opcion != 'z'){
        switch(opcion){
            case 'a': case 'A':
                agregar(nom);
                printf("-----Fin de la operacion agregar.....\n\n");
                break;
            case 'b': case 'B':
                consultar(nom);
                printf("-----Fin de la operacion consultar.....\n\n");
                break;
            case 'c': case 'C':
                modificar(nom);
                printf("-----Fin de la operacion modificar.....\n\n");
                break;
            default:
                printf("Opcion no valida!!!!\n"); break;
        }
    }
    return 0;
}

char menu(void){
    char opcion=' ';
    printf("\n\n Registro de alumnos\n");
    printf("\nA) Agregar registros de alumnos a un archivo.\n");
    printf("\nB) Consultar los registros de alumnos de un archivo.\n");

```

```

    printf("\nC) Modificar el registros de uno o varios alumnos.\n");
    printf("\nEscoge una opcion (A, B, C) o Z para salir\n");
    scanf("%c",&opcion);
    fflush(stdin);
    return(opcion);
}

void consultar(char* nom) {
    int tam=0;
    persona alumnos[MAX];
    tam=cargaDatos(alumnos, nom);
    printf("\nLos datos leidos son %d",tam);
    imprPersona(alumnos,tam);
    return;
}

void agregar(char* nom) {
    int tam=0, i=0;
    persona alumnos[MAX];
    printf("Cuantos registros va a agregar no mas de %d: ", MAX);
    scanf("%d", &tam);
    for(i=0; i<tam; i++) {
        leerPersona(alumnos,i);
    }
    guardaDatos(alumnos,tam,nom,TRUE);
    return;
}

void modificar(char* nom) {
    int i=0, tam=0, res=1;
    persona alumnos[MAX];
    tam=cargaDatos(alumnos, nom);
    while(res==1) {
        printf("\nQue registro desea modificar (0..%d): ", (tam-1));
        scanf("%d",&i);
        if(i>=0 && i<tam) leerPersona(alumnos,i);
        else printf("\nError... ese registro no existe");
        printf("\nDesea modificar otro registro (si-1 o no-0)? ");
        scanf("%d",&res);
    }
    guardaDatos(alumnos,tam,nom,FALSE);
    return;
}

void leerPersona(persona* alumnos, int i) {
    printf("-----");
    printf("\nNombre: ");
    scanf("%s %s %s", alumnos[i].nombre, alumnos[i].paterno,
        alumnos[i].materno);
    fflush(stdin);
    printf("\nPeso: ");
    scanf("%f", &alumnos[i].peso);
    printf("\nDia de su nacimiento: ");
    scanf("%d", &alumnos[i].nace.dia);
    printf("\nMes de su nacimiento: ");
    scanf("%d", &alumnos[i].nace.mes);
}

```

```

        printf("\nAnio de su nacimiento: ");
        scanf("%d", &alumnos[i].nace.anio);
        return;
    }

int cargaDatos(persona* alumnos, char* nom){
    int i=0;
    FILE *ap;
    ap=fopen(nom, "r");
    if(ap!=NULL){
        while(!feof(ap)){
            fscanf(ap, "%s %s %s", alumnos[i].nombre,
                alumnos[i].paterno, alumnos[i].materno);
            fflush(stdin);
            fscanf(ap, "%f", &alumnos[i].peso);
            fscanf(ap, "%d", &alumnos[i].nace.dia);
            fscanf(ap, "%d", &alumnos[i].nace.mes);
            fscanf(ap, "%d", &alumnos[i].nace.anio);
            i++;
        }
        fclose(ap);
        return(i);
    }
    else{
        printf("\n no puede abrir el archivo");
        exit(1);
    }
}

void nombreArchivo(char* nom){
    printf("Cual es el nombre del archivo: ");
    scanf("%s", nom);
    fflush(stdin);
    fflush(stdout);
    return;
}

void imprPersona(persona* alumnos, int tam){
    int i=0;
    for(i=0; i<tam; i++){
        printf("-----");
        printf("\n%s %s %s ", alumnos[i].nombre, alumnos[i].paterno,
            alumnos[i].materno);
        printf("tiene un peso: %4.2f Kg; ", alumnos[i].peso);
        printf("la fecha de su nacimiento: %d/", alumnos[i].nace.dia);
        printf("%d/%d", alumnos[i].nace.mes, alumnos[i].nace.anio);
        printf("\n\n");
    }
    return;
}

void guardaDatos(persona* alumnos, int tam, char* nom, int agrega){
    int i=0;
    FILE *ap;
    if(agrega) ap=fopen(nom, "a+");

```

```
else ap=fopen(nom, "w+");
if(ap!=NULL){
    for(i=0; i<tam; i++){
        fflush(stdin);
        fprintf(ap, "\n%s %s %s", alumnos[i].nombre,
            alumnos[i].paterno, alumnos[i].materno);
        fprintf(ap, "\n%f", alumnos[i].peso);
        fprintf(ap, "\n%d", alumnos[i].nace.dia);
        fprintf(ap, "\n%d", alumnos[i].nace.mes);
        fprintf(ap, "\n%d", alumnos[i].nace.anio);
    }
    fclose(ap);
    return;
}
else{
    printf("No puede abrir el archivo");
    exit(1);
}
}
```

Código 9.8: *personaArchivo.c*

Capítulo 10

Introducción al sistema operativo UNIX

Unix es un sistema operativo multiusuario, ésto significa que varias personas pueden ejecutar comandos de manera concurrente desde terminales interactivas o desde computadoras personales conectadas a la red. Un comando es un programa que realiza una tarea específica del sistema operativo.

Cuando el sistema operativo de UNIX está instalado en un servidor, como es el caso en la UAM-Azcapotzalco, podemos realizar una conexión remota desde nuestra computadora personal o desde una terminal interactiva.

10.1. Acceder al sistema operativo

Para realizar una conexión al servidor remoto debemos ejecutar un programa tal como el *SSH-Security Shell*, el cual debe existir previamente en nuestra computadora personal o en nuestra terminal. Luego de ejecutar el *SSH*, éste nos solicitará una clave del usuario, tal y como lo muestra el ejemplo de la tabla 10.1.

Petición	Descripción
login: annie	Primero me solicitará el usuario con el cual fui registrado en el servidor
password: x32jh88a	A continuación me solicitará la contraseña del mismo usuario

Tabla 10.1 *Conexión remota a un servidor*

La clave del usuario y la contraseña deben ser proporcionadas por el administrador del servidor. Ambos datos deben ser digitalizados exactamente como fueron proporcionados, respetando el orden y las letras mayúsculas y las minúsculas.

Cabe resaltar que, en general, un servidor proporciona tres oportunidades para introducir correctamente el usuario y la contraseña. Si ambos datos son correctos, el servidor enviará un mensaje de bienvenida seguido de un símbolo de dolar (\$), al cual se le denomina *Shell*. El *Shell* es un intérprete de comandos para el sistema operativo UNIX, a partir del cual podemos escribir comandos y ejecutarlos con solo pulsar la tecla de *enter*. Cada usuario tiene asignado un espacio de trabajo en el servidor. Siempre que se inicia la sesión de un usuario, se le sitúa en su “directorio de conexión”. La figura 10.1 presenta un árbol de directorios en UNIX, a partir del cual se propondrán ejemplos para la descripción de los comandos básicos de UNIX.

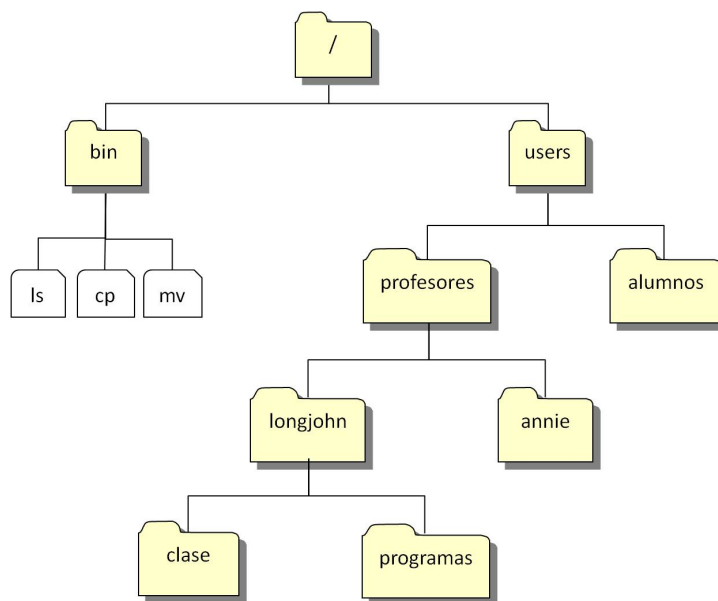


Figura 10.1 Árbol de directorios

Distinguimos dos tipos de rutas para acceder a un directorio del árbol: ruta absoluta y ruta relativa.

- Ruta absoluta o completa: describe la ruta a seguir desde la raíz del árbol hasta el archivo o directorio al que deseamos referirnos en algún comando.
- Ruta relativa: describe la ruta a seguir desde el directorio de trabajo donde nos encontramos hasta el archivo o directorio al que deseamos referirnos en

algún comando.

La tabla 10.2 muestra las rutas absoluta y relativa cuando el directorio de trabajo actual es **/users/profesores**. Con ruta **./** nos referimos al directorio actual. Con la ruta **./longjohn** nos referimos al hijo *longjohn* de *profesores*. Con la ruta **longjohn/programas** nos referimos al directorio de *programas* que es nieto del directorio de *profesores*.

Ruta relativa	Ruta absoluta o completa
./	/users/profesores
./longjohn	/users/profesores/longjohn
longjohn/programas	/users/profesores/longjohn/programas
../profesores/	/users

Tabla 10.2 Ruta relativa y absoluta

La tabla 10.3 muestra las rutas absoluta y relativa cuando el directorio de trabajo actual es **/users/profesores/longjohn/clase**. Con la ruta **../programas** nos referimos al directorio *programas* que es hermano del directorio *clase*. Con la ruta **../..** nos referimos al directorio *profesores* que es abuelo del directorio *clase*. Con la ruta **../../annie** nos referimos al directorio *annie* que es tío del directorio *clase*.

Ruta relativa	Ruta absoluta o completa
../programas	/home/longjohn/programas
../..	/home/longjohn/profesores
../../annie	/users/profesores/annie

Tabla 10.3 Ruta relativa y absoluta

10.2. Comandos básicos

Los comandos son los nombres de programas que se desean ejecutar. El formato general para ejecutar un comando es:

```
\$ comando [modificadores] [ruta | archivo]
```

Los modificadores determinan el modo como se ejecutar el comando

[x]: Indica que el elemento x es opcional

x|y: El elemento x o el elemento y

Los comandos básicos que emplearemos en el curso se describen a las siguientes secciones:

10.2.1. Mostrar el directorio actual de trabajo – pwd

Los ejemplos que se muestran en la tabla 10.4 la acción del comando *pwd* en distintas cuentas de usuario.

Comando y resultado	Descripción
\$ pwd /users/profesores/longjohn/clase	Tu directorio de trabajo actual es “clase” y la ruta de acceso es /users/profesores/longjohn
\$ pwd /users/profesores/longjohn	Tu directorio de trabajo actual es “longjohn” y la ruta de acceso es /users/profesores
\$ pwd /users/profesores	Tu directorio actual es “profesores” y la ruta de acceso es /users

Tabla 10.4 Ejemplos del comando *pwd*

10.2.2. Listar el contenido de un directorio - ls

El comando *ls* enlista el contenido de un directorio. Antes de mostrar algunos ejemplos, presentamos los atributos que caracterizan un archivo en UNIX.

-	rwxr-xr-	1	longjohn	profesores	265	Feb 13	10:47	Pipati.c
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)

Tabla 10.5 Atributos de un archivo

(a) TIPO. Los tipos de archivo más comunes se muestran en la tabla 10.6:

Tipo	Descripción
-	Archivos ordinarios, tales como los que creamos en lenguaje C
d	Directorios
l	Para indicar los enlaces simbólicos

Tabla 10.6 Tipo de archivos

- (b) **PERMISOS.** Los derechos sobre un archivo se asignan a los roles: propietario, grupo y otros (ver tabla 10.7). Los derechos posibles sobre un archivo son: lectura (r – *read*), escritura (w – *write*) y ejecución (x – *execute*)

Propietario	Grupo	Otros
$rw\bar{x}$	$r - x$	$r - -$
Todos los permisos	Permiso de lectura y ejecución	Permiso de solo lectura

Tabla 10.7 Roles y permisos

- (c) **ENLACES SIMBÓLICOS.** Indica el número de enlaces simbólicos que tiene un archivo. Un enlace simbólico corresponde a la referencia (o dirección) de un archivo.
- (d) **PROPIETARIO.** Indica quien creó el archivo.
- (e) **GRUPO.** Indica el grupo al cual pertenece el propietario en el momento de la creación del archivo.
- (f) **TAMAÑO.** expresado en *bytes* del archivo
- (g) **FECHA.** Indica la última fecha en que fué modificado el archivo
- (h) **HORA.** Indica la hora de la última modificación del archivo
- (i) **NOMBRE DEL ARCHIVO.** Indica el nombre del archivo o del directorio

En el ejemplo 1 de la tabla 10.8 el modificador $-l$ nos proporciona datos como: el tipo del archivo, los derechos, el propietario, el grupo asociado, la fecha y hora de la última modificación el nombre del archivo. Los modificadores pueden emplearse en conjunto. En el ejemplo 2 de la tabla 10.8 se solicita que muestre el contenido de los subdirectorios del directorio de trabajo actual, a saber */users/profesores/longjohn*.

Ejemplo 1							
\$ ls -l							
total 906							
-rw-r--r--	1	longjohn	profesor	666	Feb 12 2010	MCD.cpp	
-rwxr-xr-x	1	longjohn	profesor	10080	Feb 12 2010	MCD.exe	
drwxr-xr-x	2	longjohn	profesor	512	Feb 11 2009	programas	
drwxr-xr-x	2	longjohn	profesor	712	Feb 11 2009	tareas	
Ejemplo 2							
\$ ls -RI							
total 906							
-rw-r--r--	1	longjohn	profesor	649	Feb 12 2010	MCD-B.cpp	
-rwxr-xr-x	1	longjohn	profesor	10084	Feb 12 2010	MCD-B.exe	
-rw-r--r--	1	longjohn	profesor	666	Feb 12 2010	MCD.cpp	
-rwxr-xr-x	1	longjohn	profesor	10080	Feb 12 2010	MCD.exe	
./programas:							
total 16							
-rw-r--r--	1	longjohn	profesor	503	Jan 28 2011	pirinola.c	
-rwxr-xr-x	1	longjohn	profesor	6536	Jan 28 2011	pirinola.exe	
./tareas:							
total 54							
-rwxr-xr-x	1	longjohn	profesor	8340	Jun 5 2007	distancia.exe	
-rw-r--r--	1	longjohn	profesor	3254	Jun 5 2007	distancia.c	
-rwxr-xr-x	1	longjohn	profesor	8520	Jun 5 2007	laminar.exe	
-rw-r--r--	1	longjohn	profesor	4298	Jun 5 2007	laminar.c	

Tabla 10.8 *Ejemplos del comando ls*

Otros modificadores comunes se muestran en la tabla 10.9.

Modificador	Descripción
-a	Muestra todos los archivos, incluyendo aquellos que son ocultos
-C	Muestra los nombres de archivos en columnas
-c	Muestra todos los archivos ordenados por fecha de creación
-u	Muestra todos los archivos ordenados por la fecha del último acceso
-R	Muestra el contenido de los directorios
-p	Distingue a los directorios de los archivos ordinarios a través de “/”, colocado al final del nombre

Tabla 10.9 Otros modificadores para el comando *ls*

10.2.3. Conocer quién está conectado al servidor - *who*

El ejemplo 1 de la tabla 10.10 muestra la lista a los usuarios y la terminal de conexión correspondiente a cada uno de ellos. El ejemplo 2 de la misma tabla 10.10 proporciona mi usuario y la terminal desde la cual me he conectado.

Ejemplo 1				
\$ who				
ip-18-03	pts/11	Feb 13 13:13	(172.16.2.61)	
ip-16-18	pts/12	Feb 13 13:19	(172.16.20.58)	
cbrn	pts/14	Feb 13 10:28	(172.17.12.82)	
longjohn	pts/15	Feb 13 10:40	(172.17.20.128)	
ip-20-13	pts/17	Feb 13 13:28	(172.16.2.36)	
ip-21-33	pts/18	Feb 13 13:36	(172.16.20.58)	
ip-9-02	pts/19	Feb 13 12:13	(172.17.20.128)	
Ejemplo 2				
\$ who am i				
longjohn	pts/15	Feb 13 10:40	(172.17.20.128)	

Tabla 10.10 Ejemplos del comando *who*

10.2.4. Cambiar de directorio – *cd*

El comando *cd* nos permite cambiar de directorio y tiene varias posibilidades:
El ejemplo 1 de la tabla 10.11 muestra como cambiar al directorio especificado.

Observe que *clase* es un subdirectorio del directorio de trabajo *long john*; es decir, cuando estamos en un directorio específico podemos acceder a sus subdirectorios con solo escribir:

```
$cd [directorio]
```

El ejemplo 2 de la tabla 10.11 muestra como cambiar al directorio especificado a través de una ruta absoluta. Observe que nos encontramos en el directorio *clase* (*/users/profesores/longjohn/clase*) y queremos situarnos en el directorio *programas*, el cual es hermano de *clase*, en este caso podemos emplear la ruta absoluta (*/users/profesores/longjohn/programas*). La notación general es:

```
$cd [ruta_absoluta]
```

El ejemplo 3 de la tabla 10.11 muestra como cambiar al un directorio hermano evocando al padre con la ruta relativa (*../*). Observe que en este ejemplo hacemos la misma acción que el ejemplo 2; sin embargo, ahora empleamos una ruta relativa, refiriéndonos al padre del actual directorio. La notación general es

```
$cd [..|../directorio]
```

El ejemplo 4 de la tabla 10.11 muestra como cambiar al directorio raíz del sistema de archivos de UNIX

Ejemplo 1	
D. inicial	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn> \$ cd clase
D. actual	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase>
Ejemplo 2	
D. inicial	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase> \$ cd /users/profesores/longjohn/programas
D. actual	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/programas>
Ejemplo 3	
D. inicial	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase> \$ cd ../programas
D. actual	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/programas>
Ejemplo 4	
D. inicial	<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase> \$ cd /
D. actual	<longjohn@ce.azc.uam.mx> </>

Tabla 10.11 Ejemplos del comando *cd*

10.2.5. Crear un nuevo directorio - *mkdir*

El comando *mkdir* nos permite crear un nuevo directorio y tiene varias posibilidades:

El ejemplo 1 de la tabla 10.12 muestra como crear el directorio *archivos* como hijo del directorio de trabajo actual *clase*. La notación general es

```
$mkdir [directorio]
```

El ejemplo 2 de la tabla 10.12 muestra como crear el directorio *ciclos* a través de la ruta relativa (*../*). Observe que nos encontramos en el directorio *archivos* (*/users/profesores/longjohn/clase/archivos*) y queremos crear al directorio *ciclos* en el directorio *clase*. La notación general es

```
$mkdir [...]/directorio]
```

El ejemplo 3 de la tabla 10.12 muestra como crear el directorio *seleccion* en el directorio *clase* empleando la ruta absoluta */users/profesores/longjohn/clase/*. Observe que nos encontramos en el directorio */users/profesores/longjohn/*. La notación general es:

```
$mkdir [ruta_absoluta$ $nuevo_directorio]
```

Ejemplo 1		
<longjohn@ce.azc.uam.mx>		
</users/profesores/longjohn/clase>	Directorio actual <i>clase</i>	
\$ mkdir archivos	Crear directorio <i>archivos</i> en <i>clase</i>	
Ejemplo 2		
<longjohn@ce.azc.uam.mx>		
</users/profesores/longjohn/clase/archivos>	Directorio actual <i>archivos</i>	
\$ mkdir ../ciclos	Crear directorio <i>ciclos</i> en el directorio padre de <i>archivos</i> a saber: <i>clase</i>	
Ejemplo 3		
<longjohn@ce.azc.uam.mx>		
</users/profesores/longjohn/>	Directorio actual <i>longjohn</i>	
\$ mkdir /users/profesores/longjohn/clase/seleccion	Crear directorio <i>seleccion</i> en el directorio <i>clase</i>	

Tabla 10.12 Ejemplos del comando *mkdir*

10.2.6. Eliminar un archivo - rm

El comando *rm* nos permite eliminar un archivo específico y tiene varias posibilidades:

El ejemplo 1 de la tabla 10.13 muestra como eliminar el archivo *calculadora.c* del directorio de trabajo actual *seleccion*. La notación general es:

```
$rm [nombre_archivo]
```

El ejemplo 2 de la tabla 10.13 muestra como eliminar todos los archivos con extensión *txt* del directorio actual de trabajo *archivos*. El modificador *-i* solicita confirmación de la eliminación de cada archivo. La notación general es:

```
$rm -i [*.extension]
```

El ejemplo 3 de la tabla 10.13 muestra como eliminar el archivo *fuentes.txt* que se encuentra en el directorio *clase*; pero nos encontramos en el directorio

`/users/profesores/longjohn/`; por lo tanto, empleamos la ruta relativa `./clase`. La notación general es

```
$rm [ruta_absoluta|ruta_relativa]
```

Ejemplo 1	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/seleccion> \$ rm calculadora.c	Directorio actual <i>seleccion</i> Eliminar el archivo <i>calculadora</i> del directorio actual
Ejemplo 2	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/archivos> \$ rm -i *.txt	Directorio actual <i>archivos</i> Eliminar archivos con extensión <i>txt</i> del directorio actual
Ejemplo 3	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/> \$ rm ./clase/fuente.txt	Directorio actual <i>longjohn</i> Eliminar archivo <i>fuente.txt</i> del directorio <i>clase</i>

Tabla 10.13 Ejemplos del comando `rm`

10.2.7. Renombrar o mover archivos – `mv`

El comando `mv` nos permite cambiar de nombre a un archivo en el mismo directorio; además de permitir mover uno o varios archivos de un directorio a otro. Este comando tiene varias posibilidades:

El ejemplo 1 de la tabla 10.14 muestra como cambiar de nombre al archivo *oficio.txt* por *reporte.txt*, el directorio de trabajo actual es *archivos*. La notación general es:

```
$mv [nombre_inicial nombre_final]
```

El ejemplo 2 de la tabla 10.14 muestra como mover el archivo *prueba.txt* que se encuentra en *archivos* al directorio *seleccion*, pero con el nombre *datos.txt*. Si se omite el segundo nombre, se copiará con el mismo nombre. La notación general es:

```
$mv [ruta_absoluta|ruta_relativa] archivo_1 [ruta_absoluta|
ruta_relativa] archivo_2
```

El ejemplo 3 de la tabla 10.14 muestra como mover un conjunto de archivos que cumplen con el criterio *if** del directorio actual de trabajo *clase* al directorio (*seleccion* empleando la ruta relativa *./seleccion*). La notación general es:

```
$rm [ruta_absoluta|ruta_relativa] criterio_archivos [
ruta_absoluta|ruta_relativa]
```

El ejemplo 4 de la tabla 10.14 muestra como mover el directorio *multiple* al directorio *seleccion*. Observe que nos encontramos en el directorio *longjohn*; por lo tanto, empleamos la ruta relativa *./clase/*. La notación general es:

```
$rm [ruta_absoluta|ruta_relativa] directorio_fuente [
ruta_absoluta|ruta_relativa] directorio_destino
```

Ejemplo 1	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/archivos> \$ mv oficio.txt reporte.txt	Directorio actual <i>archivos</i> cambia de nombre <i>oficio.txt</i> a <i>reporte.txt</i>
Ejemplo 2	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/archivos> \$ mv prueba.txt ../seleccion/datos.txt	Directorio actual <i>archivos</i> Mueve el archivo <i>prueba.txt</i> al directorio <i>seleccion</i> con el nombre <i>datos.txt</i>
Ejemplo 3	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase> \$ mv if ./seleccion	Directorio actual <i>clase</i> Mover todos los archivos que inician con <i>if</i> al directorio <i>seleccion</i>
Ejemplo 4	
<longjohn@ce.azc.uam.mx> </users/profesores/longjohn/> \$ mv ./clase/multiple ./clase/seleccion	Directorio actual <i>longjohn</i> Mover el directorio <i>multiple</i> al directorio <i>seleccion</i>

Tabla 10.14 Ejemplos del comando *mv*

10.2.8. Copiar archivos - cp

El comando *cp* nos permite copiar el contenido de un archivo en el mismo directorio o en otros directorios y tiene varias posibilidades:

El ejemplo 1 de la tabla 10.15 muestra como copiar el contenido del archivo *reporte.txt* al archivo *reporte – ver01.txt* en el directorio de trabajo actual *archivos*. La notación general es:

```
$\textit{cp} [\textit{archivo\_original} \textit{archivo\_copia}]
```

El ejemplo 2 de la tabla 10.15 muestra como copiar el archivo *prueba.txt* que se encuentra en el directorio *seleccion* al directorio *archivos*, conservando el mismo nombre. Si se desea copiar con otro nombre se especifica al final del comando. La notación general es:

```
$cp [\textit{ruta\_absoluta}|\textit{ruta\_relativa}] \textit{archivo\_original} [\textit{ruta\_absoluta}|\textit{ruta\_relativa}] \textit{archivo\_copia}
```

El ejemplo 3 de la tabla 10.15 muestra como copiar un conjunto de archivos que cumplen con el criterio **.c* del directorio actual de trabajo *clase* al directorio (*seleccion* empleando la ruta relativa (*./seleccion*)). La notación general es:

```
$cp [\textit{ruta\_absoluta}|\textit{ruta\_relativa}] \textit{criterio\_archivos} [\textit{ruta\_absoluta}|\textit{ruta\_relativa}]
```

El ejemplo 4 de la tabla 10.15 muestra como copiar el directorio *juegos* que se encuentra en el directorio *archivos* hacia el directorio *clase*. Por lo tanto, empleamos la ruta relativa (*../*). El modificador *–R* permite que se copien también los subdirectorios. La notación general es:

```
$cp [\textit{ruta\_absoluta}|\textit{ruta\_relativa}] \textit{directorio\_original} [\textit{ruta\_absoluta}|\textit{ruta\_relativa}] \textit{directorio\_destino}
```

Ejemplo 1	
<pre><longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/archivos> \$ cp reporte.txt reporte-ver01.txt</pre>	<p>Directorio actual <i>archivos</i> copia el conteindo del archivo <i>reporte.txt</i> a <i>reporte-ver01.txt</i></p>
Ejemplo 2	
<pre><longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/seleccion> \$ cp prueba.txt ../archivos</pre>	<p>Directorio actual <i>seleccion</i> copia el archivo <i>prueba.txt</i> al directorio <i>archivos</i> con el mismo nombre</p>
Ejemplo 3	
<pre><longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase> \$ mv *.c ./seleccion</pre>	<p>Directorio actual <i>clase</i> copia todos los archivos cuya extensión es <i>.c</i> al directorio <i>seleccion</i></p>
Ejemplo 4	
<pre><longjohn@ce.azc.uam.mx> </users/profesores/longjohn/clase/archivos> \$ cp -R ../juegos ../</pre>	<p>Directorio actual <i>archivos</i> copia el contenido del directorio <i>juegos</i>, incluyendo sus subdirectorios, al directorio <i>clase</i></p>

Tabla 10.15 Ejemplos del comando *cp*

10.2.9. Mostrar el manual de comandos de UNIX – *man*

El comando *man* [*comando*] muestra la sintaxis de cualquier comando de UNIX así como los modificadores posibles. La tabla 10.16 muestra un ejemplo donde se describe el comando *ps* y sus modificadores, a partir de:

```
$man ps
```

\$ man ps	
NAME	ps - report process status
SYNOPSIS	ps [-aAcdefjlLPy] [-g grplist] [-n namelist] [-o format]... [-p proclist] [-s sidlist] [-t term] [-u uidlist] [U uidlist] [-G gidlist]
DESCRIPTION	The ps command prints information about active processes. Without options, ps prints information about processes that have the same effective user ID and the same controlling terminal as the invoker. The output contains only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the options.

Tabla 10.16 *Ejemplos del comando man*

