Introducción a la Programación

Álvarez Escudero Juan Jesús
Andrade Rodríguez Silvia Alejandra
Becerril Palma Marco Antonio
De la Mora García Mishel
López Chimil Marcos
Nieto Crisóstomo Omar
Trevilla Román Catalina
Valera Paulino Armando
Wals Selvas Jorge Enrique

Biblioteca del Estudiante

UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

Título: Introducción a la Programación
Primera reimpresión, 2008

Álvarez Escudero Juan Jesús, Andrade Rodríguez Silvia Alejandra, Becerril Palma Marco Antonio, De la Mora García Mishel, López Chimil Marcos, Nieto Crisóstomo Omar, Trevilla Román Catalina, Valera Paulino Armando, Wals Selvas Jorge Enrique.

Academia de Informática, Ciclo Básico

Responsable: Jorge Enrique Wals Selvas
Diseño de la portada: Jorge Enrique Wals Selvas
Aarón Ernesto Aguilar Almanza

ISBN: 968-9037-31-5

Hecho e impreso en México

Presentación

La presente recopilación de apuntes ha sido escrita y revisada por los profesores de la Academia de Informática; la estructura y orden del documento permite seguir a la par el programa de estudios de la materia Introducción a la Programación.

El objetivo del presente documento es proporcionar al estudiante que cursa la materia un apoyo y guía de estudio de todos los temas que se revisan en clase, proporcionando en algunos capítulos, ejercicios adicionales y propuestos para que el estudiante ejercite y fortalezca los conocimientos adquiridos en el aula.

En el capítulo uno se estudia los elementos fundamentales de la programación, componentes esenciales de una computadora, modelo de Von Neumann y los sistemas de numeración.

En el capítulo dos se abordan los fundamentos de la programación estructurada y modular, revisando temas como: solución de problemas por computadora, análisis de un problema, su diseño y solución, definición de algoritmo y sus propiedades, representación de algoritmos mediante diagramas de flujo y pseudocódigo, estudio de las estructuras de programación básicas: secuencial, selectiva y repetitiva.

Una vez analizados los elementos fundamentales de la programación estructurada se procede a estudiar en el capítulo tres, la elaboración de programas en lenguaje estructurado utilizando para tal fin el lenguaje C, mostrando como se construye un programa partiendo de los elementos básicos a los más complejos. En el se estudian las variables y constantes, los tipos de datos, operadores aritméticos, relacionales y lógicos, prioridad de los operadores, expresiones, funciones de entrada y salida.

En el capítulo cuatro se examina la estructura secuencial de un programa, la estructura condicional **if**, **if-else** y **switch**, posteriormente se inspeccionan las estructuras repetitivas **for**, **while** y **do-while** en cada estructura se dan ejemplos que demuestran su representación en pseudocódigo y diagrama de flujo, así como su importancia y uso.

En el capítulo cinco se estudia un elemento crucial y de gran importancia en la programación como son las funciones, su concepto, elementos de una función, ejemplos de cómo diseñar funciones, el alcance de las variables locales y globales, paso de parámetros a funciones, recursividad y la elaboración e integración de módulos.

En el capítulo seis se instruye en una de las estructura de datos más utilizadas como son los arreglos, su definición, arreglos unidimensionales y solución de problemas con vectores, arreglos bidimensionales y solución de problemas con matrices y arreglos de caracteres o cadenas.

Finalmente se proporcionan seis apéndices con la siguiente información:

Apéndice A Comando básicos de **MS-DOS** y Linux
Apéndice B Manual de uso del compilador **DEV-CPP** con el Sistema Operativo **Windows**
Apéndice C Paso de parámetros a funciones y apuntadores
Apéndice D Estructuras
Apéndice E Funciones matemáticas
Apéndice F Manual de uso del compilador **gcc** con el Sistema Operativo **Linux**

A continuación se listan los profesores de la Academia de Informática que participaron en la elaboración de estos apuntes.

Profesor:	**Correo Electrónico:**
Álvarez Escudero Juan Jesús	ing_juan_alvarez@yahoo.com.mx
Andrade Rodríguez Silvia Alejandra	aleandrader@yahoo.com.mx
Becerril Palma Marco Antonio	mabecerrilp@gmail.com
De la Mora García Mishel	mish_1915@yahoo.com.mx
López Chimil Marcos	uacmmlc@yahoo.com.mx
Nieto Crisóstomo Omar	nc_omar@yahoo.com.mx
Trevilla Román Catalina	c_trevilla@yahoo.com
Valera Paulino Armando	avalera@hotmail.com
Wals Selvas Jorge Enrique	wals_ucm@yahoo.com.mx

Agradeceremos cualquier observación, sugerencia o colaboración para enriquecer y mejorar estos apuntes.

Ciudad de México, noviembre de 2007

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Capítulo 1
Elementos fundamentales para la Programación.

Tabla de contenido:

Introducción.

Las primeras computadoras se programaban en realidad cableándola en cada ocasión, que se requiriera un nuevo programa. Esto prácticamente equivalía a reconstruir todo en las computadoras cuando se requería de un nuevo programa. La tarea era simplificada gracias a un panel de contactos (muy similar al de los primeros conmutadores telefónicos que eran atendidos por operadoras) con el que era posible enlazar circuitos para crear secciones dedicadas a una actividad específicas. [1]

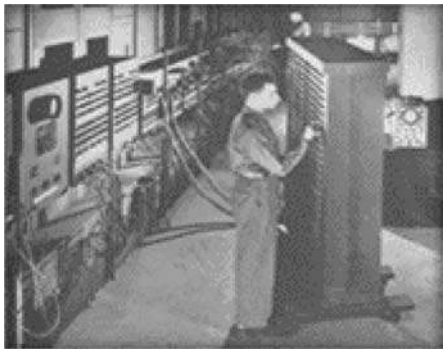


Fig 1.- ENIAC, primera computadora electrónica y su panel de conexiones imagen tomada de referencia [1]

Esto vino a cambiar con el concepto del programa almacenado, un concepto teórico muy importante que fue establecido por el matemático John Von Neumann[1] el 30 de junio de 1945 en un borrador sobre el diseño de la EDVAC. A diferencia de los primeros computadores, Von Neumann proponía que tanto el programa como sus datos fueran almacenados en la memoria de la computadora[2]. Esto no sólo simplificaba la labor de programación al no tener que llevar a cabo el recableado de la computadora sino que además libraba y generalizaba el diseño del hardware para hacerlo independiente de cualquier problema y enfocado al control y ejecución del programa. Este concepto fue tan importante y decisivo que dio lugar al concepto de arquitectura de Von Neumann aún presente en nuestros días.

---

[1] El nombre del celebre matemático húngaro **John Von Neumann**, en otras referencias distintas a las de este capítulo, también se escribe como: **John Von Neummann** e incluso **John Von Newmann**. Esto se debe al manejo del nombre en los diferentes lenguajes.

[2] La **EDVAC** (Electronic Discrete Variable Automatic Computer). Fue una de las primeras computadoras electrónicas. A diferencia de la ENIAC, no era decimal, sino binaria y tuvo el primer programa diseñado para ser almacenado

ENIAC (Electronic Numerical Integrator And Computer). Primera computadora construida para resolución numérica de problemas balísticos

1.1 Modelo de *Von Neumann*.

El modelo de *Von Neumann* está integrado por los siguientes elementos:

1. La unidad aritmético-lógica o ALU.
2. La unidad de control.
3. La memoria.
4. Los dispositivos de entrada/salida.
5. Los Buses.

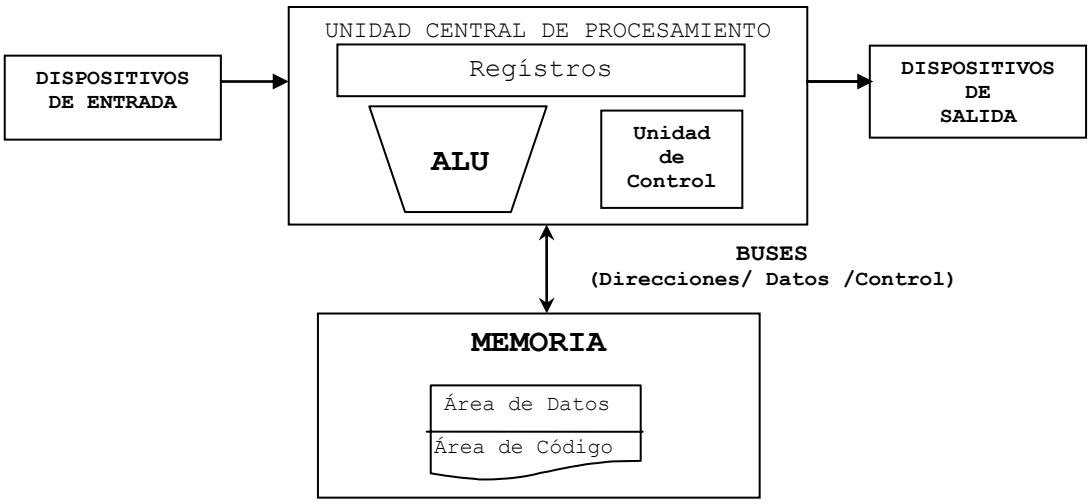


Fig 2.- Esquema lógico de Von Neumann imagen tomada de referencia [1]

¿Cuál es el funcionamiento básico de una computadora?

El funcionamiento básico de una computadora es muy simple:

Recibe los datos del usuario a través de las unidades de entrada.
Procesa los datos con la CPU.
Presenta el resultado mediante las unidades de salida.

Es importante considerar que la CPU no recibe los datos de manera directa de la unidad de entrada ni los envía directamente a la unidad de salida. Existe una zona de almacenamiento temporal llamada memoria RAM, que sirve como lugar de paso obligatorio para acceder a la CPU.

Dentro de la CPU, el funcionamiento es el siguiente: Una vez almacenado el programa a ejecutar y los datos necesarios en la memoria principal, la Unidad de Control va decodificando (analizando) instrucción a instrucción. Al decodificar una instrucción detecta las unidades (ALU, dispositivos de entrada, salida o memoria) implicadas, y envía señales de control a las mismas con las cuales les indica la acción a realizar y la dirección de los datos implicados. Las unidades implicadas a su vez, cuando terminen de operar sobre los datos, enviarán señales a la UC indicando que la acción se ha realizado o bien el problema que ha imposibilitado que se haga.

Principios básicos del funcionamiento de la computadora.

· La CPU es la única que puede procesar los datos (lo cual implica que los datos tienen que llegar de alguna forma a la CPU para ser procesados).
· La CPU sólo puede acceder a los datos o instrucciones almacenados en memoria RAM.

Estos dos principios tienen un origen muy claro que ya fue señalado anteriormente**:** ***Todos los datos, absolutamente todos, tiene que pasar por la memoria RAM para que desde allí puedan ser leídos por la CPU.***

Central Process Unit (CPU)

En la unidad central de procesamiento (CPU) es donde ocurre el procesamiento de datos. La CPU consiste de dos componentes básicos: unidad de control y unidad de aritmética y lógica.

UNIDAD DE ARITMÉTICA LÓGICA - ALU

En la unidad de aritmética lógica (ALU) es donde ocurre el procesamiento real de los datos. Se realizan todos los cálculos y todas las comparaciones y genera los resultados.
Las operaciones que la ALU puede efectuar son: suma, resta, multiplicación y división, manipulación de bits[3] de los registros, y comparación del contenido de dos registros.

UNIDAD DE CONTROL - CU

En la unidad de control (UC) se coordina y controla las demás partes de la computadora. Lee un programa almacenado, una instrucción a la vez, y dirige a los demás componentes para realizar las tareas requeridas por el programa.

REGÍSTROS

Un regístro es una colección de celdas de memoria cada una de las cuales puede almacenar un 1 o un 0. El número de células corresponde al número de bits que se pueden almacenar en el registro. Así hay registros de 4, 16, 32 bits.

Memoria

Guarda temporalmente los datos y las instrucciones del programa durante el procesamiento.
Compuesta por multitud de elementos, cada uno con una dirección única.

[3] El ***BIT*** (**BI**nary digi**T**, dígito binario) es la unidad elemental de información que equivale a un valor binario (0 ó 1) y constituye, dentro de una computadora la capacidad mínima de información.

El parámetro más importante en una memoria es su velocidad de acceso, que mide el tiempo transcurrido desde que la CPU pide el contenido de una celda hasta que ésta puede ser leída.

Unidades de Almacenamiento.

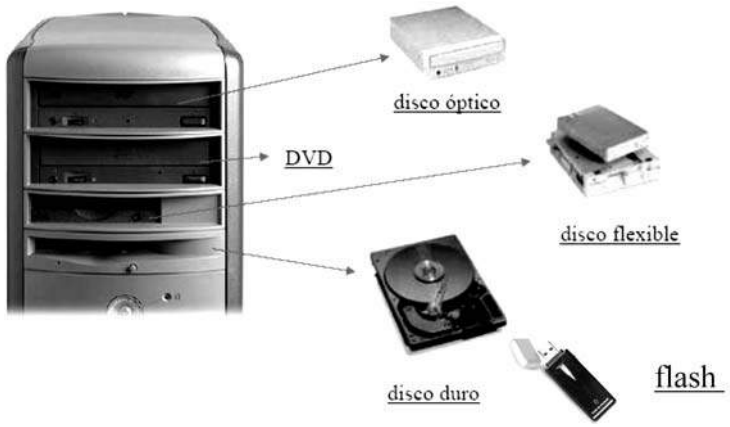


Fig 3.- imagen tomada de referencia [6]

Ya que la memoria principal de una computadora es costosa, es necesario contar con áreas adicionales de memoria para almacenar grandes cantidades de información de manera más económica. Además la memoria principal pierde los datos almacenados al interrumpirse la alimentación eléctrica, por lo que no se puede utilizar para almacenar datos de manera permanente. [2]

Los medios físicos más usuales de almacenamiento secundario son magnéticos y ópticos.

Ejemplos

Dispositivos de almacenamiento magnético

1. Discos duros.
2. Discos flexibles.
3. Cintas Magnéticas.

Dispositivos de almacenamiento óptico

1. CD-ROM.
2. CD-RW.
3. DVD

Dispositivos de almacenamiento electrónico

1. Memoria Flash USB.
2. CD-RW.

Tanto la memoria como las unidades de almacenamiento utilizan unidades para ser medidas entre las cuales se encuentran:

"Bit" es la unidad más pequeña de información en el sistema binario.

"Byte" es la unidad de almacenamiento, incluye 8 bits y puede aguantar hasta 256 valores diferentes, dependiendo de cómo se ordenan los bits. Los bytes representan números o caracteres específicos. Un byte es equivalente a un caracter. Por lo tanto, para almacenar las letras "Hola" en la memoria, la computadora necesitaría 4 bytes, uno para cada caracter.

"Kilobyte" (K) corresponde a kilo que significa 1,000. Sin embargo, como las computadoras emplean números binarios, K se refiere a 1,024 bytes.

"Megabyte" (M) corresponde a 1,024 K o sea 1024 * 1024 = 1,048,576 bytes.

"Gigabyte" (G) corresponde a 1,024 M o sea 1,024 * 1,048,576 = 1,073,741,824 bytes

"Terabyte" (T) corresponde a 1,024 G o sea 1,024 * 1,073,741,824 = 1,099,511,627,776 bytes

Nota: Existen unidades más grandes como se observa a continuación.

B = BYTE; KB = KILO BYTES (MIL BYTES); MB = MEGA BYTES (UN MILLÓN DE BYTES); GB = GIGA BYTES (MIL MILLONES DE BYTES); TB = TERA BYTES (2 BILLONES DE BYTES); PB = PETA BYTES (DOS BILLONES CIEN MIL); EB = EXA BYTES (1018 BYTES).

Unidades de Entrada/Salida

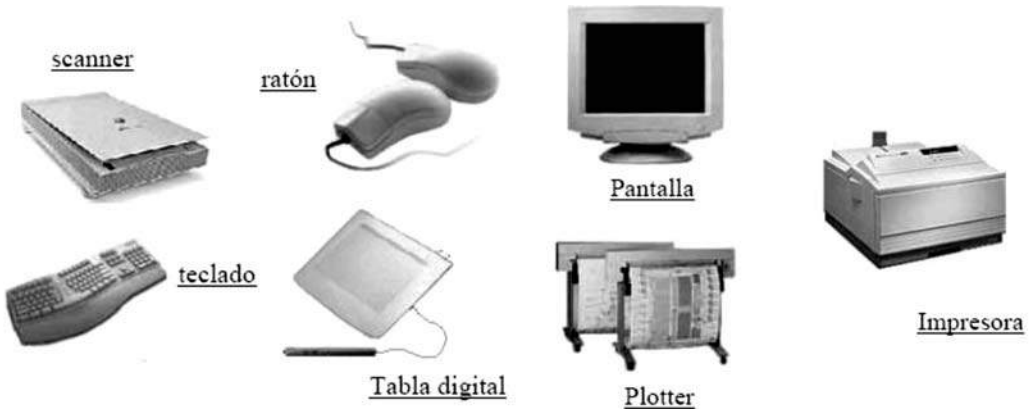


Fig 4.- imagen tomada de referencia [6]

Unidad de Entrada: es el dispositivo por donde se introducen en la computadora tanto datos como instrucciones. La información de entrada se transforma en señales binarias de naturaleza eléctrica. Una misma computadora puede tener distintas unidades de entrada. [2]
Ejemplos.: - teclado - scanner - ratón -micrófono

Unidad de Salida: es el dispositivo por donde se obtienen los resultados de los programas que se están ejecutando en la computadora. En la mayoría de los casos se transforman las señales binarias eléctricas en caracteres escritos o visualizados.
Ejemplos: - monitor - impresora - plotter - bocinas

Bus

La CPU se comunica con la memoria y dispositivos de E/S a través de unos canales denominados buses.

• Un bus se puede ver como un conjunto de líneas de transmisión donde pasa la información tanto de control, datos y direcciones.

• TIPOS DE BUSES:

- Bus de datos.
Este bus es el canal por el cual se conducen los datos entre la CPU y los demás dispositivos (memorias, puertos y otros).
– Bus de direcciones.
El bus de direcciones es un canal unidireccional por el cual la CPU envía las direcciones de memoria para ubicar información en los dispositivos de memoria, puertos u otros dispositivos.
– Bus de control para señales de control.
El bus de control, al igual que el bus de direcciones es unidireccional y se utiliza para controlar la lectura y escritura en las memorias y puertos de E/S. Este bus en general lo emplea la CPU para controlar el flujo de los datos y las direcciones de forma organizada.

Función de la ALU, Unidad de Control y Memoria.

Si pensamos en ¿cómo funciona internamente la computadora?, es decir, ¿cómo se guardan los datos en la memoria?, ¿cómo se almacenan las instrucciones? y ¿cómo se obtienen los resultados usando la computadora?, cuando tenemos un primer acercamiento o contacto con estos equipos a pesar de manejarlos a diario, no es tan fácil de contestar.

A continuación planteamos un primer programa que nos permitirá contestarnos algunas de las preguntas anteriormente planteadas.

Elaborar un programa fuente y su correspondiente programa objeto que realice las siguientes operaciones:

5 - 3 * 4

El **Programa Fuente** es aquel que esta escrito en un lenguaje similar al nuestro (pero inaccesible para la computadora), mientras que el **Programa Objeto** ya esta traducido al código que la maquina reconoce. [5]

Suponga que los datos de entrada están en las celdas de memoria 1 a la 3 y la salida en la celda de memoria 4.

Es necesario considerar lo siguiente:

Suponer a la memoria de la computadora como una especie de almacén electrónico que funciona en forma autónoma del procesador.

Las instrucciones se almacenan en la memoria; mediante el uso de números. Para codificarlas se debe considerar cuántas y cuáles instrucciones habrá disponibles, así como el esquema de codificación por emplear.

A continuación se propone un diccionario electrónico el cual contiene un código adecuado para que a cada instrucción definida corresponda un, y sólo un, valor numérico.

Instrucción	**Código Interno**
Alto	**70**
Carga	**20**
Guarda	**02**
Dividir	**38**
Multiplicar	**36**
Resta	**33**
Suma	**30**

Diccionario Electrónico de Datos

Tabla1.- tomada de referencia [5]

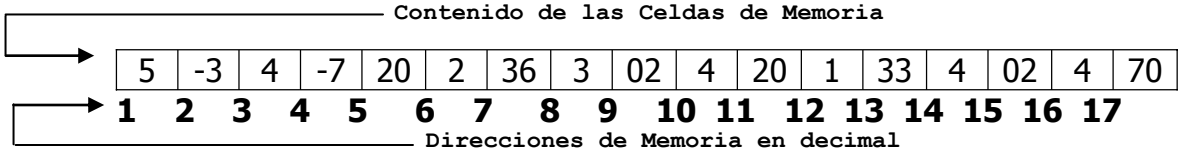
SOLUCIÓN.

Programa Fuente.		Programa Objeto.	
Carga	**2**	**20**	**2**
Multiplica	**3**	**36**	**3**
Guarda	**4**	**02**	**4**
Carga	**1**	**20**	**1**
Resta	**4**	**33**	**4**
Guarda	**4**	**02**	**4**
Alto	**70**	**70**	

Operaciones sobre la memoria.

Decimal	Hexadecimal	Contenido	Código
1	0xF1	5	
2	0xF2	-3	
3	0xF3	4	
4	0xF4	-7	
5-6	0xF5-0xF6	Carga 2	**20**
7-8	0xF7-0xF8	Multiplica 3	**36**
9-10	0xF9-0xFA	Guarda 4	**02**
11-12	0xFB-0xFC	Carga 1	**20**
13-14	0xFD-0xFE	Resta 4	**33**
15-16	0xFF-0xF10	Guarda 4	**02**
17	0xF11	ALTO	**70**

Programa Objeto cargado en memoria



Explicación:
La primera parte de la solución consistió en escribir el programa fuente, que como ya se menciono es aquel escrito en un lenguaje similar al nuestro e introducido por el usuario. Posteriormente la computadora crea un programa objeto, codificando cada una de las instrucciones con un código tomado del diccionario electrónico, en donde por ejemplo: la instrucción carga tiene el código 20, multiplica 36, etc.

Es importante mencionar que el número de una celda de memoria en la computadora esta en hexadecimal, pero para efectos de este ejercicio y para comprensión del lector también se escribe su equivalente en decimal.

En la tabla “Operaciones sobre la memoria”, observamos que los datos se encuentran en las celdas de memoria 1, 2 y 3. El resultado final de las operaciones se escribirá en la celda de memoria 4.

Las instrucciones del programa objeto se cargan de la celda de memoria 5 hasta la celda de memoria 17. Como podemos observar todas las instrucciones de programa objeto tiene 2 valores, esto es porque el primero hace referencia a la instrucción, mientras que el segundo valor hace referencia a las celdas de memoria en donde se tienen guardado los datos o el resultado. Finalmente es importante denotar que solo la última instrucción tiene un valor, esto es porque la instrucción ALTO no requiere acceder a ninguna otra celda.

En términos generales, habrá instrucciones que ocupen una, dos y hasta tres celdas de memoria.

Ejemplos:

- Utiliza una celda de memoria la instrucción ALTO.
- Utiliza dos celdas de memoria la instrucción CARGA, GUARDA, SUMA, etc.

Al final de la solución encontramos la tabla “Programa Objeto cargado en memoria”, el cual nos muestra de una forma muy sencilla como se vería la memoria una vez cargado este programa objeto.

1.2 Sistemas de Numeración.

INTRODUCCIÓN.

La información es todo aquello que puede ser manejado por un sistema, ya sea como entrada, como proceso, o bien como resultado en el caso de las computadoras.

La información, para que sea útil a nuestra computadora debe estar representada por símbolos. Tales símbolos por sí solos no constituyen la información, sino que la representan.

La información se puede clasificar como:

- ✓ Datos numéricos, generalmente combinaciones de números del 0 al 9.
- ✓ Datos alfabéticos, compuestos sólo por letras.
- ✓ Datos alfanuméricos, combinación de los dos anteriores.

Sistemas de Numeración - Codificación

Un sistema de numeración es un conjunto de símbolos y reglas que se utilizan para la representación de números y cantidades que se caracterizan por la base.

Los principales sistemas de numeración utilizados en las computadoras son:

Sistemas de Numeración

Decimal	Binaria	Octal	Hexadecimal
0	0000	0	0xF0
1	0001	01	0xF1
2	0010	02	0xF2
3	0011	03	0xF3
4	0100	04	0xF4
5	0101	05	0xF5
6	0110	06	0xF6
7	0111	07	0xF7
8	1000	010	0xF8
9	1001	011	0xF9
10	1010	012	0xFA
11	1011	013	0xFB
12	1100	014	0xFC
13	1101	015	0xFD
14	1110	016	0xFE
15	1111	017	0xFF

En resumen:

Sistema binario Base= 2 con dígitos 0 y 1.
Sistema octal Base=8 con dígitos 0,1,2,3,4,5,6 y 7.
Sistema decimal Base=10 con dígitos 0,1,2,3,4,5,6,7,8 y 9.
Sistema hexadecimal Base=16 con dígitos 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

En donde las letras en hexadecimal se les asignan los siguientes valores:

A=10, B=11, C=12, D=13, E=14 y F=15.

La computadora es una máquina constituida por circuitos electrónicos de naturaleza digital. Por este motivo, el sistema con el que se representa la información dentro de la computadora es el denominado sistema binario, o de base dos. En él caben sólo dos dígitos o símbolos: 0 y 1.

Dentro de la computadora, los dos estados lógicos aparecen en forma de señales eléctricas; por ejemplo:

Estado lógico 1: Interruptor cerrado, presencia de tensión eléctrica.
Estado lógico 0: Interruptor abierto, ausencia de tensión eléctrica.

CONVERSIÓN DE UN SISTEMA A OTRO.

- **Conversión de un número entero de base b a otro de base 10.**

a) Conversión de binario a decimal

Paso 1. Se multiplica el primer dígito de la derecha por 2^0.
Paso 2. El resultado del paso anterior se suma a la multiplicación del segundo dígito de la derecha por 2^1. Como se observa el exponente se incrementa en 1, conforme se va recorriendo la posición de los dígitos hacia la izquierda.

Ejemplo: Convertir el número binario 10101 a decimal.

1	0	1	0	1
2^4	2^3	2^2	2^1	2^0
$1x2^4$	$0x2^3$	$1x2^2$	$0x2^1$	$1x2^0$
16	0	4	0	1
Suma 21_{10}				

b) Conversión de octal a decimal

Paso 1. Se multiplica el primer dígito de la derecha por 8^0.
Paso 2. El resultado del paso anterior se suma a la multiplicación del segundo dígito de la derecha por 8^1. Como se observa el exponente se incrementa en 1, conforme se va recorriendo la posición de los dígitos hacia la izquierda.

Ejemplo: Convertir el número octal 5052 a decimal.

5	0	5	2
8^3	8^2	8^1	8^0
$5x8^3$	$0x8^2$	$5x8^1$	$2x8^0$
2560	0	40	2
Suma 2602_{10}			

c) Conversión de hexadecimal a decimal

Paso 1. Se multiplica el primer dígito de la derecha por 16^0. Si el dígito es una letra se toma el valor de la letra, es decir, si es A=10, B=11, …..F=15.
Paso 2. El resultado del paso anterior se suma a la multiplicación del segundo dígito de la derecha por 16^1. Como se observa el exponente se incrementa en 1, conforme se va recorriendo la posición de los dígitos hacia la izquierda.

Ejemplo: Convertir el número hexadecimal 2B6 a decimal.

2	B	6
2	11	6
16^2	16^1	16^0
$2x16^2$	$11x16^1$	$6x16^0$
2x256	11x16	6x1
512	176	6
Suma 694_{10}		

- **Conversión de un número entero en base 10 a otro de base b.**

Ahora el procedimiento es inverso:

Paso 1. Hacer las divisiones sucesivas del número entre la base hasta que el dividendo sea menor que la base. Los residuos obtenidos componen el número en base b, pero colocados en orden inverso al que han sido obtenidos.

a) Conversión de decimal a binario.

Ejemplo: Convertir el número decimal 98 a binario.

Dividendos

Cociente	49	24	12	6	3	1	0
2	98	49	24	12	6	3	1
Residuos	0	1	0	0	0	1	1

Se concluye que $98_{10} = 1100010_2$

b) Conversión de decimal a octal.

Ejemplo: Convertir el número decimal 350 a octal.

Dividendos

Cociente	43	5	0
8	350	43	5
Residuos	6	3	5

Se concluye que $350_{10} = 536_8$

c) Conversión de decimal a hexadecimal.

Ejemplo: Convertir el número decimal 3960 a hexadecimal.

Dividendos

Cociente	247	15	0
16	3960	247	15
Residuos	8	7	15
	8	7	F

Se concluye que $3960_{10} = F78_{16}$

- **Conversión entre bases.**

Para las siguientes conversiones se requiere consultar la siguiente tabla de equivalencias.

Decimal	Binaria	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

a) Binario a Hexadecimal.

Paso 1. Se separan las cifras en grupos de cuatro.
Paso 2. Se sustituye el número consultado la tabla de equivalencias.

Ejemplo: Convertir el número binario 11010011 a hexadecimal.

1	1	0	1	0	0	1	1
D				3			
Resultado 11010011_2 es igual a $D3_{16}$							

b) Hexadecimal a Binario.

Paso 1. Se toma cada una de las cifras y se representa con 4 posiciones, se busca el valor de la tabla de equivalencias.

Ejemplo: Convertir el número hexadecimal FD7 a binario.

F				D				7			
15				13				7			
1	1	1	1	1	1	0	1	0	1	1	1
Resultado $FD7_{16}$ es igual a $1111\ 1101\ 0111_2$											

1.3 Representación y codificación ASCII y complemento a dos.

CODIFICACIÓN ASCII.

Básicamente, las computadoras sólo trabajan con números. Almacenan letras y otros caracteres mediante la asignación de un número a cada uno.

ASCII = American Standard Code for Information Interchange (Estándar Americano de Codificación para el Intercambio de Información).

ASCII es un estándar para representar caracteres y símbolos en forma electrónica. Usar estándares aumenta la eficiencia y elimina errores. Para la codificación ASCII usamos 1 byte (pero sólo 7 bits dentro de él, por lo que tiene 128 caracteres). Se trata de código de 7 bits que sustituye las letras del alfabeto y otros caracteres escritos por cifras. [3]

CÓDIGO ASCII. [3]

Binary	Decimal	Hex	Graphic	Binary	Decimal	Hex	Graphic	Binary	Decimal	Hex	Graphic
0010 0000	32	20	(blank) (□)	0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p
0011 0001	49	31	1	0101 0001	81	51	Q	0111 0001	113	71	q
0011 0010	50	32	2	0101 0010	82	52	R	0111 0010	114	72	r
0011 0011	51	33	3	0101 0011	83	53	S	0111 0011	115	73	s
0011 0100	52	34	4	0101 0100	84	54	T	0111 0100	116	74	t
0011 0101	53	35	5	0101 0101	85	55	U	0111 0101	117	75	u
0011 0110	54	36	6	0101 0110	86	56	V	0111 0110	118	76	v
0011 0111	55	37	7	0101 0111	87	57	W	0111 0111	119	77	w
0011 1000	56	38	8	0101 1000	88	58	X	0111 1000	120	78	x
0011 1001	57	39	9	0101 1001	89	59	Y	0111 1001	121	79	y
0011 1010	58	3A	:	0101 1010	90	5A	Z	0111 1010	122	7A	z
0011 1011	59	3B	;	0101 1011	91	5B	[	0111 1011	123	7B	{
0011 1100	60	3C	<	0101 1100	92	5C	\	0111 1100	124	7C	|
0011 1101	61	3D	=	0101 1101	93	5D	]	0111 1101	125	7D	}
0011 1110	62	3E	>	0101 1110	94	5E	^	0111 1110	126	7E	~
0011 1111	63	3F	?	0101 1111	95	5F	_				

Tabla 2.- tomada de referencia [3]

ASCII EXTENDIDO. [3]

Oct	Dec	Hex	Carac	Descripción
244	164	A4	¤	SIGNO MONETARIO
245	165	A5	¥	SIGNO DEL YEN
246	166	A6	¦	BARRA VERTICAL PARTIDA
247	167	A7	§	SIGNO DE SECCIÓN
250	168	A8	¨	DIÉRESIS
251	169	A9	©	SIGNO DE DERECHOS DE COPIA
252	170	AA	ª	INDICADOR ORDINAL FEMENINO
253	171	AB	«	SIGNO DE COMILLAS FRANCESAS DE APERTURA
254	172	AC	¬	SIGNO DE NEGACIÓN
255	173	AD		GUIÓN SEPARADOR DE SÍLABAS
256	174	AE	®	SIGNO DE MARCA REGISTRADA
257	175	AF	¯	MACRÓN
260	176	B0	°	SIGNO DE GRADO
261	177	B1	±	SIGNO MÁS-MENOS
262	178	B2	²	SUPERÍNDICE DOS
263	179	B3	³	SUPERÍNDICE TRES
264	180	B4	´	ACENTO AGUDO
265	181	B5	µ	SIGNO DE MICRO
266	182	B6	¶	SIGNO DE CALDERÓN
267	183	B7	·	PUNTO CENTRADO
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERÍNDICE 1
272	186	BA	º	INDICADOR ORDINAL MASCULINO
273	187	BB	»	SIGNO DE COMILLAS FRANCESAS DE CIERRE
274	188	BC	¼	FRACCIÓN VULGAR DE UN CUARTO
275	189	BD	½	FRACCIÓN VULGAR DE UN MEDIO
276	190	BE	¾	FRACCIÓN VULGAR DE TRES CUARTOS
277	191	BF	¿	SIGNO DE INTERROGACIÓN ABIERTA
300	192	C0	À	A MAYÚSCULA CON ACENTO GRAVE
301	193	C1	Á	A MAYÚSCULA CON ACENTO AGUDO
302	194	C2	Â	A MAYÚSCULA CON CIRCUNFLEJO
303	195	C3	Ã	A MAYÚSCULA CON TILDE
304	196	C4	Ä	A MAYÚSCULA CON DIÉRESIS
305	197	C5	Å	A MAYÚSCULA CON CÍRCULO ENCIMA
306	198	C6	Æ	AE MAYÚSCULA
307	199	C7	Ç	C MAYÚSCULA CON CEDILLA
310	200	C8	È	E MAYÚSCULA CON ACENTO GRAVE
311	201	C9	É	E MAYÚSCULA CON ACENTO AGUDO
312	202	CA	Ê	E MAYÚSCULA CON CIRCUNFLEJO
313	203	CB	Ë	E MAYÚSCULA CON DIÉRESIS
314	204	CC	Ì	I MAYÚSCULA CON ACENTO GRAVE
315	205	CD	Í	I MAYÚSCULA CON ACENTO AGUDO
316	206	CE	Î	I MAYÚSCULA CON CIRCUNFLEJO
317	207	CF	Ï	I MAYÚSCULA CON DIÉRESIS
320	208	D0	Ð	ETH MAYÚSCULA

Oct	Dec	Hex	Carac	Descripción
321	209	D1	Ñ	N MAYÚSCULA CON TILDE (EÑE)
322	210	D2	Ò	O MAYÚSCULA CON ACENTO GRAVE
323	211	D3	Ó	O MAYÚSCULA CON ACENTO AGUDO
324	212	D4	Ô	O MAYÚSCULA CON CIRCUNFLEJO
325	213	D5	Õ	O MAYÚSCULA CON TILDE
326	214	D6	Ö	O MAYÚSCULA CON DIÉRESIS
327	215	D7	×	SIGNO DE MULTIPLICACIÓN (ASPA)
330	216	D8	Ø	O MAYÚSCULA CON BARRA INCLINADA
331	217	D9	Ù	U MAYÚSCULA CON ACENTO GRAVE
332	218	DA	Ú	U MAYÚSCULA CON ACENTO AGUDO
333	219	DB	Û	U MAYÚSCULA CON CIRCUNFLEJO
334	220	DC	Ü	U MAYÚSCULA CON DIÉRESIS
335	221	DD	Ý	Y MAYÚSCULA CON ACENTO AGUDO
336	222	DE	Þ	THORN MAYÚSCULA
337	223	DF	ß	S AGUDA ALEMANA
340	224	E0	à	A MINÚSCULA CON ACENTO GRAVE
341	225	E1	á	A MINÚSCULA CON ACENTO AGUDO
342	226	E2	â	A MINÚSCULA CON CIRCUNFLEJO
343	227	E3	ã	A MINÚSCULA CON TILDE
344	228	E4	ä	A MINÚSCULA CON DIÉRESIS
345	229	E5	å	A MINÚSCULA CON CÍRCULO ENCIMA
346	230	E6	æ	AE MINÚSCULA
347	231	E7	ç	C MINÚSCULA CON CEDILLA
350	232	E8	è	E MINÚSCULA CON ACENTO GRAVE
351	233	E9	é	E MINÚSCULA CON ACENTO AGUDO
352	234	EA	ê	E MINÚSCULA CON CIRCUNFLEJO
353	235	EB	ë	E MINÚSCULA CON DIÉRESIS
354	236	EC	ì	I MINÚSCULA CON ACENTO GRAVE
355	237	ED	í	I MINÚSCULA CON ACENTO AGUDO
356	238	EE	î	I MINÚSCULA CON CIRCUNFLEJO
357	239	EF	ï	I MINÚSCULA CON DIÉRESIS
360	240	F0	ð	ETH MINÚSCULA
361	241	F1	ñ	N MINÚSCULA CON TILDE (EÑE)
362	242	F2	ò	O MINÚSCULA CON ACENTO GRAVE
363	243	F3	ó	O MINÚSCULA CON ACENTO AGUDO
364	244	F4	ô	O MINÚSCULA CON CIRCUNFLEJO
365	245	F5	õ	O MINÚSCULA CON TILDE
366	246	F6	ö	O MINÚSCULA CON DIÉRESIS
367	247	F7	÷	SIGNO DE DIVISIÓN
370	248	F8	ø	O MINÚSCULA CON BARRA INCLINADA
371	249	F9	ù	U MINÚSCULA CON ACENTO GRAVE
372	250	FA	ú	U MINÚSCULA CON ACENTO AGUDO
373	251	FB	û	U MINÚSCULA CON CIRCUNFLEJO
374	252	FC	ü	U MINÚSCULA CON DIÉRESIS
375	253	FD	ý	Y MINÚSCULA CON ACENTO AGUDO
376	254	FE	þ	THORN MINÚSCULA
377	255	FF	ÿ	Y MINÚSCULA CON DIÉRESIS

Tabla 3.- tomada de referencia [3]

Se pueden conseguir tecleando Alt + código ASCII. Por ejemplo, Alt126 = ~, Alt64 = @, etc.

Para poder usar más caracteres que no se consiguen con la especificación ASCII original se recurre a ASCII extendido, que usa los 8 bits permitiendo hasta 256 (de 0 a 255) caracteres distintos, algunos de ellos de alto interés como las vocales acentuadas y la letra ñ.

Ejemplo.

Escriba la palabra “Computadora” en código ASCII.

Si se sabe que en código ASCII del número decimal 65 al 90 son letras mayúsculas del abecedario y del 97 al 122 corresponden las letras minúsculas.

Letra	ASCII
C	67
o	111
m	109
p	112
u	117

t	116
a	97
d	100
o	111
r	114
a	97

Complemento a dos.

Formas de representación binaria

Recordemos que las computadoras utilizan el sistema de numeración binaria y con ello se representan números y caracteres.
La necesidad de representar no sólo números enteros naturales (positivos), sino también valores negativos e incluso fraccionarios (racionales), ha dado lugar a diversas formas de representación binaria de los números. [4] En lo que respecta a los **enteros**, se utilizan principalmente cuatro:

- **Binario sin signo.**
- **Binario con signo**
- **Binario en complemento a uno**
- **Binario en complemento a dos**

Código binario sin signo:
Las cantidades se representan de izquierda a derecha (el bit más significativo a la izquierda y el menos significativo a la derecha) como en el sistema de representación decimal. Los bits se representan por ceros y unos.
Por ejemplo, la representación del decimal 33 es 0010 0001
Es recomendable utilizar un octeto para representar números pequeños, y mantener la costumbre de separar las cifras en grupos de 4 para mejorar la legibilidad.
Con este sistema todos los bits están disponibles para representar una cantidad; por consiguiente, un octeto puede albergar números de rango
0 <= **n** <= 255

Código binario con signo

Ante la necesidad de tener que representar enteros negativos, se decidió reservar un bit para representar el signo. Es tradición destinar a este efecto el bit más significativo (izquierdo); este bit es **0** para valores **positivos** y **1** para los **negativos**. Por ejemplo, la representación de 33 y -33 sería:

+33 ⇨ 0010 0001
-33 ⇨ 1010 0001

Como en un octeto sólo quedan siete bits para representar la cantidad. Con este sistema un Byte puede representar números en el rango:

- 127 <= **n** <= 127

El sistema anterior se denomina **código binario en magnitud y signo**.

Código binario en complemento a uno

En este sistema los números positivos se representan como en el sistema binario en magnitud y signo, es decir, siguiendo el sistema tradicional, aunque reservando el bit más significativo, que debe ser cero. Para los números negativos se utiliza el **complemento a uno**, que consiste en tomar la representación del correspondiente número positivo y cambiar los bits 0 por 1 y viceversa (el bit más significativo del número positivo, que es cero, pasa ahora a ser 1).

Como puede verse, en este sistema, el bit más significativo sigue representando el signo, y es siempre **1** para los números negativos. Por ejemplo, la representación de 33 y -33 sería:

+33 ⇨ 0010 0001
-33 ⇨ 1101 1110

Código binario en complemento a dos:

Para el manejo de operaciones básicas como el caso de la suma, resta, etc. Las representaciones previas no son muy apropiadas. Debido a esto es necesaria la representación de otra forma como el caso del complemento a dos.
En este sistema, los números positivos se representan como en el anterior, reservando también el bit más significativo (que debe ser cero) para el signo. Para los números negativos, se utiliza un sistema distinto, denominado complemento a dos, en el que se cambian los bits que serían ceros por unos y unos por ceros, y al resultado se le suma uno.
Este sistema sigue reservando el bit más significativo para el signo, que sigue siendo 1 en los negativos. Por ejemplo, la representación de 33 y -33 sería:
+33 ⇨ 0010 0001
-33 ⇨ 1101 1110 + 0000 0001 ⇨1101 1111

Método del Complemento a dos.

a) Escribir el número positivo correspondiente en binario.
b) Cambiar ceros por uno y unos por ceros.
c) Sumarle 1.

Ejemplo. Representar en complemento a 2 el número -5.

a) Escribir el número positivo 5 **..... Solución** 0101
b) Cambiar los ceros a unos y unos a ceros **... Solución** 1010
c) Sumarle 1 al resultado anterior **.....Solución** 1010 +1= 1011

Resultado $(-5)_{10}$ -> $(1011)_2$ Complemento a 2
El valor en base 10 de un número binario en complemento a 2 se halla:

Comprobación: $-1(2^3)+ 0(2^2)+ 1(2^1)+ 1(2^0) = -8+0+2+1 = -8+3= -5$.

Técnica para representar números
Complemento a dos.

Técnica para representación de Complemento a dos.
Para la representar un número con complemento a dos, podemos utilizar potencias.
Un aspecto muy importante a ser considerado es que la casilla más a la izquierda será el bit más significativo, es decir, si este bit es cero se representa un número positivo y si es 1 se estará representando a un número negativo.
Ejemplos de complemento a dos.
Representar el número 35 y -35 con 8 bits utilizando la técnica anteriormente mencionada.
El + 35 siendo un número positivo con 7 bits se representa.

0	1	0	0	0	1	1
64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

El - 35 siendo un número negativo con 6 bits se representa.

1	0	1	1	1	0	1
-64	32	16	8	4	2	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

Representar el número 35 y -35 con 8 bits utilizando la técnica anteriormente mencionada.
El + 35 siendo un número positivo con 8 bits se representa.

0	0	1	0	0	0	1	1
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

El - 35 siendo un número negativo con 8 bits se representa.

1	1	0	1	1	1	0	1
-128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

BIBLIOGRAFÍA.

[1] Artículo consultado en Internet.
www.divulgamat.net/weborriak/ Historia/Gaceta/Historia23.pdf
ENIAC, matemáticas y computación científica
Manuel Perera Domínguez.

[2] Fundamentos de los ordenadores (capítulo 1)
Pedro de Miguel Anasagasti
Editorial. Paraninfo

[3] Enciclopédia en línea Wikipedia

[4] Curso de C++
http://www.zator.com/Cpp/E2_2_4a.htm
Tema: Formas de representación binarias de las magnitudes numéricas.

[5] Computación y programación moderna
Perspectiva Integral de la Informática
Guillermo Levine
Addison Wesley

[6] Enciclopédia Encarta 2005
Microsoft

Capítulo 2
Fundamentos de programación estructurada y modular.

Tabla de contenido:

2.1. Solución de problemas por computadora

El uso de la computadora como herramienta para resolver problemas, es la principal razón de entender los *lenguajes de programación y las técnicas para el diseño de programas*. Según [Polya 45] la solución de un problema computacional se divide principalmente en cuatro pasos:

1. **Comprender el problema**: Este paso puede ser muy difícil, pero es definitivamente el más crucial. En este caso se plantea el problema de forma general, además se identifica los datos resultantes, es decir a lo que se quiere llegar; así como, los datos con los que se cuenta (datos de estrada). También es necesario verificar que la información dada sea suficiente para resolver el problema.

2. **Idear un plan**: Una vez que se ha comprendido el problema, se debe pensar en un plan de acción para resolver el problema. Un plan está formado por procedimiento del dato para la obtención del resultado, de acuerdo a la relación que existe en ellos. Las técnicas más generales incluyen:

- Buscar si existe problemas similares conocidos,
- Relacionar el problema original de tal forma que se parezca a uno conocido,
- Restringir el problema para resolverlo en una forma particular,
- Generalizar un problema restringido, y
- Buscar trabajos existentes que puedan ayudar en la búsqueda de una solución.

3. **Ejecutar el plan**. Una vez que el plan está definido, este se debe seguir por completo. Cada elemento del plan debe ser verificado como es aplicado. Si se encuentra partes del plan que no son satisfactorios, se debe revisar el plan.

4. **Evaluación**: Finalmente, el resultado debe ser examinado en orden para asegurar que este es valido y que el problema ha sido resuelto.

Como se sabe, existe más de una forma correcta de resolver un problema. Cuando varias personas se enfrentan con el mismo problema, se espera que cada una de ellas alcance la solución en forma diferente. De cualquier modo, para ser eficiente, una persona debe adoptar un método sistemático de solución de problemas. Cuándo se usa una computadora para resolver un problema, el uso de un método sistemático es crucial.

Basados en el método de solución de problemas de Polya, se introduce un método de solución de problemas de siete pasos [CodeWarrior 95] , que puede ser adaptado a cada uno de los estilos de solución de problemas de cada persona. Este método está relacionado muy estrechamente con el *conocido ciclo de vida del software* (varias etapas en la vida de un programa):

Paso 1 Definición de un problema. El que plantea el problema (el usuario) y el que resuelve el problema, deben trabajar conjuntamente y en orden, para asegurar que ambos entienden el problema. Esto debe conducir a las *especificaciones* completas del

problema, incluyendo las definiciones precisas de los ***datos de entrada*** (datos dados) y los ***datos salida*** (el resultado).
En algunos casos, en esta parte es conveniente plantear al problema como caja negra e identificar los datos necesarios para resolver el problema, así como, los datos resultantes después de procesar la información.

Paso 2 El diseño de la solución. En este paso, se desea definir un bosquejo de la solución. Se inicia con el problema original, se divide en un número de subproblemas. Estos subproblemas, necesariamente más pequeños que el problema original, son fáciles de resolver y sus soluciones serán los componentes de la solución final. El mismo método de "*divide y vencerás*" es aplicado otra vez a los subproblemas hasta que el problema entero ha sido convertido en un plan de pasos muy bien comprendidos. Para ilustrar este apartado, se tomará un problema sencillo que no es de computación, una receta para preparar chiles rellenos. Basados en la receta, el problema se puede dividir en tres subproblemas:

1. Limpiar los chiles.
2. Rellenar los chiles
3. Cocinar los chiles

La figura 2.1 muestra una gráfica con la estructura de la solución del problema en tres niveles. En el primer nivel se muestra el planteamiento del problema (una receta para preparar chiles rellenos). En el segundo nivel se descompone el problema principal en tres componentes: Limpieza de chiles, Rellenar los Chiles, Cocinar los chiles. En el tercer nivel se detalla los sub-componentes de la última parte.

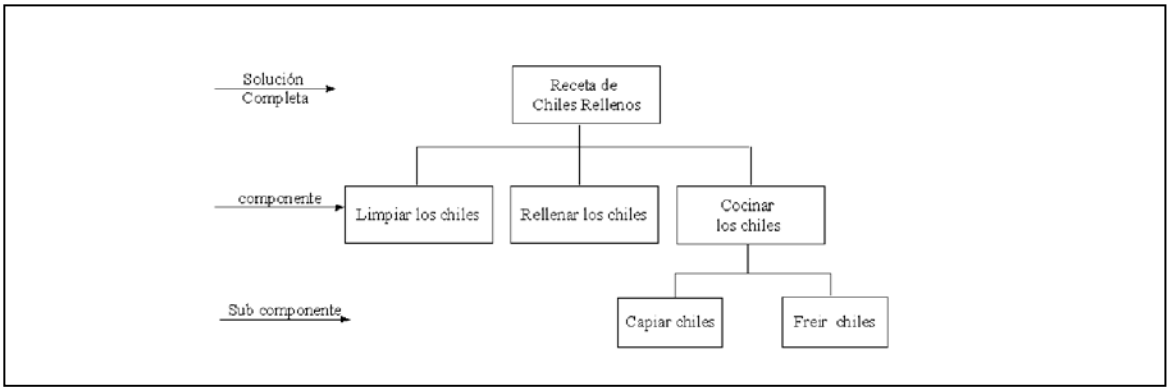
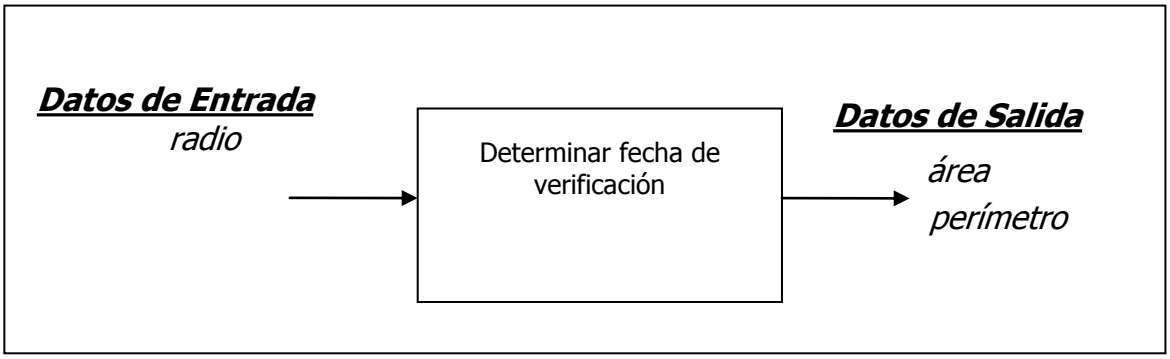


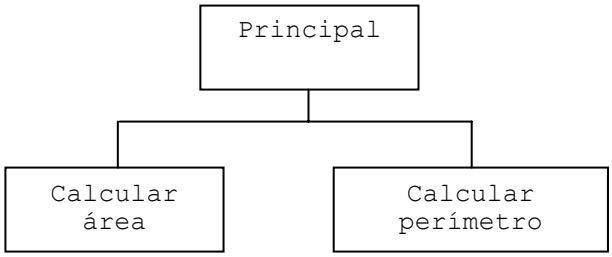
Figura 2.1: Gráfica de la estructura para cocinar chiles rellenos

Ejemplo 2. Elaborar un algoritmo que calcule el área y perímetro de una circunferencia.

***Paso 1*:** En este apartado se identifican los datos de entrada (*radio*) para calcular el área y el perímetro. Hay que tener cuidado con la identificación de los datos de entrada, el valor de **π** no es un dato de entrada debido a que su valor es constante (no cambia su valor) y por tanto se puede determinar, en contraste, el valor de *radio* cambia para diferentes circunferencias y por lo tanto, este valor es dado por el usuario para calcular el área y perímetro de la circunferencia que el usuario requiera.



***Paso 2*:** El problema se puede descomponer en tres bloques: *Principal, Calcula área* y *Calcular perímetro.*



Principal: Ente módulo es la interface con el usuario y llama a los demás módulos según sea requerido.

Calcula área: Módulo encargado de calcular el área dado el radio

Calcular perímetro: Módulo encargado de calcular el perímetro basado en el radio

Paso 3: La solución refinada. La solución diseñada en el paso previo está en una forma de muy alto nivel. En ese paso se observa, una tarea general dividida en varias sub-tareas, que no dan una indicación de como las tareas se realizan. Esta solución general, debe ser refinada agregando más detalles. Cada caja en la gráfica de la estructura del procedimiento en el paso previo (figura 2.1) debe ser asociado con un método preciso que realice cada tarea. Un método preciso consiste de una secuencia de pasos bien definidos, que se llevan a cabo, para ejecutar la tarea correspondiente. A esta secuencia de pasos es llamada *algoritmo*.

Los algoritmos son desarrollados usualmente en *pseudocódigo* -Un lenguaje usado para describir las operaciones realizadas en los datos. Este lenguaje es una simple combinación de lenguaje natural y notaciones matemáticas. El pseudocódigo es independiente de cualquier lenguaje de programación.

Por ejemplo el pseudocódigo para la tarea de limpiar los chiles, en el paso previo, se puede escribir como me muestra en 2.1.

Algoritmo 2.1: Algoritmo para la limpieza de chiles rellenos.

Seleccionar los chiles (poblanos ó chilacas)

Se tuestan en un comal con flama baja

Se colocan dentro de una bolsa de plástico durante unos minutos.

Se pelan cuidadosamente sin romperlos

Se abren de un solo lado y se quitan las semillas si se desea que no queden muy picosos se les quitan las "venas"

La solución particular del algoritmo 2.1 no ha sido completada porque el nivel de detalle es insuficiente: ¿Cuántos chiles se seleccionan? ¿Qué tanto tiempo se tuestan los chiles, hasta quedar negros?, etc. Cada una de las instrucciones del pseudocódigo tiene que ser expandidas para refinar la solución en una forma utilizable. Este proceso de mejorar las instrucciones del pseudocódigo evita cualquier mal entendido. El pseudocódigo 2.2 muestra otro ejemplo donde se calcula el impuesto a un artículo.

Algoritmo 2.2: Pseudocódigo del calculo del impuesto a un articulo.

Si el artículo es suntuoso

Impuesto es igual a IVA del artículo más impuesto suntuoso del artículo

De lo contrario

Impuesto es igual al IVA del artículo

Aquí se asocia el nombre del impuesto con un valor calculado. El pseudocódigo 2.2 se puede usar con más notaciones matemáticas como se muestra en 2.3

Algoritmo 2.3: Pseudocódigo del calculo del impuesto con notaciones matemáticas

```
Si (articulo == suntuoso)
    Impuesto = IVA * costoArticulo + ImpuestoSuntuoso *
    costoArticulo
De lo contrario
    Impuesto = IVA * costoArticulo
Fin Si
```

Paso 4: El desarrollo de la estrategia de prueba. El paso previo produce algoritmos que serán usados para escribir el programa de computadora, que corresponde a la solución del problema. El programa será ejecutado por una computadora y se espera que produzca la solución correcta. Es necesario probar este programa con varias combinaciones de entrada de datos, para asegurar que el programa dará resultados correctos en todos los casos.

Cada uno de esas pruebas consiste de diferentes combinaciones de datos de entrada, las cuales se refieren a los casos de prueba. Para probar el programa completamente, se debe definir un surtido de casos de prueba que cubren, no solo los valores normales de entrada, si no también valores de entrada extremos, que probarán los límites del programa. Además, combinaciones especiales de valores de entrada se necesitaran para aspectos particulares del programa.

```
Por ejemplo: en el programa que calcula impuestos, un caso típico
puede ser:
IVA = 15%,
Impuesto Suntuoso = 10%,
costo del articulo = $100,
resultado del impuesto = $25
```

NOTA: Para todos los casos de prueba, el resultado esperado debe ser calculado y especificado antes de proseguir con el siguiente paso.

Paso 5: Programa de codificación y prueba. Los algoritmos en pseudocódigo no pueden ser ejecutados directamente por la computadora. El pseudocódigo debe ser primero convertido a lenguaje de programación, a este proceso se le conoce como *codificación*.
El pseudocódigo producido por la solución de refinamientos, en el paso 3, es usado como esquema del programa. Dependiendo del lenguaje de programación usado, este paso de codificación puede ser de forma mecánica; de cualquier modo, todos los lenguajes de programación tienen limitaciones. Esas limitaciones algunas veces pueden hacer que el paso de codificación sea totalmente un reto. Algunos de los lenguajes de programación más comunes son: Pascal, C, C++, Java, Python, etc. Para dar un ejemplo, el pseudocódigo del cálculo del impuesto, se puede codificar en lenguaje C como:

Algoritmo 2.4: Codificación en lenguaje C del pseudocódigo para el calculo del impuesto.

```
if (articulo == suntuoso)
{
   Impuesto = IVA * costoArticulo +
   ImpuestoSuntuoso * costoArticulo;
}
else
{
   Impuesto = IVA * costoArticulo;
}
```

Una vez que los algoritmos han sido codificados, el programa de computadora resultante, tiene que ser verificado usando la estrategia de prueba. El programa debe ejecutarse para cada uno de los casos de prueba desarrollados en el Paso 4, y los resultados producidos deben coincidir con los calculados durante el desarrollo de la estrategia de prueba. Si existe una discrepancia, el resultado producido debe ser analizado en orden para descubrir la fuente de error. El error puede estar en uno de los siguientes cuatro lugares:

- *En la codificación:* el error pudo haberse formado cuando el algoritmo fue trasladado a lenguaje de programación.
- *En el algoritmo:* el error pudo haber estado en el algoritmo y nunca se había notado.
- *En el diseño del programa*: el diseño del programa puede estar defectuoso y conducir a errores durante la ejecución.
- *En el cálculo de los resultados de prueba*: Los resultados, en los casos de prueba, pudieron haberse calculado mal.

Una vez que el error se ha descubierto, se debe hacer la revisión apropiada y se deben ejecutar nuevamente las pruebas. Esto es repetido hasta asegurar que el programa produce una correcta solución al problema en todas las circunstancias. A este proceso de codificación y prueba es llamado *implementación*.

Paso 6: Terminación de la documentación.
La documentación empieza con el primer paso del desarrollo del programa y continúa a lo largo del tiempo de vida del programa. Una documentación del programa debe incluir lo siguiente:

- Explicación de todos los pasos del método,
- La decisiones del diseño que fueron hechas, y
- Los problemas que fueron encontrados durante la escritura y prueba del programa.

La documentación final debe también incluir una copia impresa del programa, llamada *listado del programa*. Este listado debe ser legible, manteniendo un cuidado en el arreglo, de la misma forma como se hace en un reporte técnico. Los programas se pueden hacer más fáciles, de manera que el lector los comprenda, si todas las partes complicadas incluyen una explicación en forma de comentarios. Después, cuando el programa se cambie, esa explicación ayudará a hacer los cambios más fáciles para implementar.
Para completar, la documentación debe incluir algunas instrucciones de usuario. En los programas que se usarán por personas no familiarizadas con las computadoras, es necesario incluir el manual de usuario que explique, sin un vocabulario técnico, el uso del programa, como preparar los datos de entrada y como interpretar los resultados del programa.

Paso 7: Mantenimiento del programa. El mantenimiento del programa no es parte directa del proceso de implementación original. Este incluye todas las actividades que ocurren después que el programa está en funcionamiento. El programa puede fallar en un conjunto particular de circunstancias, en las cuales no se probó; esta etapa se encarga de corregir estas fallas no contempladas. Así, el mantenimiento del programa en gran parte,

completa el proceso de implementación. El programa de mantenimiento, también incluye el mejoramiento de la implementación descubierta mientras se usa el programa. El programa de mantenimiento incluye:

- Eliminación de nuevos errores de programa detectados
- Modificación del programa actual
- Agregar nuevas características al programa
- Actualización de la documentación

El mantenimiento de la documentación, producido específicamente para ayudar a las personas que se encargan del mantenimiento de programa, puede incluir documentos de diseño, código del programa e información relacionada a las pruebas.

2.2 Solución a los pasos 1 y 2

2.2.1 Análisis del problema (paso 1)

El propósito del análisis de un problema es ayudar al programador para llegar a una cierta comprensión de la naturaleza del problema. El problema debe estar bien definido si se desea llegar a una solución satisfactoria.
Para poder definir con precisión el problema, se requiere que las especificaciones de la *entrada* y la *salida* sean descritas a detalle. Una buena definición del problema, junto con una descripción detallada de las especificaciones de la entrada y la salida, son los requisitos más importantes para llegar a una solución eficaz.
El análisis del problema exige una lectura previa del problema a fin de obtener una idea general de lo que se solicita. La segunda lectura, servirá para responder a las siguientes preguntas:

- ¿Qué información debe proporcionar la resolución del problema?
- ¿Que datos se necesitan para resolver el problema?

La respuesta a la primera pregunta indicará los resultados o las *salidas* del problema. La respuesta a la segunda pregunta indicará qué datos se proporcionan o las *entradas del problema*.

Ejemplo 2.2.1 Leer el radio de un círculo y calcular e imprimir el área y el perímetro.

Las *entradas* de datos en este problema se concentran en el **radio** del círculo. Dado que el radio puede tomar cualquier valor dentro del rango de los números reales, el tipo de datos debe ser real.
Las *salidas* serán dos variables: **Área** y **Perímetro**, que también serán de tipo real, por tanto:

Entradas: Radio del círculo (variable **R**)

Salidas: Área y Perímetro del círculo (variable **A** y **P**, respectivamente)

Es decir, se requieren las siguientes variables:
Variables: **R**, **A** y **P** de tipo real.

2.2.2 El diseño de la solución (paso 2)

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

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

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se rompen en *subproblemas* que sean más fácil de solucionar que el original. A este método se le conoce, como ya se ha mencionado, "*divide y vencerás*" (*divide and conquer*) y consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar el área y el perímetro de un círculo se puede dividir en tres problemas más simples o *subproblemas* (figura 2.2).

La descomposición del problema original en subproblemas más simples y después, dividir estos subproblemas en otros más simples, se denomina **diseño descendente** (*top-down design*). Normalmente los pasos diseñados en el primer esbozo del algoritmo, son incompletos e indicarán solo unos pocos pasos. Tras esta primera descripción, éstos se amplían en una descripción más detallada con pasos más específicos. Este proceso se denomina **refinamiento por pasos o gradual** (*stepwise refinement*).

Para problemas complejos, se necesitan con frecuencia diferentes *niveles del algoritmo*, antes de que se pueda obtener un algoritmo claro, preciso y completo.

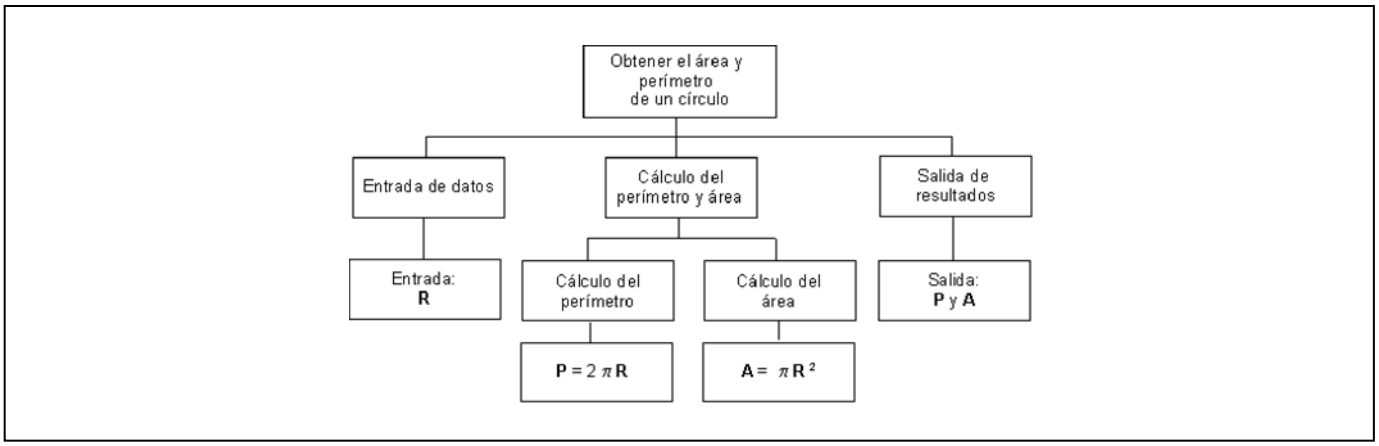


Figura 2.2: Diagrama con tres niveles de detalle, en la solución del problema para encontrar el área y perímetro de un círculo **.**

El problema del cálculo del perímetro y el área se un círculo se puede descomponer en *subproblemas* más simples:
1) leer datos de entrada
2) calcular el área y perímetro y
3) escribir los resultados (datos de salida).

Las *ventajas* más importantes del diseño descendente son:

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

Tras los pasos anteriores (*diseño descendente* y *refinamiento por pasos*) es preciso representar el algoritmo mediante formas específicas como son: *diagrama de flujo*, *pseudocódigo*, etc.
En la siguiente sección se hablará más a detalle de los algoritmos, sus características, representación y las estructuras básicas.

2.3 Algoritmos (paso 3)

2.3.1 Definición de algoritmo

Un *algoritmo* es un plan preciso para ejecutar una secuencia de acciones ordenadas para alcanzar un propósito propuesto en un tiempo finito, es decir, es el conjunto de operaciones y procedimientos que debe seguirse para resolver un problema. Algunos ejemplos de algoritmos son:

- Preparar la cena
- Calcular el promedio de un grupo de valores
- Cambiar una llanta de un carro
- Jugar con dados
- Decidir cuanto cobrar a la entrada del cine

Observe que cada una de los algoritmos de arriba, especifican dos cosas:

1. Una acción especificada por un verbo como: ”preparar”, ”jugar”, ”cobrar”
2. Los datos que participan, se especifican por un sustantivo o por un grupo de palabras que actúan como sustantivo como: ”una llanta de carro”

2.3.2 Propiedades generales de los algoritmos

Un algoritmo debe poseer las siguientes cuatro propiedades [CodeWarrior 95]:

1. **Completo**: un algoritmo que es completo, todas sus acciones deben estar definidas de forma exacta. Por ejemplo, considere el siguiente “algoritmo”:

Dirección de una tienda:

1) Tome la calle 16, hasta encontrar la calle 9.
2) De vuelta el la calle 9 y recorra alrededor de tres cuartos de kilómetro y encontrará en el lado izquierdo la tienda.

   ¿En que dirección debe manejar en la calle 16, norte o sur? Si se tiene conocimiento del área posiblemente conocerá hacia que dirección. Por tanto esta dirección, no es un algoritmo.

2. **No ambiguo**: Un conjunto de instrucciones no serán ambiguas, si solo hay una forma de interpretarlas. Siguiendo con el ejemplo de la dirección, este procedimiento es ambiguo, porque al llegar a la calle 9 no se sabe para donde dar vuelta, a la izquierda o a la derecha. La ambigüedad se puede resolver solo a prueba y error (se prueba primero por la derecha si no se encuentra en un kilometro se regresa en dirección opuesta).

3. **Determinístico** (**definido**): Esta tercera propiedad significa que si las instrucciones son seguidas, el resultado deseado siempre se alcanzará. El siguiente conjunto de instrucciones no satisfaces esta condición:

Algoritmo para convertirse en millonario:

1. Saque todo su dinero del banco y cambiarlo por monedas
2. Vaya a las vegas
3. Juegue en una máquina traga-monedas hasta que gane cuatro millones o hasta que se le acabe el dinero.

Claramente este "algoritmo" no siempre alcanza el resultado deseado de convertirse en millonario. Es mas probable que termine sin dinero. Sin embargo, solo se puede alcanzar la meta de convertirse en millonario, ganando cuatro millones. El punto es que, no se tiene la certeza de alcanzar el resultado. Este "algoritmo" toma un conjunto de instrucciones no determinísticas y de esta forma no constituye un algoritmo.

4. **Finito**: El cuarto requerimiento significa que las instrucciones deben de terminar después de un limitado número de pasos. El "algoritmo" para convertirse en millonario falla en este criterio también. Es posible que nunca alcance el objetivo de convertirse en millonario o perder todo el dinero. En otras palabras, si se sigue las instrucciones, podría jugar en la máquina por siempre.

El requerimiento de finito no solo significa la terminación después de un número de pasos, también significa que las instrucciones deben usar un finito numero de variables para alcanzar el resultado deseado.

La definición de un algoritmo debe describir tres partes: Entrada, *Proceso* y *Salida*.

2.3.3 Atributos deseables de los algoritmos

Aunque un algoritmo pueden satisfacer el criterio de precisión, no ser ambiguo, determinístico y finito; este algoritmo no puede ser aceptable para un problema con solución computacional. Los atributos básicos deseados que un algoritmo debe reunir son [CodeWarrior 95]:

- **Generalidad**: Un algoritmo debe resolver una clase de problemas, no solo un problema. Por ejemplo, un algoritmo para calcular el promedio de cuatro números, no es tan efectivo de forma general, como uno que calcula el promedio de un número arbitrarios de valores.

- **Buena estructura**: Este atributo es aplicado a la construcción del algo- ritmo. Un algoritmo bien estructurado debe ser creado usando bloques bien construidos que facilitan la:
  - Explicación
  - Comprensión
  - Prueba, y
  - Modificación del algoritmo

Los bloques, a partir de los cuales el algoritmo se construyó, deben estar interconectados de tal forma que uno de ellos puede ser fácilmente reemplazado con una mejor versión para mejorar el algoritmo completo, sin tener que reconstruir todo el algoritmo.

- **Eficiencia**: La velocidad de operación del algoritmo es frecuentemente una propiedad importante, como también el tamaño o lo compacto. Inicialmente, estos atributos no son importantes. Primeramente se debe crear los algoritmos bien estructurados, que realice la tarea deseada bajo todas las condiciones. Entonces, y solo entonces, se mejorará el algoritmo con la eficiencia.

- **Fácil de usar**: Esta propiedad describe la conveniencia y facilidad con que el usuario puede aplicar el algoritmo a sus datos. Algunas veces, cuando un algoritmo se hace fácil de comprender para el usuario, también se hace difícil en su diseño (y vice versa).

- **Elegante**: Esta propiedad es difícil de definir, pero se puede relacionar con las cualidades de armonía, balance, economía de estructuras y contraste, que contribuyen a la belleza en el arte de lo apreciable.

Otras propiedades deseables, tales como **robustez** (resistencia a fallas cuando se presentan datos inválidos) y **economía** (costo efectivo), se dejan aun lado para comprender solo los atributos básicos.

2.4 Representación de algoritmos

Los algoritmos pueden ser representados en muchas formas. Para cualquier algoritmo puede haber muchas representaciones de este, algunas mucho mejores que otras. No existe alguna representación que sea la mejor para todos los algoritmos. A continuación de describen algunas posibles representaciones:

- *Verbal*: El algoritmo es expresado en palabras.
- *Algebraica*: El algoritmo es expresado matemáticamente con símbolos y fórmulas.
- *Tabular*: El algoritmo es expresado por una o más tablas.
- *Diagrama de flujo*: El algoritmo es representado en la forma de un diagrama con cajas de acción, ligadas por líneas que muestran el orden en que se ejecutan o la secuencia de acciones. Este se refiere como el flujo de control.
- *Pseudocódigo*: El algoritmo es representado como un conjunto de instrucciones escritas, usando una mezcla de lenguaje natural y notaciones matemáticas. La forma de las instrucciones son similares a las que tiene los lenguajes de programación.

2.4.1 Representación en forma verbal

Los algoritmos 2.5 y 2.6 muestran ejemplos de la representación verbal de los algoritmos. Estos aparecen en una forma verbal simple y se podrá representar mas adelante en otra forma.
El algoritmo 2.5 especifica todo el procedimiento para el juego de dados llamado *crash*. El segundo algoritmo 2.6 determina los pasos necesarios para saber si un año es bisiesto.

Algoritmo 2.5: Juego de dados "*craps*".

Dos dados son tirados. Si la suma de las caras superiores es de 7 u 11, entonces el jugador gana. Si en la primer tirada la suma es 2, 3 ó 12 (conocido como "craps "), el jugador pierde. Si en la primer tirada la suma es 4, 5, 6, 8, 9 ó 10, entonces dicha suma se convierte en "puntos" o en la tirada. Para ganar, el jugador deberá continuar tirando los dados hasta que haga su "tirada" o sus "puntos". El jugador perderá si antes de hacer su tirada, sale una tirada de 7.

Algoritmo 2.6: Año bisiesto.

Un año bisiesto es divisible por cuatro pero, si es divisible por 100, entonces no es un año bisiesto, amenos que también sea divisible por 400.

2.4.2 Representación en forma algebraica (fórmulas y expresiones)

Muchos algoritmo, especialmente en matemáticas e ingeniería, los algoritmos son expresados en forma matemática como una fórmula o expresión algebraica. Por regular es la forma más conveniente para llevarlos a una computadora. Algunos ejemplos se muestran en los algoritmos 2.7 y 2.8.

El algoritmo ilustrado en 2.7 muestra la forma de convertir escalas de temperatura en grados Fahrenheit a grados centígrados o Celsius y viceversa.

Algoritmo 2.7: Conversión entre escalas de temperatura

$$^{\circ}C = \frac{5}{9}(^{\circ}F - 32)$$

$$^{\circ}F = \frac{9}{5}C + 32$$

donde:
$^{\circ}C$: representan los grados centígrados,
$^{\circ}F$: representan los grados Fahrenheit.

Para obtener las medidas estadísticas como son la media y la varianza se muestran en el algoritmo 2.8. La media μ de *N* valores dados, se puede encontrar sumando todos los valores y dividiendo esta suma por *N*. La varianza σ^2 es la cantidad de variación de los valores con respecto al valor de la media. Esta se calcula tomando el promedio de las diferencias de los valores y la media al cuadrado.

Algoritmo 2.8: Algoritmo para las formulas de la media y la varianza

$$\mu = \frac{(x_1 + x_2 + \cdots + x_N)}{N}$$

$$\sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_N - \mu)^2}{N}$$

donde:
μ: es la media,
σ^2: representan la varianza.

2.4.3 Representación en forma tabular (tablas, arreglos y matrices)

Las tablas son cuadrículas rectangulares que almacenan valores. A menudo es conveniente la representación de algoritmos utilizando esta forma. Diferentes tipos de tablas se pueden encontrar: en matemáticas (matrices), en negocios (tablas de decisión o hojas de cálculo), en lógica (tablas de verdad) y en computación (arreglos).
El algoritmo para calcular el número de días en cada mes es mostrado en 2.9. En esta tabla se muestra los meses representados por los números enteros del 1 al 12. A cada mes le corresponde un número que indica el número de días que tiene el mes.
En el caso de febrero (mes 2) depende de si el año es bisiesto, en tal caso tendrá 29 días y si no es bisiesto tendrá 28 días.

Tabla 2.9: Algoritmo para obtener los días de cada uno de los meses.

Mes	Días	
1	31	
2	Año bisiesto	29
	Año no bisiesto	28
3	31	
4	30	
5	31	
6	30	
7	31	
8	31	
9	30	
10	31	
11	30	
12	31	

El algoritmo mostrado en la tabla 2.10, compara tres variables A, B y C; y determina cual de los dos valores (0 ó 1) aparece en mayor número. Por ejemplo, si A = 1, B = 0 y C = 1, entonces el valor de Mayoría es 1 (antepenúltimo renglón de la tabla). En lógica, tablas similares se llaman tablas de verdad.

Tabla 2.10: Algoritmo para obtener la mayoría.

A	B	C	Mayoría
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

2.4.4 Representación en diagramas de flujo de control

Los diagramas de flujo de control son una de las formas más comunes de mostrar la secuencia de acciones en un algoritmo. Ellos consisten de cajas unidas por líneas con flechas, mostrando el flujo de la secuencias de acción. Las cajas son de diferente forma dependiendo si representan acciones, decisiones ó afirmaciones. La tabla 2.11 muestra los símbolos principales para la construcción de un diagrama de flujo [Joyanes 90].

Tabla 2.11: Símbolos principales de los diagramas de flujo

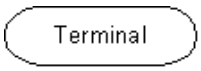
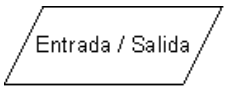
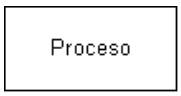
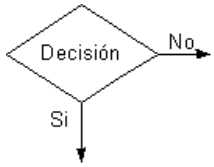
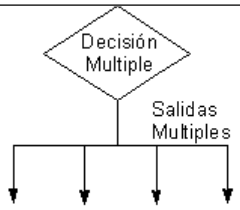
Símbolo	Descripción
	*Terminal*. Representa el comienzo "*inicio"* y el final "*fin"* de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa.
	*Entrada / Salida*. Cualquier tipo de introducción de datos a la memoria, desde cualquier tipo de periféricos de "*entrada*" o registro de la información procesada en un periférico de "*salida*".
	*Proceso*. Cualquier tipo de operación que pueda originar cambios de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etc.
	*Decisión*. Indica operaciones lógicas o de comparación entre datos normalmente dos, y en función del resultado, determina cuál camino debe seguir. Normalmente tiene dos salidas a las respuestas "*Si*" (*verdadero*) o "*No*" (*falso*), pero puede tener tres o más según el caso.
	*Decisión múltiple*. En función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado.
	*Conector*. Enlaza dos partes cualesquiera de un diagrama a través de un conector en la entrada. se refiere a la conexión en la misma página del diagrama.

Tabla 2.11: Símbolos *principales* de los diagramas de flujo (continuación)

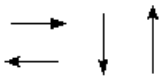
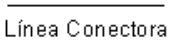
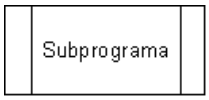
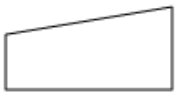
Símbolo	Descripción
	*Líneas de flujo.* Indicador de dirección o líneas de flujo que determinan el sentido de la ejecución de las operaciones.
	*Línea conectora*. Línea que conecta dos símbolos.
	*Conector*. Conecta dos puntos del organigrama situados en páginas diferentes.
	*Llamada a subprograma*. Llama a un proceso determinado o subrutina (en el lenguaje C se le conoce como función). Una subrutina es un módulo independiente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal.

Tabla 2.12: Símbolos *secundarios* de los diagramas de flujo

Símbolo	Descripción
	*Pantalla*. Representa la salida a la pantalla, en ocasiones se usa en lugar del símbolo E/S.
	*Impresora*. Indica salida a la impresora, en ocasiones se usa en lugar de símbolo de E/S.
	*Teclado*. Entrada por medio del teclado, se utiliza en ocasiones en lugar del símbolo de E/S.
	*Comentario*. Este símbolo es utilizado para agregar comentarios clasificadores a otros símbolos del diagrama de flujo.

Además, existen otros símbolos secundarios para la construcción del diagrama de flujo. La tabla 2.12 muestra cuatro de ellos [Joyanes 90].

Ejemplo: desarrollar el algoritmo que calcule la media de *N* números dados, primero se tendrá que descomponer el problema en una serie de pasos secuenciales, que se describen a continuación:

1. Inicio
2. Leer la cantidad de números *N*
3. Inicializar la variable *Contador* y *Suma* a cero
4. Mientras *Contador* sea menor a *N*, realizar lo siguiente:
   - Leer *Num*
   - Agrega el nuevo valor *Num* a *Suma*
   - Incrementa *Contador* en uno
5. Obtener la *Media* dividiendo la *Suma* entre *N*
6. Imprime el valor del Promedio

El diagrama de flujo, del ejemplo previo, se muestra en la figura 2.3

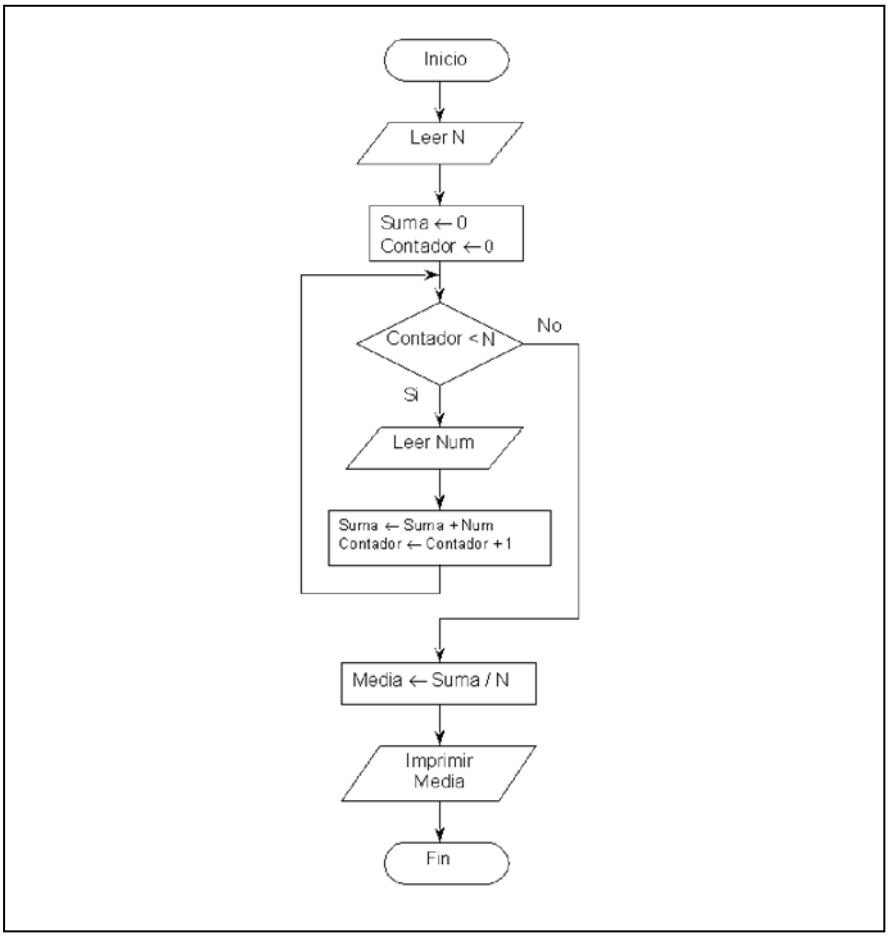


Figura 2.3: Diagrama de flujo para obtener la media de N números dados.

2.4.5 Representación en pseudocódigo

Otra forma de representar el flujo de control en un algoritmo es a través de pseudocódigo, una mezcla de lenguaje natural y notación matemática, independiente del lenguaje de programación. El algoritmo 2.1 muestra la representación del pseudocódigo para el juego de dados *craps*.

Pseudocódigo 2.1: Juego de dados *Craps*

```
inicio
    Tirar dos dados y obtener la Suma
    si Suma == 11 ó 7 entonces
        Gana Usuario
    Sino si  Suma == 2,3 ó 12 entonces
        Pierde el Usuario
    sino
        Establecer Puntos a Suma
        Tirar dos dados y obtener el Total
        mientras (Total ≠ 7) y (Total ≠ Puntos) hacer
            Tirar dos dados y obtener el Total
        fin mientras
        si Total = 7 entonces
            Pierde el Usuario
        sino
            Gana el Usuario
        fin Si
    fin Si
fin
```

2.5 Teorema de la programación estructurada: Estructuras básicas

Un *programa propio* puede ser escrito utilizando básicamente tres tipos de estructuras de control [Joyanes 90]:

- *Secuenciales*
- *Selectivas*
- *Repetitivas*

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

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

La *programación estructurada* significa la escritura de programas de acuerdo a las siguientes reglas [Joyanes 90]:

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente o ascendente.
- Cada módulo se codifica utilizando las tres estructuras de control básicas: *secuenciales*, *selectivas* y *repetitivas* (con la ausencia total de la sentencia **GOTO**).

2.5.1 Estructura secuencial

La estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que, la salida de una instrucción es la entrada de la siguiente y así sucesivamente hasta el fin del proceso. La figura 2.4 muestra una estructura secuencial. La estructura secuencial tiene una entrada y una salida. Su representación en pseudocódigo se muestra en 2.2.

Pseudocódigo 2.2: Pseudocódigo de una estructura secuencial.

```
inicio
    Instrucción 1
    Instrucción 2
    .
    .
    .
    Instrucción n
fin
```

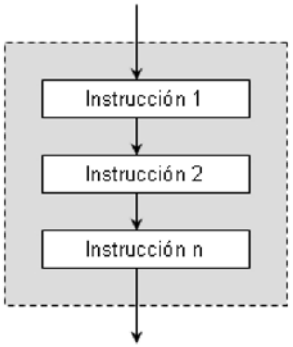


Figura 2.4: Forma secuencia.

Ejemplo 2.5.1 Obtener la distancia entre dos puntos (x_{ini}, y_{ini}) y (x_{fin}, y_{fin}).

Para obtener la distancia de dos puntos, se puede utilizar la siguiente fórmula:

$$distancia = \sqrt{(x_{fin} - x_{ini})^2 + (y_{fin} - y_{ini})^2}$$

El diagrama de flujo y el pseudocódigo, se muestran en 2.5 y 2.3 respectivamente.

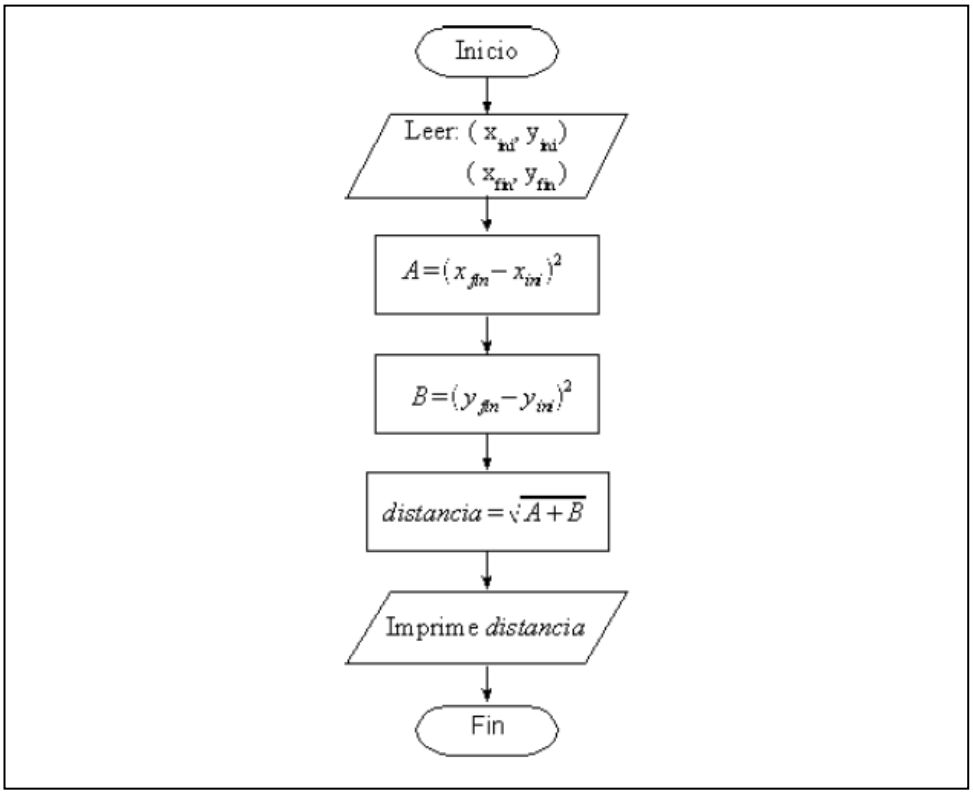


Figura 2.5: Diagrama de flujo para obtener la distancia entre dos puntos

Pseudocódigo 2.3: Pseudocódigo para obtener la distancia de dos puntos.

```
inicio
    Leer x_ini, y_ini, x_fin, y_fin
    A ← (x_fin − x_ini)^2
    B ← (y_fin − y_ini)^2
    distancia ← √(A + B)
    Imprime distancia
fin
```

2.5.2 Estructura selectiva

La estructura selectiva se utiliza para tomar decisiones lógicas; de ahí que se les llame también *estructura de decisión o alternativas*.

En las estructuras selectivas se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La representación de una estructura selectiva se hace con palabras en pseudocódigo: **Si**, **entonces**, **Sino** (en inglés: **if**, **then**, **else**) en cuanto a la representación en el diagrama de flujo, se realiza usando un rombo.

Las estructuras selectivas o alternativas pueden ser:

- *Simples*
- *Dobles*
- *Múltiples*

Alternativa simple (si-entonces/if-then)

La estructura alternativa simple **si-entonces** (en inglés **if-then**) ejecuta una acción determinada cuando se cumple una condición. La selección **si-entonces** evalúa la condición y

- Si la condición es verdadera, entonces ejecuta acción
- Si la condición es falsa, entonces no hace nada

La representación gráfica de la estructura condicional simple se muestra en la figura 2.6 y el pseudocódigo se muestra en 2.4 y 2.5 para español e inglés respectivamente.

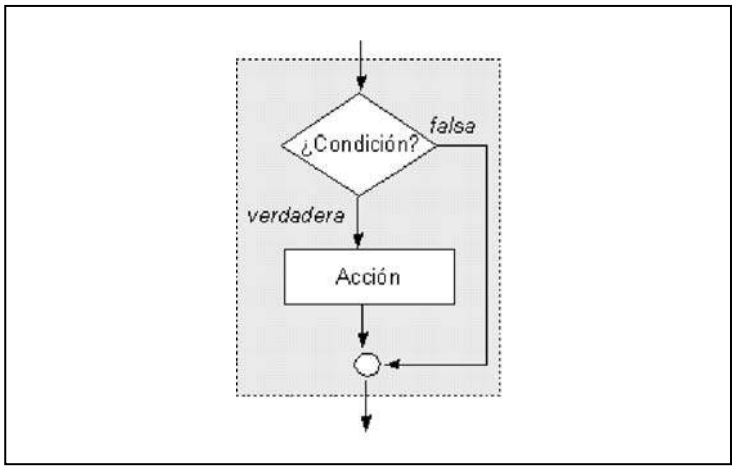


Figura 2.6: Diagrama de flujo de la estructura alternativa simple.

Pseudocódigo 2.4: Pseudocódigo para la la estructura alternativa simple en español.

```
inicio
    si condición entonces
        Acción
    fin Si
fin
```

Pseudocódigo 2.5: Pseudocódigo para la la estructura alternativa simple en inglés.

```
begin
    if condición then
        Acción
    end
end
```

Alternativa doble (si-entonces-sino/if-then-else)

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento de una condición determinada.

Si la *condición*, en la figura 2.7, es verdadera entonces se ejecuta la *acción 1*.
Si la *condición* es falsa, entonces se ejecuta la *acción 2*.

La representación de la estructura alternativa doble, en pseudocódigo, se muestra en 2.6 y 2.7

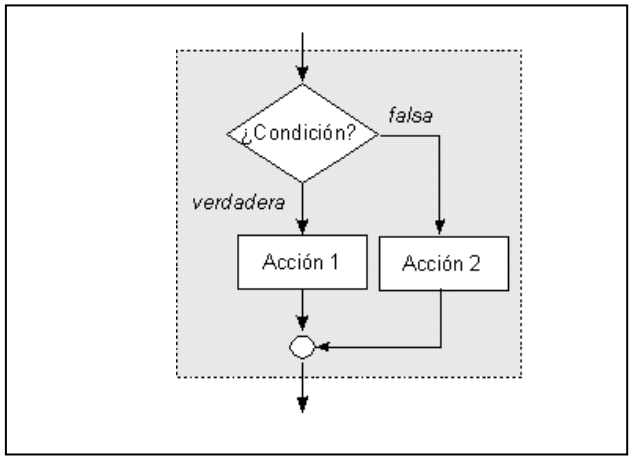


Figura 2.7: Diagrama de flujo de la estructura alternativa doble.

Pseudocódigo 2.6: Pseudocódigo para la estructura alternativa doble en español.

```
inicio
    si condición entonces
        Acción 1
    sino
        Acción 2
    fin Si
fin
```

Pseudocódigo 2.7: Pseudocódigo para la estructura alternativa doble en inglés.

```
begin
    if condición then
        Acción 1
    else
        Acción 2
    end
end
```

Ejemplo 2.5.2 Elaborar un algoritmo que resuelva una ecuación de primer grado *Ax + B = 0*, dado los valores de los coeficientes *A* y *B*.

Las posibles soluciones son:

- **Si** A ≠ 0 , **entonces** $x = \frac{-B}{A}$
- **Si** A == 0 y B ≠ 0, **entonces** "*Solución imposible*"
- **Si** A == 0 y B == 0, **entonces** "*Solución indeterminada*"

El algoritmo en forma de pseudocódigo se muestra en 2.8.

Pseudocódigo 2.8: Pseudocódigo para la solución de la ecuación de primer grado.

```
inicio
    /* Solución de la ecuación de primer grado Ax + B = 0    */
    Leer A y B
    si A ≠ 0 entonces
        A ← B/A
        Imprimir x
    sino
        si B ≠ 0 entonces
            Imprimir "Solución imposible"
        sino
            Imprimir "Solución indeterminada"
        fin Si
    fin Si
fin
```

Alternativas múltiples (según sea, caso/switch, case)

La estructura de decisión múltiple evaluará una expresión que tomará *n* valores distintos. Según lo que se elija en la condición se realizará una de las *n* acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los *n* posibles.
La representación gráfica de la estructura de decisión múltiple se muestra en la figura 2.8.

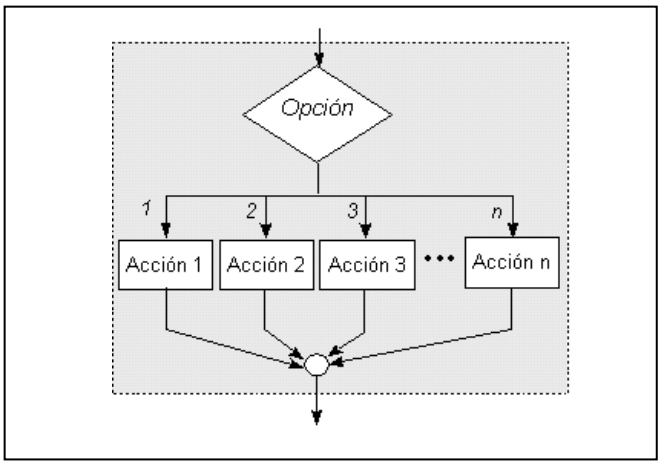


Figura 2.8: Diagrama de flujo de la estructura alternativa múltiple.

La estructura de decisión múltiple se puede representar en varias formas, una de ellas se muestra en el pseudocódigo 2.9 en español, y 2.10 en ingles.

Pseudocódigo 2.9: Pseudocódigo para la la estructura alternativa multiple en español.

```
inicio
    según sea condición hacer
        caso 1:
         | Acción 1
        caso 2:
         | Acción 2
        caso 3:
         | Acción 3
        ⋮
        caso n:
         | Acción n
        otros  Acción otro caso
    fin según
fin
```

Pseudocódigo 2.10: Pseudocódigo para la la estructura alternativa multiple en inglés.

```
begin
    switch condición do
        case 1:
         | Acción 1
        case 2:
         | Acción 2
        case 3:
         | Acción 3
        ⋮
        case n:
         | Acción n
        otherwise Acción otro caso
    end
end
```

Ejemplo 2.5.3 Diseñar un algoritmo que escriba el nombre de los días de la semana en función del valor de una variable *Día* introducida por el teclado.

Los días de la semana son 7, por tanto, los valores posibles de *Día* serán 1,2,···, 7, en caso de que *Día* tome un valor fuera del rango, se deberá producir un mensaje de error, advirtiendo la situación anormal.

Pseudocódigo 2.11: Pseudocódigo que imprime los días de la semana, a partir de su número.

```
inicio
    Leer Dia
    según sea Dia hacer
        caso 1:
         | Imprimir "Lunes"
        caso 2:
         | Imprimir "Martes"
        caso 3:
         | Imprimir "Miércoles"
        caso 4:
         | Imprimir "Jueves"
        caso 5:
         | Imprimir "Viernes"
        caso 6:
         | Imprimir "Sábado"
        caso 7:
         | Imprimir "Domingo"
        otros Imprimir "ERROR"
    fin según
fin
```

2.5.3 Estructura repetitiva

Las instrucciones que repiten una secuencia de instrucciones un número determinado de veces se denomina *bucle* o *lazo* (en inglés *loop*); se nombra *iteración* al hecho de repetir la ejecución de una secuencia de acciones. Existen tres tipos de estructuras repetitivas:

- *mientras*
- *repetir*
- *desde*

Estructura repetitiva mientras

La estructura repetitiva **mientras** (en inglés *while*: "*hacer mientras"*) es aquella en que el cuerpo del bucle se repite mientras se cumple una condición determinada. La representación gráfica se muestra en la figura 2.9.

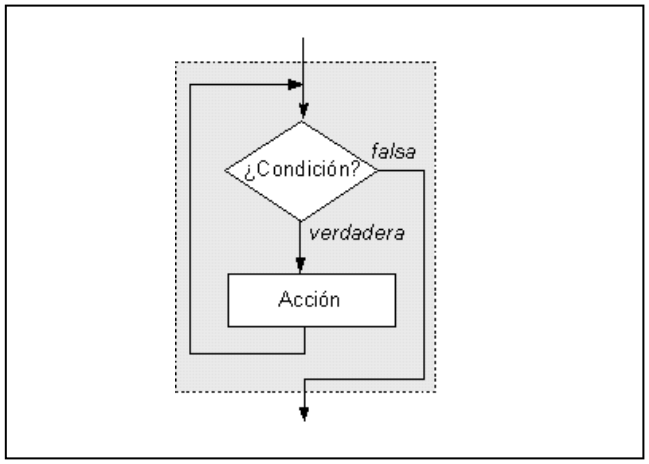


Figura 2.9: Diagrama de flujo de la estructura repetitiva mientras.

Cuando se ejecuta la instrucción **mientras**, la primera cosa que sucede es que se evaluará la condición (una expresión booleana). Si la condición es *falsa*, ninguna acción es ejecutada y el programa prosigue en la siguiente instrucción después del bucle. Si la expresión booleana es *verdadera*, entonces se ejecuta el cuerpo del bucle (*Acción*), después se evaluará nuevamente la condición. Este proceso se repite una y otra vez mientras la condición sea verdadera.

El pseudocódigo 2.12 y 2.13 representan la estructura repetitiva *mientras* para el español e inglés respectivamente.

Pseudocódigo 2.12: Pseudocódigo para la estructura repetitiva mientras en español

```
inicio
    mientras condición hacer
        Acción 1
        Acción 2
        Acción 3
        ⋮
        Acción n
    fin mientras
fin
```

Pseudocódigo 2.13: Pseudocódigo para la estructura repetitiva mientras en inglés

```
begin
    while condición do
        Acción 1
        Acción 2
        Acción 3
        ⋮
        Acción n
    end
end
```

Ejemplo 2.5.4 Diseñar un algoritmo que calcule el promedio de *N* números dados.

El promedio se obtiene utilizando la siguiente fórmula:

$$promedio = \frac{\sum_{i=1}^{N} dato_i}{N}$$

Para realizar el algoritmo es necesario sumar todos los datos, y posteriormente, dividirlos por el número de elementos. La suma se realiza agregando el *dato1*, el *dato2* y así sucesivamente hasta *datoN*, a una variable que acumula los datos en forma parcial. Si se usa un contador *i* que comienza desde *1* y termina hasta *N* y una variable llamada *Suma*, se puede realizar la sumatoria. Finalmente, el resultado obtenido, es dividido por la cantidad de datos leídos (*N*), su resultado es almacenado en la variable *Promedio*. El pseudocódigo 2.14 muestra el algoritmo para obtener la media de *N* números dados.

Pseudocódigo 2.14: Pseudocódigo para obtener el promedio

```
inicio
    Leer N
    i ← 1
    Suma ← 0
    mientras i ≤ N hacer
        Leer dato
        Suma ← Suma + dato
        i ← i + 1
    fin mientras
    Promedio ← Suma / N
    Imprime Promedio
fin
```

Estructura repetir

Existen muchas situaciones en las que se desea que el bucle se ejecute al menos una vez antes de comprobar la condición de repetición. En la estructura **mientras** si el valor de la expresión booleana es inicialmente falso, el cuerpo del bucle no se ejecutará; por ello se necesitan otros tipos de estructuras repetitivas.
Las estructura **repetir** (*repeat* en inglés) se ejecuta hasta que se cumpla una condición determinada que se comprobará al final del bucle.
El bucle **repetir-hasta-que** se repite mientras el valor de la expresión booleana de la condición sea falsa, justo lo opuesto a la sentencia **mientras**.
La figura 2.10 muestra el diagrama de flujo para la estructura **repetir** y los algoritmo 2.15 y 2.16 muestra el pseudocódigo en español e inglés respectiva- mente.

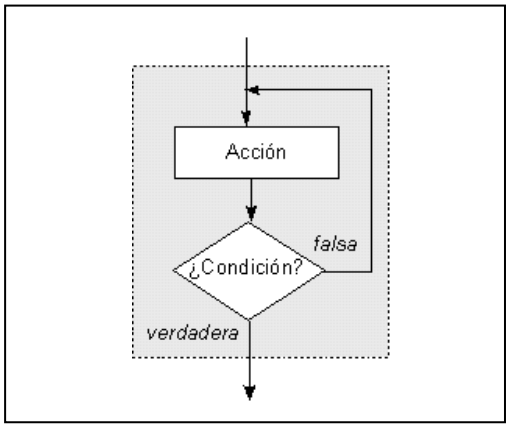


Figura 2.10: Diagrama de flujo de la estructura repetir.

Pseudocódigo 2.15: Pseudocódigo para la estructura repetir en español.

```
inicio
    repetir
        Acción 1
        Acción 2
        Acción 3
        ⋮
        Acción n
    hasta que Condición
fin
```

Pseudocódigo 2.16: Pseudocódigo para la estructura repetir en inglés.

```
begin
    repeat
        Acción 1
        Acción 2
        Acción 3
        ⋮
        Acción n
    until Condición
end
```

Con la estructura **repetir**, el cuerpo del bucle *se ejecuta siempre al menos una vez*. Cuando una instrucción **repetir** se ejecuta, la primera cosa que sucede, es la ejecución del bucle y posteriormente, se evalúa la expresión booleana resultante de la condición. Si esta última se evalúa como *falsa*, el cuerpo del bucle se repite y nuevamente la expresión booleana es evaluada; si por el contrario, la condición es *verdadera*, el bucle termina y el programa sigue en la siguiente instrucción de **hasta-que**.

NOTA: Hay que tener cuidado, debido a que en el lenguaje C, esta estructura se realiza con la instrucción **do-while**, que al evaluar la condición, se repite si es verdadera y termina si es falsa. Es decir, la estructura *repetir-hasta-que* se puede realizar en lenguaje C con la instrucción *do-while*, utilizando la condición en forma negada.

*Diferencias de las estructuras **mientras** y **repetir**.*

- La estructura **mientras** termina cuando la condición es falsa, mientras que **repetir** termina cuando la condición es verdadera.

- En la estructura **repetir** el cuerpo del bucle se ejecuta al menos una vez; por el contrario, **mientras** es más general y permite la posibilidad de que el bucle pueda no ser ejecutado. Para usar la estructura **repetir** se debe asegurar que el cuerpo del bucle (bajo cualquier circunstancia), se repetirá al menos una vez.

Ejemplo 2.5.5 Desarrollar un algoritmo para calcular el factorial de un número dado *n*. El factorial del un número, está dado por:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

El algoritmo se muestra en el pseudocódigo 2.17

Pseudocódigo 2.17: Pseudocódigo par obtener el factorial del un número n.

```
inicio
    Leer n
    i ← 1
    fact ← 1
    repetir
        fact ← fact * i
        i ← i + 1
    hasta que i > n
    Imprime fact
fin
```

Estructura desde/para (for)

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones de un bucle. En estos casos, el número de iteraciones es fijo, se debe usar la estructura **desde/para** (**for** en inglés).

La estructura **desde** ejecuta las acciones del cuerpo del bucle un número específico de veces y de modo automático, controla el número de iteraciones o pasos a través del cuerpo del bucle.
El pseudocódigo utilizado para este tipo de estructura se puede escribir de dos formas: el primer modelo se muestra en 2.18, que representa la forma **desde hasta hacer**; el segundo modelo mostrado en 2.19, representa la forma **para desde hasta hacer**. Finalmente, el pseudocódigo 2.20 muestra la representación en inglés.

Pseudocódigo 2.18: Pseudocódigo para la estructura **desde** (modelo 1).

```
inicio
    desde var = ini hasta fin hacer
        Acción 1
        Acción 2
        ⋮
        Acción n
    fin desde
fin
```

Pseudocódigo 2.19: Pseudocódigo para la estructura **desde** (modelo 2).

```
inicio
    para var de ini hasta fin hacer
        Acción 1
        Acción 2
        ⋮
        Acción n
    fin para
fin
```

Pseudocódigo 2.20: Pseudocódigo para la estructura **desde** en inglés.

```
begin
    for var = ini to fin do
        Acción 1
        Acción 2
        ⋮
        Acción n
    end
end
```

El bucle **desde** (**for**) se representa con los símbolos de proceso y de decisión mediante un contador, como muestra la figura 2.11. También se puede representar utilizando otro símbolo, como se ilustra en la figura 2.12

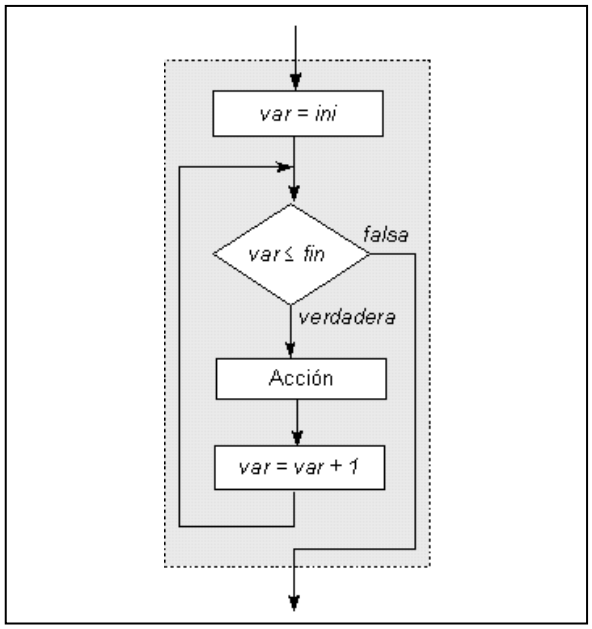


Figura 2.11: Diagrama de flujo para la estructura desde. Modo 1

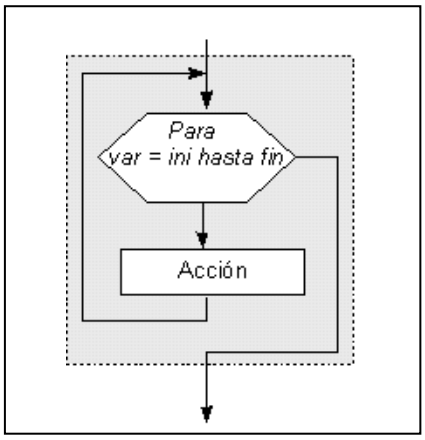


Figura 2.12: Diagrama de flujo para la estructura desde. Modo 2

La estructura **desde** comienza con un valor inicial de una variable índice (*var*, en el pseudocódigo y diagrama de flujo), las acciones especificadas se ejecutan a menos que el valor inicial sea mayor que el valor final. La variable índice (*var*) se incrementa en uno y si este nuevo valor no excede el final, se ejecuta de nuevo las acciones. Por consiguiente, las acciones específicas en el bucle se ejecutan para cada valor de la variable índice (*var*), desde el valor inicial hasta el valor final, con incremento de uno en uno.

El incremento de la variable índice siempre es 1 si no se indica expresamente lo contrario. Dependiendo del tipo de lenguaje, es posible que el incremento sea distinto de uno, positivo o negativo.

Si el valor inicial de la variable índice es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de acciones no se ejecutaría. De igual modo si el valor inicial es mayor que el valor final, el incremento debe ser en este caso negativo, es decir, *decremento.*

Ejemplo 2.5.6 Diseñar un algoritmo que imprima las 10 primeras potencias de un valor dado. Por ejemplo, si el número dado es 4, entonces se imprimirá:

```
4 elevado a la 1 es: 4
4 elevado a la 2 es: 16
                .
                .
                .
4 elevado a la 10 es: 1048576
```

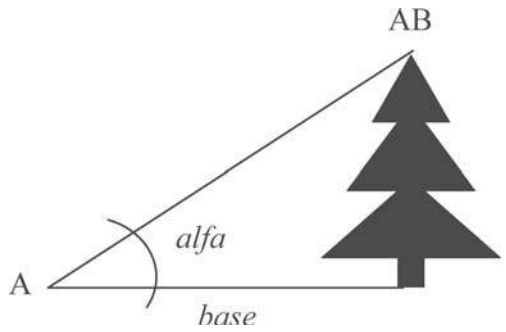
El algoritmo se muestra en el pseudocódigo 2.21

Pseudocódigo 2.21: Pseudocódigo para la impresión de potencias de un número.

```
inicio
    Leer num
    potencia ← 1
    para i de 1 hasta 10 hacer
        potencia ← potencia * num
        Imprimir num elevado a la i es igual a potencia
    fin para
fin
```

2.6 Ejercicios propuestos

Ejercicio 2.6.1 Diseñar un algoritmo que convierta de millas a kilómetros.
Ejercicio 2.6.2 Diseñar un algoritmo que calcule el punto medio entre dos puntos de un eje coordenado.
Ejercicio 2.6.3 Diseñar un algoritmo que calcule la altura de un árbol, conociendo la distancia del árbol y un punto A (*base*), así como el ángulo formado por los segmentos de recta AB (*alfa*).



Ejercicio 2.6.4 Elaborar un programa que calcule el área de un triángulo cualquiera, mediante la siguiente expresión

$$Area = \sqrt{s(s-a)(s-b)(s-c)}$$

Donde. $s = \frac{a+b+c}{2}$

Ejercicio 2.6.5 Elaborar un algoritmo que determine las raíces reales de una ecuación de segundo grado e imprima un mensaje cuando son raíces complejas.
Ejercicio 2.6.6 Elaborar un algoritmo que lea el último dígito de la placa de un vehículo y determine el día que no circula, así como los meses que le toca su verificación.

Ejercicio 2.6.7 Elaborar un programa que realice el siguiente menú:

1. Obtener el perímetro de una circunferencia
2. Obtener el área de una circunferencia
3. Salir

¿Cuál es tú opción? _

Ejercicio 2.6.8 Elaborar un algoritmo que imprima 1000 veces "¡*Viva México!* ".

Ejercicio 2.6.9 Elaborar un algoritmo que obtenga las tablas de multiplicar (emplear ciclos).

Ejercicio 2.6.10 Elaborar un algoritmo que calcule el promedio de las calificaciones de un estudiante.

Ejercicio 2.6.11 Elaborar un algoritmo que simule una caja registradora dando como entrada el costo del artículo y su cantidad.

Ejercicio 2.6.12 Elaborar un algoritmo que imprima el siguiente patrón, dando como entrada el número de líneas.

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

Ejercicio 2.6.13 Elaborar un algoritmo que calcule la potencia entera de un número cualquiera.

Ejercicio 2.6.14 Elaborar un algoritmo que aproxime el valor de π, usar la siguiente ecuación:

$$\pi = 2 \cdot \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdots$$

Ejercicio 2.6.15 Elaborar un algoritmo que lea *N* números esteros y determine la cantidad de números pares y la cantidad de números impares.

Bibliografía

[Joyanes 90] Joyanes Aguilar, ”Fundamentos de programación”, McGraw Hill 1ra. ed. México 1990.
[CodeWarrior 95] CodeWarrior, ”Principles of Programming”, Metrowerks Inc., 1995.
[Meter Bradley 94] Meter Aiken, Bradley Jones, Aprendiendo C en 21 días Prentice Hall Primera Ed., México 1994.
[Schildt 88] Herbert Schildt , C Manual de Referencia, McGraw Hill Primera Ed. España 1988.
[Hernán 03] Hernán Ruiz Programación C Manuales USERS, Marcelo MP Edi- ciones Primera Ed., Argentina 2003.
[Deitel 00] Deitel, H., Deitel P. J. , Cómo programar con C/C++ , Prentice-Hall Tercera Ed., México 2000.
[Polya 45] George Polya, How to Solve It, Princeton, New Jersey: Pinceton University Press, 1945.
[Brian Kernighan] C Brian W. Kernighan, Dennos M. Ritchie, El lenguaje de programación, Prentice Hall, 2a. Ed.

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Capítulo 3
Elaboración de programas en lenguaje C.

Tabla de contenido:

3.1 Elementos de programación en lenguaje C

Todo lenguaje es un medio de comunicación, en particular, los lenguajes de programación son un medio para comunicarnos con la computadora, pero dicho medio debe de tener reglas tales que permitan establecer una comunicación sin ambigüedades. Inicialmente, consideremos la siguiente caracterización de lo que es un lenguaje de programación:

Es un lenguaje con estrictas reglas gramaticales, símbolos y palabras reservadas, usadas para escribir un programa de computadora.

De lo señalado anteriormente, se desprende que para elaborar un programa en lenguaje C requerimos conocer los símbolos permitidos por dicho lenguaje y las reglas para elaborar las palabras que constituirán las órdenes que se le darán a la computadora. Sin embargo, la estructura de dichas órdenes deberá cumplir con una serie de reglas definidas por el mismo lenguaje. Esto es, existen reglas sintácticas y semánticas que debemos considerar para elaborar simples o sofisticados programas. A continuación empezaremos a conocer los conceptos y algunas reglas asociadas para el desarrollo de tales programas escritos en lenguaje C.

Estructura de un programa.

Comenzamos el aprendizaje del lenguaje C adquiriendo una noción de la estructura que tiene un programa en dicho lenguaje, lo cual tiene que ver con algunos conceptos fundamentales que definen la elaboración de programas, en particular, para el lenguaje C. La programación a través del lenguaje C, tiene como base fundamental el concepto de **función**. Una posible aproximación a dicho concepto puede ser la siguiente:

Una función es una colección de enunciados que realizan una tarea específica, la cual puede invocar otra función pero no puede tener definida otra función en ella misma.

Partiendo del concepto de función, veremos que en el siguiente ejemplo aparece una función principal llamada **main** que cumple ciertas tareas y no regresa nada. Además podemos advertir un cierto esquema para analizar la estructura de un programa simple en lenguaje C.

En dicho programa se halla la suma de los primeros n números, y por ello, encontramos en la primer línea una directiva de preprocesador, la cual le informa al preprocesador que se debe incluir un archivo de cabecera, el cual a su vez será utilizado por el compilador para verificar el adecuado uso de las funciones de

entrada y salida **printf**() y **scanf**(). Cabe aclarar que el preprocesador es un programa que modifica el programa fuente antes de que lo procese el compilador.

Después de la directiva del preprocesador, tenemos la definición de la función **main**() que debe aparecer para que un programa pueda ser ejecutado. Dentro de la función **main**() encontramos, cinco órdenes dadas a la computadora, las primeras dos líneas corresponde a la declaración de variables de memoria, inmediatamente se tiene la llamada a una función de lectura de datos, posteriormente se encuentra un comando repetitivo, observe que este comando corresponde a una estructura de control, pero sigue siendo una orden compuesta de otras dos órdenes, y finalmente la llamada de un función que desplegará los resultados en el monitor. Por lo anterior, podemos sintetizar y decir, de modo muy general, que en un archivo con un programa en C, se tienen directivas de preprocesador y definiciones de funciones, a su vez dentro de las funciones tendremos generalmente, declaraciones de variables y enunciados u órdenes para la computadora.

```
#include <stdio.h>

main() {

  int contador =1, n;
  int suma     =  0;

  scanf("%d", &n);
  while (contador <= n ){
      suma     = suma+contador;
      contador = contador + 1;
   }

  printf("\n Suma final = %d\n ", suma);

       }
```

Directivas del preprocesador

Función Principal

El compilador, el cual es un programa muy sofisticado, al aplicarse a un programa en código fuente C, principalmente, verifica que esté escrito de acuerdo a las reglas del lenguaje, a nivel lexicográfico, sintáctico y semántico. Si lo está, se genera el código de máquina equivalente llamado código objeto, el cual será ligado a las bibliotecas del lenguaje C o del usuario, por un programa llamado ligador, generando finalmente un programa ejecutable. Si el compilador encontrará un error, lanzaría un mensaje y no crearía ningún código objeto. A diferencia de un compilador, un intérprete es un programa que en el momento de leer una instrucción de código fuente, la verifica y si está correctamente escrita, genera el código de máquina correspondiente y lo ejecuta.

3.2 Variables y Constantes.

Uno de los elementos principales en la mayoría, sino es que en todos los programas, son las variables, a partir de éstas se procesa la información en un programa. A continuación estudiaremos los elementos que componen una variable y las reglas asociadas a éstas.

Una **variable** es un área de almacenamiento de memoria que tiene un nombre conocido como **identificador**, a través del cual podemos hacer referencia a dicha área de memoria, la que a su vez, será de un **tipo** determinado, esto es, en el área de memoria se almacenará un grupo o conjunto de valores: números enteros, o números de simple precisión, o números de doble precisión, o caracteres, etc. (Ver 3.4)

Reglas de los identificadores

Los identificadores de una variable tienen las siguientes reglas:

1. Los caracteres que forman un identificador pueden ser dígitos, letras del alfabeto inglés y el carácter de subrayado _ .

2. El primer carácter de un identificador debe ser una letra o el carácter de subrayado _ .

3. Se diferencian las mayúsculas de las minúsculas.

4. No pueden ser alguna de las palabras reservadas.

5. El número de caracteres que puede tener un identificador es, a lo más, de 31 caracteres. (Puede variar para algunos compiladores)

Los caracteres que permite utilizar el lenguaje C en la construcción de los identificadores se presentan a continuación:

1. Las letras mayúsculas y minúsculas del alfabeto inglés:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

2. Los dígitos decimales

```
0 1 2 3 4 5 6 7 8  9
```

3. El carácter de subrayado

```
_
```

Ejemplos:

Identificadores válidos:	Identificadores no válidos:
a) indiceActual b) A111 c) Humano312456789033333333333 d) int3 e) _67945	a) indiceActual# b) año c) índice d) 500veces e) float Indique porqué son inválidos

Convenciones en los identificadores para un estilo de programación.

Se recomienda el uso de las siguientes convenciones para crear un identificador:

1. Se usarán minúsculas.

2. Si se requiere de más de una palabra, la segunda deberá iniciar con mayúscula.

3. Use nombres descriptivos que indiquen y den una idea de lo que la variable almacena.

Ejemplos de Identificadores elaborados con dicho estilo:

1) indiceActual
2) sumaTotal
3) matriculaEstudiante.
4) indiceTemporalUnico

A continuación se presenta una lista de palabras reservadas, es decir, que tiene un significado especial para el lenguaje C, las cuales no podrán ser utilizadas como identificadores:

Palabras reservadas

auto	else	Int	switch
break	entry	Long	typedef

case	enum	Register	union
char	extern	Return	unsigned
continue	float	Short	until
default	for	Syzeof	void
do	goto	Static	while
double	if	Struct	

Constantes

Se pueden definir identificadores, que tengan asociados valores constantes, o reservar un área de memoria que tenga un valor constante., esto es:

> *Una **constante** es un valor que una vez considerado por el compilador no cambia a lo largo de la ejecución de un programa.*

Existen dos maneras de indicar una constante en C, o mediante una directiva de preprocesador **define** o anteponiendo a la declaración de una variable la palabra reservada const.

Por ejemplo:

```
#define PI  3.1416
```

Un modo equivalente, pero no idéntico, de declarar algo parecido es el que se muestra a continuación:

```
const  float PI = 3.1416;
```

3.3 Tipos de datos: `char, int, float, double.`

En el programa presentado con anterioridad, se le ordena a la computadora reservar el espacio de memoria para dos variables llamadas contador y suma cuyos tipos de datos son int y al iniciar la ejecución del programa, dichas áreas de memoria contendrán el valor de 1 y 0 respectivamente. A tales órdenes se les conoce como definición o declaración de variables, aunque entre estos términos hay sutiles diferencias.

Ejemplo de definición de variables:

```
int contador =  1;
int suma     =  0;
```

Observe el orden que se requiere para colocar a los elementos de una declaración:

```
tipo  identificador  [=valor de inicio]
```

Los corchetes denotan que dicho elemento es opcional.

En general, los tipos de datos primitivos permitidos en el lenguaje C, son los que se muestran a continuación. Un buen ejercicio es realizar el cálculo del rango de valores de los números enteros en complemento a dos, dado el tamaño del tipo int . Pero recuerde que para un tipo de datos, el tamaño puede variar y por lo tanto, también el rango, verifique esta situación en un procesador específico.

Tipo	**Valor**	**Tamaño**
int	Número entero base 10	16 – 32 bits
float	Número con punto decimal	16 – 32 bits
double	Número con punto decimal	Doble de float
char	Caracter	8 bits

A continuación presentamos algunos ejemplos de valores que pueden ser almacenados en áreas definidas para un cierto tipo de datos.

```
a) 1522    Entero decimal que se almacena en una variable
int.
b) 01341  Entero octal que se almacena en una variable int.
c) 0x999AB Entero hexadecimal que se almacena en una variable
int.
d) 0X12FD Entero hexadecimal que se almacena en una variable
int.
e) 19.777 Real que se almacena en una variable double.
f) -1667L Entero long que se almacena en una variable long
int.
g) 1.8e5 Real que se almacena en una variable double.
h) 2.3E-2 Real que se almacena en una variable double.
i) 662F Real que se almacena en una variable float.
```

Valores de tipo `char`

Son valores enteros de 0 a 127 ó de 0 a 255 pero que son interpretados como caracteres.

Ejemplos:

a) `'a'`
b) `'\n'`
c) `'B'`
d) `':'`

Constantes de tipo cadena

Son grupos de caracteres que se consideran como paquetes únicos. Observe que las constantes de carácter se escriben entre comillas simples y las constantes de cadena entre comillas dobles.

Ejemplos:

a) `"hola"`
b) `"que tal"`
c) `"\n\t\tTabla de valores numéricos"`

Modificadores de tipos de datos: `unsigned`, `short`, `long`.

El tipo de datos **int** utiliza a su vez algunos modificadores, como **short** que al aplicarlo a **int** solicita reservar el tamaño mas pequeño posible de memoria para una variable entera y **long** que aplicado a **int** solicita reservar el tamaño más grande posible para una variable entera. Además, si deseamos trabajar con solo números enteros positivos y deseamos que el bit más significativo no sea negativo podemos aplicar el modificador **unsigned** al tipo **int**.

3.4 Expresiones

Otro de los elementos de mayor nivel de construcción respecto a las variables son las expresiones.

> *Una expresión es una combinación de operadores y operandos que al evaluarse producen valor numérico.*

Los operandos pueden ser constantes literales, variables o invocaciones de funciones que regresen un valor numérico.

Los operadores se muestran a continuación.

Operadores

A continuación presentaremos algunos de los operadores permitidos en el lenguaje C:

Tipo	Operadores
Aritméticos	*, /, %, +, -
Relacionales	>, >=, <, <=, ==, !=
Lógicos	&&, || , !
Manejo de bits	&, |, <<, >>
Asignación	=, op=, ++ --
Condicional	? :
Acceso de datos	*, &, [], . , ->

Operadores relacionales y lógicos

Haremos hincapié en algunos tipos de expresiones que son muy importantes para enunciados condicionales y cíclicos que corresponden a diferentes estructuras de control. En tales estructuras de control existen condiciones en las que se usan expresiones lógicas que pueden incluir:

Operadores Relacionales

`<` `<=` `>` `>=` `==` `!=`

Operadores lógicos

`!` `&&` `&` `||` `|` `^`

Las expresiones de tipo relacional tienen un esquema binario que se forma con dos expresiones unidas por un operador.

Ejemplos:

	Expresión A	**Operador**	**Expresión B**
1)	`temperatura`	`>`	`humedad`
2)	`B*B-4.0*A*C`	`>`	`0.0`
3)	`numero`	`==`	`35`
4)	`inicio`	`!=`	`'Q'`

Ejemplo del cálculo del valor de operadores relacionales

Consideremos las siguientes declaraciones:

```
int  x, y ;
x = 4;
y = 6;
```

el resultado de las siguientes expresiones sería:

EXPRESION	VALOR
`x < y`	`1 (verdadero)`
`x + 2 < y`	`0 (falso)`
`x != y`	`1 (verdadero)`
`x + 3 >= y`	`1 (verdadero)`
`y == x`	`0 (falso)`

Recuerde que el cálculo de las expresiones toma en cuenta el orden de prioridades definido por el lenguaje.

Operadores lógicos básicos

El uso de operadores lógicos es tan frecuente que debemos recordar en que consisten las operaciones lógicas.

OPERACIÓN	EXPRESION LÓGICA	SIGNIFICA	DESCRIPCION
Negación	**! p**	**NO p**	**! p es F si p es V** **! p es V si p es F**
Conjunción	**p && q**	**p Y q**	**p && q es V si ambas p y q son V.** **En otro caso es F.**
Disyunción	**p || q**	**p OR q**	**p || q es V si o p o q o ambas son V.** **En otro caso es F.**

Expresiones abreviadas (Short circuit)

El lenguaje C usa evaluación corto circuito de expresiones lógicas con operadores:

&&	||

Las expresiones lógicas son evaluadas de izquierda a derecha y la evaluación se detiene tan pronto como el valor final de verdad puede ser determinado.

Ejemplos del cálculo del valor de operadores lógicos

Ejemplo 1

```
int   edad, peso;
edad = 25;
peso = 70;
```

EXPRESION		
(edad > 50)	&&	(peso > 60)
0		0

El resultado final de la expresión completa anterior es 0, sin embargo, al mecanismo de evaluación le basta con evaluar la primera expresión, pues al ser ésta 0, concluye que toda la expresión, la cual es una conjunción tiene el valor de 0, es decir, como el primer miembro de la expresión es falso, entonces se desprende que la expresión completa es falsa, sin necesidad de evaluar el segundo miembro.

Ejemplo 2

```
int   edad, peso;
edad = 25;
peso = 70;
```

EXPRESION		
(peso > 60)	||	(edad > 40)
Verdadero		

Análogamente al primer caso, la evaluación de la expresión no se realiza en su totalidad, pues ya que se sabe que para que el resultado de una disyunción sea verdadero, basta con que uno de los miembros sea verdadero. En el caso anterior como el primer miembro de la expresión es verdadero, entonces se desprende que la expresión completa es verdadera sin necesidad de evaluar el segundo miembro.

Operadores de asignación

El operador de asignación tiene un miembro izquierdo y un miembro derecho, el miembro izquierdo es una variable que recibirá el resultado de la expresión que se localizará en el miembro derecho. En particular, cuando tenemos un operador de asignación donde a la izquierda del signo de asignación existe un operador aritmético es una manera abreviada de incluir la variable de la izquierda en la expresión de la derecha, esto es:

```
i += 3    equivale a        i = i+3
```

Se reconoce como operador de asignación el `++` y `--`. Suponga las siguientes declaraciones:

```
int a= 0;
int b= 10;
```

La expresión siguiente significa que primero se asigna b y luego se incrementa:

```
a = b++;
```

La variable a termina con el valor de 10 y la variable b con el valor de 11.

Si en vez de la expresión anterior tuviéramos la siguiente:

```
a = ++b;
```

La variable a termina con el valor de 11 y la variable b con el valor de 11.

Prioridad de operadores

En el cálculo del valor de una expresión es importante considerar las siguientes reglas de prioridad, las cuales toman en cuenta la tabla mostrada debajo, la prioridad de los operadores es determinada por el renglón al que pertenecen, entre mas arriba está el renglón al que pertenecen, mayor prioridad tendrán. Si los operadores a considerar están en el mismo renglón de prioridad, la expresión se evaluará tomando en cuenta los operadores de la expresión de izquierda a derecha.

Tabla de prioridades

Asociatividad	Operadores
Izquierda a derecha	() [] . ->
Derecha a izquierda	- ~ ! * & ++ -- sizeof (tipo)
Izquierda a derecha	:>
Izquierda a derecha	* / %
Izquierda a derecha	+ -
Izquierda a derecha	<< >>
Izquierda a derecha	> >= < <=
Izquierda a derecha	== !=
Izquierda a derecha	&
Izquierda a derecha	^
Izquierda a derecha	|
Izquierda a derecha	&&
Izquierda a derecha	||
Derecha a izquierda	?:
Derecha a izquierda	= *= /= %= += -= <<= >>= &= |= ^=

Uso de la tabla de prioridades

A continuación presentamos algunos ejemplos de expresiones y sus resultados después de evaluarse, para ello consideremos las siguientes asignaciones:

```
     int contador=1 ;
     int indice = 1000;

          Expresión                    Cálculo resultante
     a)    1                                1
     b)  'A'                                'A' (o su código
ASCII)
     c)  2+3                                5
     d)  contador + 1                       2
     e)  indice < 100                       0
```

En el cálculo de expresiones debe considerar que si una condición lógica se evalúa como cero entonces se considera FALSA, si su evaluación es diferente de cero, se considera VERDADERA, pero todos los resultados de una expresión lógica serán números enteros.

3.5 Enunciados

El siguiente elemento importante en la construcción de un programa, y de mayor complejidad respecto a las expresiones, es el enunciado cuyos sinónimos son sentencia, orden, comando ó instrucción:

> *El enunciado es la unidad mínima ejecutable de un programa en C, y controla el flujo u orden de ejecución. Un enunciado puede contener una palabra clave (***if-else***,* **while***,* **do-while***,* **for***,* etc.*), expresiones, declaraciones o llamadas a funciones*.

Reglas:

1. Todo **enunciado simple** termina con punto y coma.

2. Dos o más enunciados pueden estar en una misma línea, separados por punto y coma.

3. Todo **enunciado nulo** consta de un punto y coma.

Enunciado compuesto o bloque:

Si se tienen varios enunciados se puede formar una sentencia compuesta encerrándolos dentro de llaves y ésta se llamará *bloque*. Un bloque puede componerse de otros bloques. Después de un bloque no se requiere un punto y coma. Los enunciados pueden definir una estructura de control, a continuación se presentan las estructuras de control y su equivalente como enunciados en el lenguaje C.

En el lenguaje C existen enunciados especiales que corresponden a estructuras de control como la condicional o como la iteración, esto es, son enunciados u órdenes que se deben de realizar siguiendo los esquemas definidos por estructuras de control.

Estructura de control en seudocódigo	**Enunciados**

si (C)
entonces
e1
sino
e2
	If (C)
	e1;
	else
	e2;
	}

mientras (C) {
e
}
	while (C) {
	e;
	}

Haz e mientras (C)	do{ e; } while (C);
Repite e hasta (C)	do{ e; } (! C);

Ejecuta i = 1, n
e
	for (i = 1 ; i < n; i++) {
	e;
	}

3.6 Funciones de entrada / salida.

Cuando se inicia a programar en lenguaje C generalmente será necesario desplegar tanto resultados parciales o finales de un programa y para ello, se requerirá utilizar funciones ya definidas en el lenguaje que desplieguen datos en pantalla. Así mismo, se requerirá introducir datos a un programa en tiempo de ejecución y para ello, el lenguaje también cuenta con funciones predefinidas. En particular usaremos las funciones **printf()** y **scanf()** para desplegar y recibir datos en un programa, a reserva de aprender, mas adelante, el uso de otras funciones que con que cuenta el lenguaje C para realizar procesos similares. El teclado y el monitor son conocidos como la entrada y la salida estándar en el lenguaje C.

Función `printf()`

El comando para desplegar letreros en pantalla es:

```
printf(letrero [,var1, var2,...,varn ])
```

Donde letrero es un texto entre comillas dobles. Opcionalmente, puede incluir dentro de dicho texto, secuencias de escape, por ejemplo:

\n	Salto de línea en el letrero.
\t	Salto de tabulador en la misma línea.
\b	Retroceso
\f	Avance de página.

Las secuencias de escape son secuencias de caracteres que inician con la barra invertida y que representan caracteres de control.

En el letrero se pueden incluir especificadores de conversión, indicando que dentro del letrero se insertarán diferentes tipos de valores, por ejemplo:

Especificador de conversión	Acción
%d	Insertará un número entero en base diez, de tipo **int**.
%f	Insertará un número con punto decimal, de tipo **float**.
%ld	Insertará un número entero en base diez, de tipo **long**.
%lf	Insertará un número con punto decimal, de tipo **double**.
%c	Insertará un carácter, de tipo **char**.
%s	Insertará una cadena, o lo que es lo mismo un arreglo de caracteres **char**[], lo que sería otro texto.

Dichos valores serán tomados de las variables var_1, var_2, ..., var_n. Por ello, el número de símbolos debe corresponder tanto al número de variables, y estas deben ser del mismo tipo que se indica en el símbolo.

Ejemplos**:**

```
int a =0;
float g=0.0;
char b= 'a';
char s[20]="Cadena de texto";

printf("\n\tEste mensaje tiene saltos de línea y
tabulador\n");
printf("\n\tVariable entera= %d y de carácter=%c \n", a,
b);
printf("\n\tVariable float= %f y de cadena=%s \n", g,
s);
```

Función scanf()

El comando para leer datos del teclado es:

```
scanf(letrero, &var1, &var2,...,&varn)
```

Donde letrero es un texto entre comillas dobles que debe incluir símbolos %, indicando que espera que el usuario inserte diferentes tipos de valores, por ejemplo:

%d	Esperará un número entero en base diez, de tipo **int**.
%f	Esperará un número con punto decimal, de tipo **float**.

%ld	Esperará un número entero en base diez, de tipo **long**.
%lf	Esperará un número con punto decimal, de tipo **double**.
%c	Esperará un carácter, de tipo **char**.
%s	Esperará una cadena, o lo que es lo mismo un arreglo de caracteres **char**[], lo que sería otro texto.
[^\n]	Esperará una cadena que abarque los caracteres que introdujo el usuario antes de dar **enter**.

Dichos valores serán almacenados en las variables `var1, var2, ..., varn`. Por ello, el número de símbolos debe corresponder tanto al número de variables, y estas deben ser del mismo tipo que se indica en el especificador de conversión.

Ejemplos:

```
int a =0;
float g=0.0;
char b= 'a';
char s[20]="Cadena de texto";

scanf("%d %c", &a, &b);
scanf("%f %s", &g, s);
```

Observe que las cadenas son arreglos de carácter y son las únicas variables que no necesitan utilizar **&**, en el comando **scanf**`()`.

El símbolo **&** aplicado al identificador de una variable, indica que se debe considerar la dirección de la variable para que la función que la recibe pueda modificar su contenido, a este paso de parámetros a una función se conoce como paso de parámetros por referencia.

Cabe mencionar que el caso de las cadenas es especial, y la lectura también puede realizarse con:

```
gets(var)
```

donde *var* es una variable de tipo cadena, por ejemplo:

```
char s[20];
```

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Capítulo 4
Estructuras de control.

Tabla de contenido:

INTRODUCCIÓN

Las estructuras de control forman una parte fundamental en cualquier lenguaje de programación de propósito general. Sin la utilización de éstas sería muy difícil crear programas que se lleven a cabo o tengan cierta lógica, ya que permiten la creación de ciclos.

Se abordarán tres tipos de estructuras de control: *secuenciales, condicionales* y *repetitivas.*

4.1 Estructura secuencial.

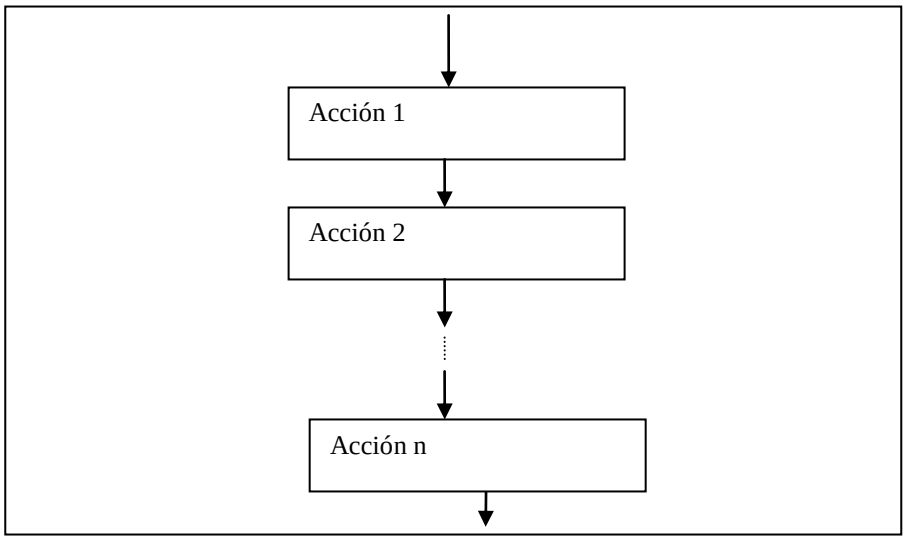
Este tipo de estructura permite la ejecución de una expresión tras otra. Es la más simple de las estructuras.

A continuación se muestran el Pseudocódigo y Diagrama de Flujo correspondientes a esta estructura.

Pseudocódigo

```
Acción 1
Acción 2
          .
          .
          .
Acción n
```

Diagrama de Flujo



EJEMPLO 4.1.1 Calcular el área de un triángulo

Este programa calcula el área de un triángulo solicitando al usuario las medidas de la base y la altura.

```
#include <stdio.h>   //Directiva de preprocesador.

main()   //Función principal.
{
  float base, altura, area;   //Declaración de variables.

//Solicitud de datos y guardado en las variables
//correspondientes.
  printf("¿Cuánto mide la base del triángulo? ");
  scanf("%f", &base);
  printf("\n ¿Cuánto mide la altura del triángulo? ");
  scanf("%f", &altura);

 //Cálculo del área.
  area = (base * altura) / 2;

//Impresión del valor del área en pantalla.
  printf("El área del triángulo es: %f\n", area);
}
```

EJEMPLO 4.1.2 Leer el tamaño de un ángulo en radianes e imprimir el seno, coseno, arco tangente, exponencial, logaritmo natural y logaritmo base 10.

Este programa lee el tamaño de un ángulo en radianes. Imprime el seno, coseno, arco tangente, exponencial, logaritmo natural y logaritmo base 10. Recuérdese que el logaritmo base 10 de un ángulo se calcula dividiendo el logaritmo natural del ángulo entre 2.3025.

```
#include <stdio.h>
#include <math.h>  //Se incluye la biblioteca matemática

main(){
  float angulo, seno, coseno, arctan, LogNat, LogBase10;

  printf("¿Cuál es la medida del ángulo en radianes? ");
  scanf("%f", &angulo);

//Cálculo de seno, coseno, arco tangente, logaritmo natural
//y logaritmo base10, utilizando funciones de la biblioteca
//math.h  ´

  seno = sin(angulo);
  coseno = cos(angulo);
  arctan = atan(angulo);
  LogNat = log(angulo);
  LogBase10 = log(angulo) / 2.3025;

  printf("El seno del ángulo %f es: %f\n", angulo, seno);
  printf("El coseno del ángulo %f es: %f\n", angulo, coseno);
  printf("El arco tangente del ángulo %f es: %f\n", angulo,
  arctan);
  printf("El logaritmo natural del ángulo %f  es: %f\n",
  angulo, LogNat);
  printf("El logaritmo base 10 del ángulo %f  es: %f\n",
  angulo, LogBase10);
}
```

4.2 ESTRUCTURA CONDICIONAL

También llamada estructura selectiva o alternativa. Es una estructura lógica que permite controlar la ejecución de aquellas acciones que requieran de cierta condición o condiciones para llevarse a cabo. Pueden presentarse tres casos: que **sólo si se cumple la condición**, se ejecuta la acción; que **si se cumple una condición o si no se cumple**, en ambos casos se ejecute alguna acción; o bien,

que **dependiendo de los valores que tome cierta variable**, se ejecuten determinadas acciones.
Así pues, las hay: **simples, dobles** y **múltiples**. Las estructuras condicionales simples pertenecen al primer caso que se mencionó en el párrafo anterior, las dobles al segundo y las múltiples al último.

Dado que es necesario trabajar con operadores lógicos, se mostrará la *tabla de verdad* que será indispensable y de gran utilidad en las estructuras condicionales y repetitivas.

Tabla de verdad

X	Y	NOT X	NOT Y	X AND Y	X OR Y
V	V	F	F	V	V
V	F	F	V	F	V
F	V	V	F	F	V
F	F	V	V	F	F

Donde *X* y *Y* son expresiones lógicas, *NOT* equivale al *no* de negación, *AND* al *y* de conjunción lógica y *OR* al *o* de disyunción lógica. Recuérdese que en lenguaje C *NOT* se representa con "!", *AND* con "&&", *OR* con "||"[4]. Los operadores de relación $\leq$ con *<=*, $\geq$ con *>=* y la comparación de igualdad con *==*.

Es importante hacer notar que el lenguaje C no trabaja con valores booleanos sino con enteros que representan booleanos.

Por ejemplo, sea `a = 4` y `b = 6`, y las expresiones `X` y `Y`:

Expresión `X`: `a <= 2*b`
Expresión `Y`: `2 + a*b == 2`

Es claro que si se sustituyen los valores de *a* y *b*, resulta que la expresión *X* es verdadera y la expresión *Y* es falsa.

Teniendo en cuenta esto y lo que se presentó en la tabla anterior, la correspondiente tabla de verdad que se obtiene al evaluar estas expresiones es:

X	Y	!X	!Y	X && Y	X || Y
a <= 2*b	2+a*b == 2	a > 2*b	2+a*b != 2	(a <= 2*b) **&&** (2+a*b == 2)	(a <= 2*b) **||** (2+a*b == 2)
4 <= 2*6	2+4*6 == 2	4 > 2*6	2+4*6 != 2	(4 <= 2*6) && (2+4*6 == 2)	(4 <= 2*6) || (2+4*6 == 2)
4 <= 12	26 == 2	4 > 12	26 != 2	(4 <= 12) && (26 == 2)	(4 <= 12) || (26 == 2)
V	F	F	V	F	V

[4] Las comillas son sólo para resaltar los símbolos. En el código no se escriben.

4.2.1 SELECCIÓN SIMPLE `if`

También llamada *estructura if.* El equivalente al condicional en español es *si,* de condición.

Permite controlar la ejecución de una acción (o acciones) cuando sólo existe una alternativa de acción. Es decir, se utiliza cuando una acción está condicionada para que se ejecute, pero si esa condición no se cumple, no pasa nada, la ejecución del programa continúa con la siguiente acción fuera de este bloque *if.*

La sintaxis es:

```
if (Condición)
  /* Acción a ejecutar si la condición es verdadera. */
```

Si son más de una acción las que se tienen que ejecutar en caso de que la condición evaluada se cumpla (condición verdadera), se agregan llaves al inicio y fin del bloque de acciones, como se muestra a continuación.

```
if (Condición){
  /* Acciones a ejecutar si la condición es verdadera.*/
}
```

Los paréntesis son obligatorios alrededor de la *Condición.*

Es importante mencionar, que la *Condición* es una expresión lógica que denota la situación específica que da un resultado de 0 si es falso y diferente de 0 si es verdadero. De donde se ejecutan las acciones correspondientes sólo si la evaluación es distinta de cero.

Nótese que es el mismo razonamiento cuando el argumento de la función o estructura *if* consta sólo de una variable o alguna constante, si es cero, no se ejecutan las acciones siguientes, pero si es distinto de cero, sí se ejecutan.

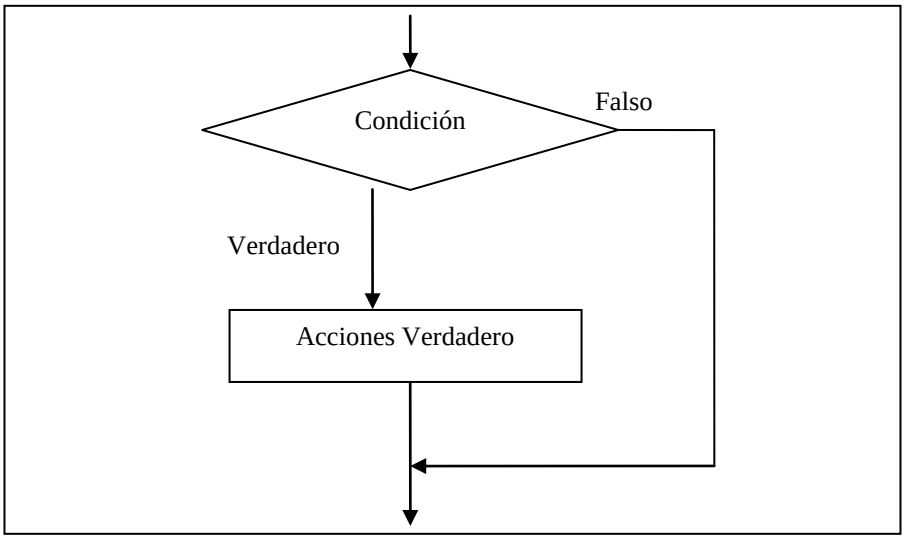
El Pseudocódigo*, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación.

PSEUDOCÓDIGO	**LENGUAJE C**

Si (Condición), **entonces**
 	
Acciones Verdadero
 	
Fin_Si
	if (Condición)
	{
	Acciones Verdadero;
	}

* En el Capítulo 2 se mostró **otra** forma de escribir el pseudocódigo, utilizando las palabras clave en inglés, como *if* para *si* en este caso. Lo mismo para el resto de las estructuras que se verán en este capítulo.

Diagrama de Flujo



EJEMPLO 4.2.1.1 Determinar si la figura que tiene el usuario es un triángulo, un cuadrado o un polígono.

Este programa pregunta al usuario de cuántos lados consta su figura y dependiendo del número de lados le enviará un mensaje en el que le indique qué tipo de figura es. Considerando únicamente: triángulo, cuadrado y polígono.

```
#include <stdio.h>

main()
{
  int lados;

  printf("¿Cuántos lados tiene tu figura? ");
  scanf("%d", &lados);

  if (lados==3)
    printf("La figura es un triángulo ”);

  if (lados==4)
    printf("La figura es un cuadrado ”);

  if (lados>=5)
    printf("La figura es un polígono ”);
}
```

EJEMPLO 4.2.1.2 Calcular el perímetro de un triángulo y decir de qué tipo es.

Este programa calcula el perímetro de un triángulo, dadas las medidas de los lados por el usuario e indica qué tipo de triángulo es (equilátero, isósceles o escaleno).

```
#include <stdio.h>

main()
{
  float lado1, lado2, lado3, perimetro;

  printf("¿Cuánto mide el primer lado del triángulo? ");
  scanf("%f", &lado1);
  printf("\n ¿Cuánto mide el segundo lado del triángulo? ");
  scanf("%f", &lado2);
  printf("\n ¿Cuánto mide el tercer lado del triángulo? ");
  scanf("%f", &lado3);

  perimetro = lado1 + lado2 + lado3;

  printf("El perímetro del triángulo es: %f\n", perimetro);

 //Comparaciones de los lados para definir el tipo de
 //triángulo.

  if (lado1 == lado2 && lado1 == lado3)
    printf("El triángulo es equilátero”);

  if ((lado1 == lado2 && lado1 != lado3) || (lado1 == lado3
    && lado1 != lado2) || (lado2 == lado3 && lado1 != lado2))
    printf("El triángulo es isósceles”);

  if (lado1 != lado2 && lado1 != lado3)
    printf("El triángulo es escaleno”);
}
```

En este ejemplo se tienen tres condicionales **if's** en los que hay: **comparaciones y los conectores lógicos AND (&&) y OR (||).**

Puesto que un **if** simplemente prueba el valor numérico de una expresión, que el compilador se encarga de determinar y que se recomienda no usar porque puede ser confuso, son posibles ciertas abreviaciones de código. Lo más obvio es escribir

```
if (expresión)
```

En lugar de

```
if (expresión != 0)
```

Ya que sería incluso redundante.

Por otro lado, el error más común cuando se evalúan condiciones, es hacer una asignación en lugar de una igualación en la estructura **if**.
Si por ejemplo se tiene que verificar si una variable "a" tiene el valor 5, debe tenerse el cuidado de NO escribir

```
if (a = 5)     /* Condición incorrecta, si lo que se quiere es
                  verificar si la variable a vale 5. De la
                  forma en que está escrita (con un signo "="[5]
                  ), se refiere a asignación y no a comparación
                  y siempre se ejecutarán las acciones
                  correspondientes a ese bloque pues es un
                  valor distinto de cero */
```

En lugar de

```
if (a == 5)     /* Condición correcta, la comparación utiliza
                   los signos "=="[6] y se ejecutarán las
                   acciones correspondientes a este bloque,
                   sólo si esta comparación es verdadera, que
                   es lo que se busca en este caso */
```

Cuando se hace una asignación dentro de un **if** (de manera errónea) se corre el riesgo de que siempre se ejecuten las acciones correspondientes, cuando esta asignación es diferente de cero. En el ejemplo, siempre ejecutaría las acciones, ya que la expresión *a = 5* siempre es verdadera.
Unos compiladores dan un mensaje de advertencia, si el programador intenta hacer una asignación de esa manera, pero otros lo aceptan sin dar mensajes de advertencia.

4.2.2 SELECCIÓN DOBLE `if - else`

También llamada *estructura if-else*. Equivalente al condicional en español:

si – si no
ó
si – en otro caso.

Esta estructura permite controlar la ejecución de acciones cuando se presentan dos opciones alternativas de acción. Por la naturaleza de éstas, se debe ejecutar una o la otra, pero no ambas a la vez, es decir, son mutuamente excluyentes.

La sintaxis de la estructura **if-else** es:

```
if (Condición)
  /* Acción a ejecutar si la condición es verdadera. */
else
  /* Acción a ejecutar si la condición es falsa. */
```

[5] Las comillas son sólo para resaltar el símbolo. En el código no se escriben.

[6] Las comillas son sólo para resaltar los símbolos. En el código no se escriben.

Los paréntesis son obligatorios alrededor de la Condición al igual que en la estructura **if** de la estructura condicional simple.

También puede ser un bloque de acciones (más de una), y la sintaxis cambia (se agregan las llaves como en la estructura **if**):

```
if (Condición) {
  /* Acciones  a ejecutar si la condición es verdadera.*/
}
else {
  /* Acciones  a ejecutar si la condición es falsa.*/
}
```

Si se necesita elegir entre varios bloques de acciones, se pueden utilizar **if's** anidados de la siguiente manera:

```
if (Condición_una) {
  /* Acciones  a ejecutar si la Condición_una es verdadera.*/
}
else {
  if (Condición_dos) {
    /* Acciones  a ejecutar si la Condición_dos es
    verdadera.*/
  }
  else  {
    /* Acciones  a ejecutar si todas las demás condiciones
    son falsas.*/
  }
                    }
```

Esto permite hacer una selección múltiple.

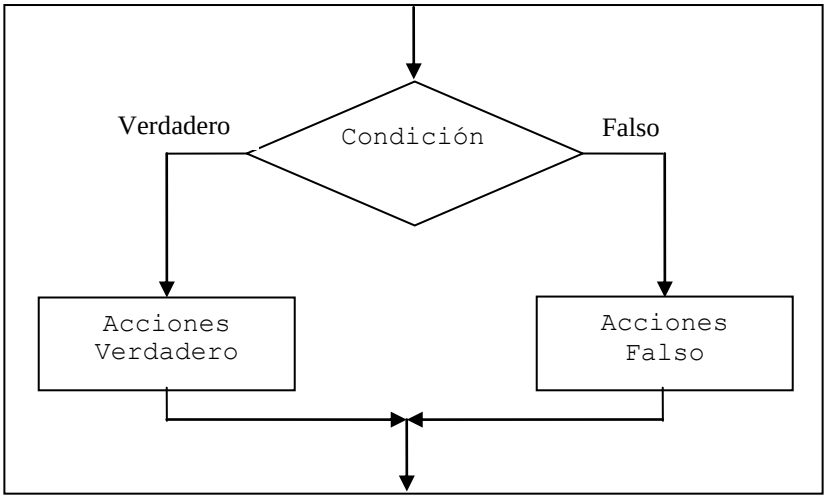
El ejemplo *4.3.2.3* trata un ejemplo de anidamiento de **if's**.

El Pseudocódigo, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación.

PSEUDOCÓDIGO	**LENGUAJE C**

Si (Condición), **entonces**
 	
Acciones Verdadero
 	
Si no
Acciones Falso
 	
Fin_Si
	if (Condición)
	{
	Acciones Verdadero;
	}
	
	else
	{
	Acciones Falso;
	}

Diagrama de Flujo



EJEMPLO 4.2.2.1 Dado un número entero, decir si es par o impar.

Este programa indica si el número dado por el usuario es par o impar.

```
#include <stdio.h>

main(){
  int numero;

  printf(" Dame un número entero ");
  scanf("%d", &numero);

//Estructura para verificar si el número es par o impar.
  if ( numero % 2 == 0 )
    printf("\n El número es par " );
  else
    printf("\n El número es impar ");
}
```

Recuérdese que "**%**"[7] en la condición es el operador *módulo*.

4.2.3 SELECCIÓN MÚLTIPLE `SWITCH`

Permite controlar la ejecución de acciones cuando se tienen más de dos opciones alternativas a ejecutarse; y que los posibles valores que tomará la variable de control (*opcion*), estén comprendidos en un conjunto ordenado y finito de valores como lo es el tipo entero y caracter.

[7] Las comillas son sólo para resaltar el símbolo. En el código no se escribe. Véase Capítulo 3.

La sintaxis es:

```
switch (opcion) {
case constante_1: Acciones_1;
                  break;
case constante_2: Acciones_2;
                  break;
    .
    .
    .

case constante_n: Acciones_n;
                  break;
default:  Acciones;
}
```

La estructura **switch** evalúa la expresión **opcion** y después busca su valor en las constantes que tienen la palabra reservada **case** antepuesta. Si encuentra el valor en la lista de constantes, las acciones correspondientes son ejecutadas. Si no encuentra el valor y hay un **default** (que es opcional), se ejecuta la lista de acciones correspondientes al **default**. En ambas situaciones o en caso de no encontrar ningún valor en la lista de constantes, la ejecución continúa con la siguiente línea de código, fuera de la estructura.

Es muy importante usar la palabra reservada: **break**; para terminar una lista de acciones. Si no se termina con **break**, la ejecución va a continuar con la próxima lista de acciones, del siguiente caso. Esto se puede usar si se debe ejecutar la misma lista de acciones para diferentes valores de la expresión **opcion**, lo cual no es recomendable.

El Pseudocódigo, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación. El código en Lenguaje C considera que **opcion** es de tipo entero.

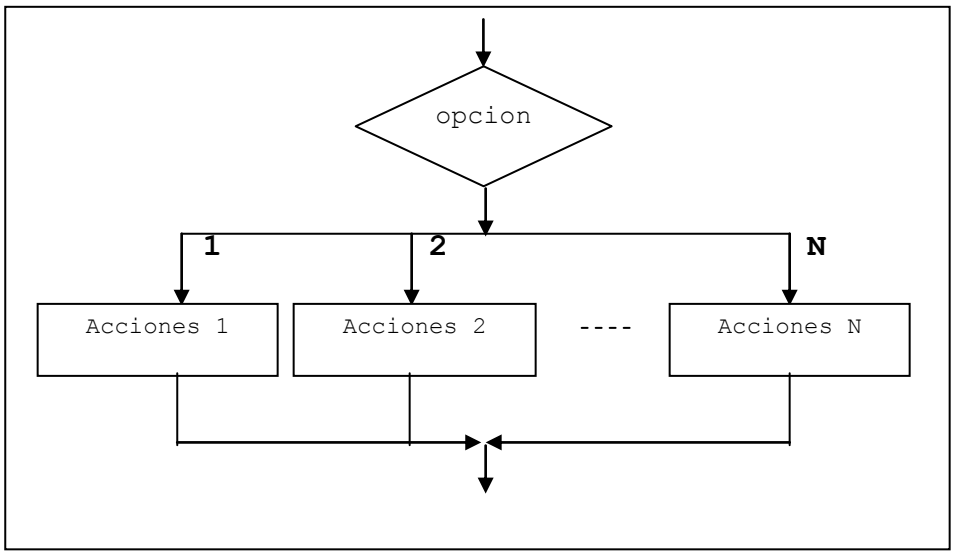
PSEUDOCÓDIGO

```
Seleccionar  (opcion)
   Caso 1  Acciones 1
   Caso 2  Acciones 2
      .
            .
            .
   Caso N  Acciones N

Fin_Seleccionar
```

LENGUAJE C

```
switch  (opcion)
{
case 1:  Acciones 1;
          break;
      case 2:   Acciones 2;
                break;
    .
          .
    .
 case N:  Acciones N;
                 break;
       }
```

Diagrama de Flujo



EJEMPLO 4.2.3.1 Ejemplo con `switch`.

Este programa indica el mes del año, dado un número de 1 a 12.

```
#include <stdio.h>

main()
{
  int numero;

  printf(" Dame el número de mes (de 0 a 12) ");
  scanf("%d", &numero);

  printf("\n El número %d corresponde al mes de: ", numero );

 //Comienza la evaluación.

 switch( numero )
  {
    case 1:
          printf("\n Enero ");
          break;
    case 2:
          printf("\n Febrero ");
          break;
    case 3:
          printf("\n Marzo ");
          break;
```

```
        case 4:
              printf("\n Abril ");
              break;
        case 5:
              printf("\n Mayo ");
              break;
        case 6:
              printf("\n Junio ");
              break;
        case 7:
              printf("\n Julio ");
              break;
        case 8:
              printf("\n Agosto ");
              break;
        case 9:
              printf("\n Septiembre ");
              break;
        case 10:
              printf("\n Octubre ");
              break;
        case 11:
              printf("\n Noviembre");
              break;
        case 12:
              printf("\n Diciembre ");
              break;
        default:
              printf("\n Ningún mes, porque el número que diste no
              está en el rango solicitado ");

      }  //Fin del switch.
    }  //Fin de la función principal.
```

Es importante recordar, que si se define la variable que se evaluará en el **switch** como **char**, se debe escribir el caracter correspondiente entre apóstrofes (o comillas simples) en el **case**.[8]

4.3 ESTRUCTURA REPETITIVA

Como su nombre lo indica, son estructuras que permiten **repetir procesos** hasta que se cumpla o deje de cumplir una condición.

Para abordarlas, es necesario conocer dos conceptos muy importantes: **contador** y **acumulador**, ya que se requieren en muchos casos.

[8] Véase el ejemplo 4.3.2.2.

CONTADOR: Es una variable de tipo entero que va a recorrer un conjunto de datos. Se inicializa generalmente en cero o uno, aunque esto puede variar según el problema. Dentro del ciclo tiene un incremento o decremento, para así contar los elementos necesarios del conjunto de datos, el cual varía dependiendo del problema. Tal vez se requiera recorrer los elementos del conjunto o de una lista en orden ascendente de uno en uno, en este caso el contador se incrementará. Si el requerimiento es en orden descendente, el contador decrementará en uno en cada iteración. Cabe mencionar que esos incrementos o decrementos pueden ser de dos en dos, de tres en tres, etc.

ACUMULADOR: Es una variable de tipo numérico, cuya función es contener la suma o agregado de un determinado conjunto de datos. Generalmente se inicializa con cero o uno y dentro del ciclo generalmente se incrementa con lo que tenga la variable que contiene el dato a acumular. Al final, después del fin del ciclo, el contenido del acumulador es el total acumulado.

4.3.1 ESTRUCTURA `WHILE`

Su equivalente al español es **mientras**.

Es una estructura de control que permite hacer una repetición en un intervalo de cero a *n* veces. Esto se debe a que la condición de control del ciclo se coloca al principio de la estructura y se ejecutan las acciones correspondientes contenidas en el ciclo, mientras la condición sea verdadera. En caso de que no se cumpla la condición se termina el ciclo.

La estructura **while** tiene la siguiente sintaxis:

```
while (Condición)
  /* Acción a ejecutar en cada ciclo.*/
```

Como en las estructuras anteriores, la estructura **while** puede también ser un bloque de acciones (más de una), como en el siguiente caso, en donde se utilizan las llaves para delimitarlas:

```
while (Condición){
  /* Bloque de acciones a ejecutar en cada ciclo.*/
}
```

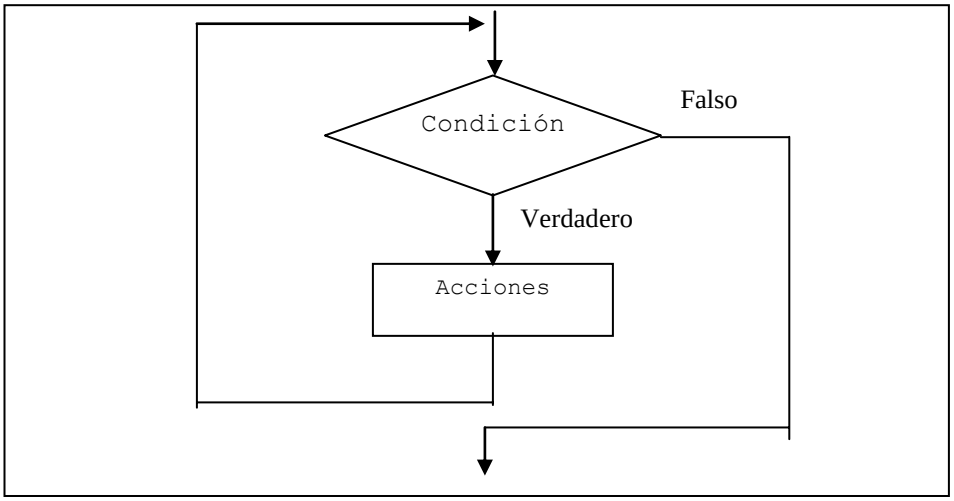
Primero es evaluada la *Condición*. Si es verdadera, la acción o acciones son ejecutadas. Después es evaluada la *Condición* otra vez. Estos dos pasos son repetidos hasta que la *Condición* se evalúa y el resultado es cero, es decir, es falsa. Es necesario notar que los paréntesis son obligatorios alrededor de la C*ondición*.

El Pseudocódigo, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación.

PSEUDOCÓDIGO	LENGUAJE C

Mientras (Condición) **Hacer**
Acciones
Fin_Mientras
	while (Condición)
	{
	Acciones;
	}

Diagrama de Flujo



EJEMPLO 4.3.1.1 Calcular la suma de los primeros *n* números naturales.

Dado un número entero positivo, calcula la suma desde el cero al número entero positivo dado.
Si el usuario proporciona el 4, el resultado será 10, puesto que 0 + 1 + 2 + 3 + 4 = 10.
En general, si el usuario ingresa *n*, el resultado será *suma = 0 + 1 + 2 + … + n.*

```
#include <stdio.h>
main(){
  int numero, contador, suma;
  printf("¿Hasta qué número quieres sumar? ");
  scanf("%d", &numero);
  contador = 1;
  suma = 0;
//Comienza el ciclo repetitivo. La suma se acumula en la
//variable suma.
  while (contador <= numero) {
     suma += contador;
     contador ++;
  }
  printf("\n La suma de los %d números es %d.", numero, suma);
}
```

4.3.2 ESTRUCTURA `do - while`

La estructura **do - while** es casi igual a la estructura **while**.

Permite controlar la ejecución de una acción o grupo de acciones en forma repetitiva, siempre que se cumpla la condición del ciclo repetitivo. Se usa muy frecuentemente en los menús y para validar la información de entrada.

Su sintaxis es:

```
do
  /* Acción a ejecutar en cada ciclo.*/
while (condición);
```

Las llaves son utilizadas para escribir más que una acción, como en el siguiente caso:

```
do {
  /* Bloque de acciones a ejecutar en cada ciclo.*/
} while (Condición);
```

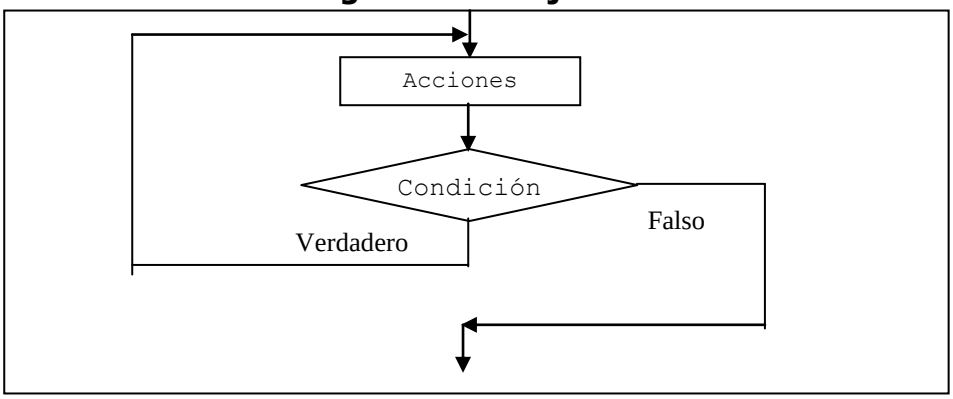
La única diferencia entre la estructura **do - while** y la estructura **while**, es que la estructura **do - while** ejecuta la acción o acciones antes de evaluar la *Condición*, es decir, por lo menos se ejecutan una vez. Estos dos pasos son repetidos hasta que la *Condición* se evalúa y el resultado es cero (falsa). Nótese que los paréntesis en la *Condición* son obligatorios, así como el punto y coma después de la misma.

El Pseudocódigo, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación.

PSEUDOCÓDIGO	**LENGUAJE C**

Haz
 	
Acciones
 	
Mientras (Condición)
	do
	{
	Acciones;
	}
	while (Condición);

Diagrama de Flujo



EJEMPLO 4.3.2.1 Calcular la suma de los primeros *n* números naturales
Dado un número entero positivo, calcula la suma desde el cero al número entero positivo dado. Si el usuario proporciona el `4`, el resultado será `10`, puesto que `0 + 1 + 2 + 3 + 4 = 10`. En general, si el usuario ingresa *n*, el resultado será *suma = 0 + 1 + 2 + … + n.*

```
#include <stdio.h>

main()
{
  int numero, contador, suma;
//Validar datos.
  do{
     printf("¿Hasta qué número quieres sumar? ");
     scanf("%d", &numero);
  }while (numero<=0); //Termina la validación de datos.

  contador = 0;   //Inicialización del contador.
  suma = 0;  //Inicialización del acumulador.

//Se hace la suma.
  do{
     suma += contador;
     contador ++;
  }while (contador <= numero); //Termina la suma.
  printf("\n La suma de los %d números es %d.", numero, suma);
}
```

EJEMPLO 4.3.2.2 Un menú, que usa *switch* y *do – while.*

Este programa solicita dos números enteros y muestra un menú en el que el usuario puede elegir alguna de las siguientes operaciones con esos números: suma, resta, multiplicación o división.

```
#include <stdio.h>

main()
{
  char c;
  int a, b;

  printf("Dame un número entero. a = ");
  scanf("%d", &a);
  printf("Dame otro número entero. b = ");
  scanf("%d", &b);

//Esta estructura repetitiva logra que el Menú se muestre
//siempre que el usuario elija una opción que no se incluye en
//el mismo.
```

```
  do {
    printf("\nMenú\n");
     printf("1)Suma: a+b\n");
     printf("2)Resta: a-b\n");
     printf("3)Multiplicación: a*b\n");
     printf("4)División: a/b\n");
     printf("5)Salir\n");

    printf("\n¿Qué quieres hacer con esos números? Escoge una
           opción: ");
     c=getch();
     printf("\n");

  //Inicio de la estructura condicional switch, por medio de la cual se
  //realiza la operación según la opción elegida en el Menú por el
  //usuario.

     switch (c) {
     case '1':
       printf("%d + %d = %d\n", a, b, a+b);
       break;
     case '2':
       printf("%d - %d = %d\n", a, b, a-b);
       break;
     case '3':
       printf("%d * %d = %d\n", a, b, a*b);
       break;
     case '4':
       if (b != 0)
         printf("%d / %d = %d\n", a, b, a/b);
       else {
         printf("Error: división con cero \n");
       }
       break;
     case '5':
       printf("\nHasta la vista \n");
       break;

     default:
       printf("\nOpción inválida \n");
       break;
     } //Fin del switch.
   } while (c != '1' && c != '2' && c != '3' && c != '4' && c != '5');
   //Fin del do-while.
} //Fin de la función principal.
```

EJEMPLO 4.3.2.3 Un menú, que usa *if's* anidados y *do – while.*

Este programa solicita dos números enteros y muestra un menú en el que el usuario puede elegir alguna de las siguientes operaciones con esos números: suma, resta, multiplicación o división.

```
#include <stdio.h>
main(){
  char c;
  int a, b;

  printf("Este programa permite sumar, restar, multiplicar o
         dividir dos números \n");
  printf("Dame un número entero. a = ");
  scanf("%d", &a);
  printf("Dame otro número entero. b = ");
  scanf("%d", &b);

  do {
    printf("\nMenú\n");
    printf("1)Suma: a+b\n");
    printf("2)Resta: a-b\n");
    printf("3)Multiplicación: a*b\n");
    printf("4)División a/b\n");
    printf("¿Qué quieres hacer con esos números? Escoge
            una opcion: ");
    scanf("\n%c", &c);
    printf("\n");
//Uso la estructura if-else en lugar de la estructura switch.
    if (c == '1')
      printf("%d + %d = %d\n", a, b, a+b);
    else {
       if (c == '2')
          printf("%d - %d = %d\n", a, b, a-b);
       else {
          if (c == '3')
             printf("%d * %d = %d\n", a, b, a*b);
          else {
             if (c == '4') {
                if (b != 0)
                   printf("%d / %d = %d\n", a, b, a/b);
                else
                   printf("Error: división con cero \n");
             }  //Fin de if (c == '4')
             else  //De if (c == '4').
                 printf("Escoge opción 1, 2, 3 o 4\n");
          } //Del else de if (c == '3').
       } //Del else de if (c == '2').
    } //Del else de if (c == '1').
  } while (c !='1'&&c!='2'&&c!='3'&&c!='4'&&c!='5');
} //Fin del programa principal
```

Como puede observarse, es un tanto complicada la lectura de varios **if's** anidados, y se corre el riesgo de perderse fácilmente en las llaves de apertura y cierre de los bloques, incluso al escribirlas. En este caso se recomienda el uso de la estructura **switch**.

4.3.3 ESTRUCTURA *FOR*

Permite formar un ciclo repetitivo, el cual es controlado por un contador que tiene que definirse con un valor inicial, un valor final y un incremento o decremento. Esto significa que debe conocerse de antemano el número de veces que se debe repetir el ciclo. Este tipo de ciclo se repite *n* veces, es decir, un número conocido y finito de veces.

Su sintaxis es:

```
for (inicialización; condición; incremento o decremento)
  /* Acción ejecutada en cada ciclo. */
```

Nótese que los paréntesis son obligatorios.

Como siempre se puede usar las llaves para agrupar un bloque de acciones:

```
for (inicialización; condición; incremento o decremento) {
  /* Bloque de acciones a ejecutar en cada ciclo.*/
}
```

Primero se ejecuta la expresión *inicialización*, antes de comenzar el ciclo. Se puede usar para inicializar las variables que se utilizan dentro del ciclo. Por supuesto este puede hacerse en una línea antes del ciclo **for**. Sin embargo, incluir la inicialización dentro del ciclo ayuda a identificar las variables del mismo.

La expresión *condición* se usa para determinar si debe continuar o concluir el ciclo. Cuando la expresión condicional se evalúa como falsa, el ciclo terminará. La expresión *condición* es evaluada antes del primer ciclo.

Una vez más, cabe mencionar que en el lenguaje C si la expresión tiene un valor diferente de cero, significa verdadero, si la expresión evalúa a cero, significa falso.La expresión *incremento o decremento* se emplea para modificar la variable del ciclo en cierta forma, cada vez que ha sido ejecutado la acción en el cuerpo del ciclo **for**.

El siguiente es un ejemplo de un ciclo **for** básico que imprime en pantalla los números del 0 al 100 de uno en uno.

EJEMPLO 4.3.3.1 Numeración del cero al cien de uno en uno.

Este programa imprime en pantalla los números del cero al cien.

```
#include <stdio.h>

main()
{
   //Inicio del ciclo.

   for (i = 0; i <= 100; i++)
      printf("i = %d\n", i);  //Fin del ciclo.
}
```

En este ejemplo, no es necesario solicitar dato alguno al usuario, pues se cuenta con la información necesaria para cumplir el objetivo del mismo.

Cabe mencionar que se puede omitir cualquiera de las tres expresiones en el **for**. Si la *condición* no está presente, se toma como permanentemente verdadera. El compilador no marca ningún error, pero evidentemente no es muy común que se quiera un programa con estas características, como se muestra a continuación.

EJEMPLO 4.3.3.2 Ciclo infinito.

Este programa carece de las tres expresiones, formando un ciclo infinito.

```
#include <stdio.h>

main()
{

  for (;;)
  printf("Este es un ciclo infinito");
}
```

El código anterior imprime en la pantalla lo que se encuentra entre comillas de manera infinita, ya que no existe condición para salir de él; éste solo podrá ser suspendido por la invocación de una llamada al sistema (Ctrl-C) o "matando" el proceso que generó el programa.

El Pseudocódigo, Diagrama de Flujo y código en Lenguaje C de esta estructura se muestran a continuación. En lo que respecta al Diagrama de Flujo, se presentan 4 formas distintas. El estudiante elegirá la que mejor le convenga.

PSEUDOCÓDIGO

```
Desde inicio Hasta fin Incremento/decremento

   Acciones

Fin_Desde
```

LENGUAJE C

```
for (inicialización; condición; incremento/decremento)
{
   Acciones;
}
```

Diagrama de Flujo 1

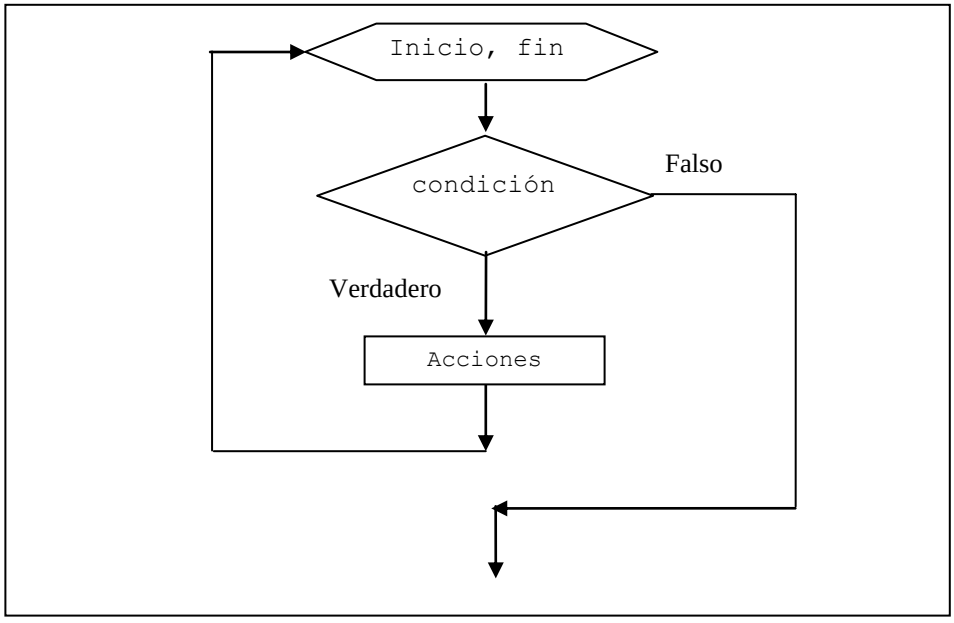
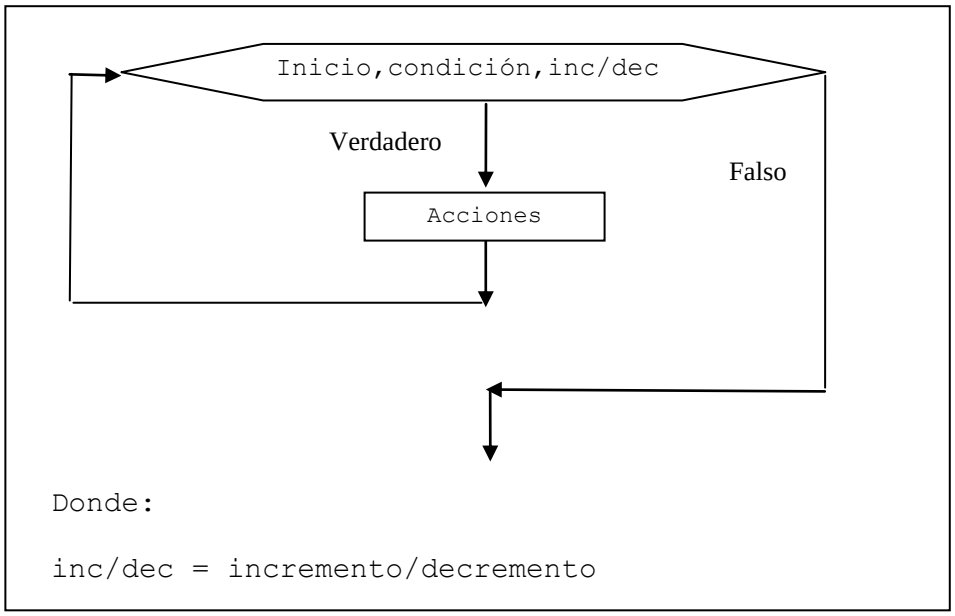


Diagrama de Flujo 2



Donde:

inc/dec = incremento/decremento

Diagrama de Flujo 3

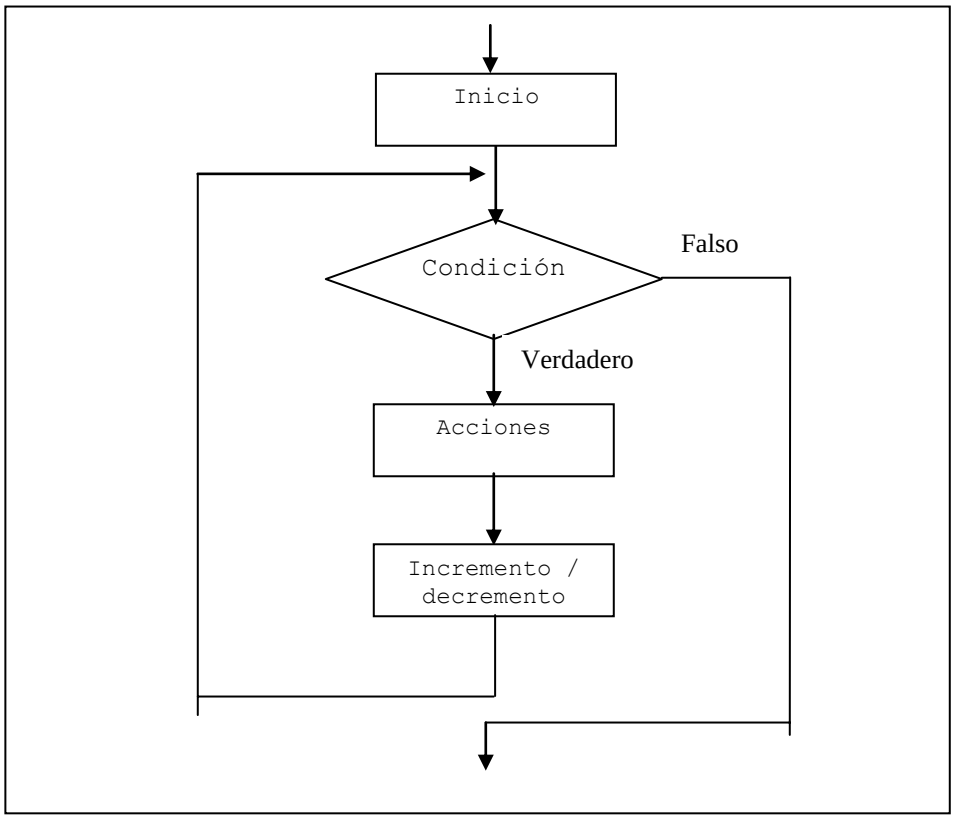
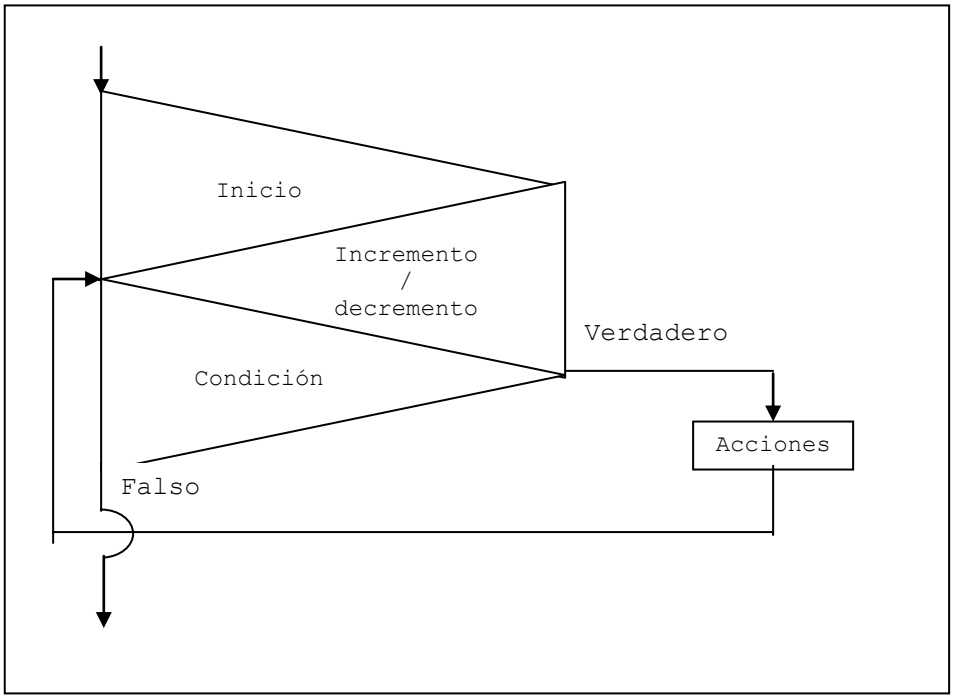


Diagrama de Flujo 4



EJEMPLO 4.3.3.3 Calcular la suma de los primeros *n* número enteros positivos.

Este programa calcula la suma de los primeros *n* números dado el número *n*, usando un **for**.

```
#include <stdio.h>

main(){
  int numero, suma, contador;

  printf("¿Hasta qué número quieres saber la suma? ");
  scanf("%d", &numero);

  suma = 0;  //Inicializa acumulador en 0.

  for (contador=1; contador<=numero; i++)
    suma = suma + contador;

  printf("\n la suma del 0 al %d es %d ", numero, suma);
}
```

Cabe mencionar que la variable *suma* puede inicializarse al declararse. Otro comentario importante es que sólo existe una acción en el ciclo, por lo que no se

escribe entre llaves.

En este ejemplo, el **contador** se llama *contador* y el **acumulador** se llama *suma.* Se recomienda hacer una prueba de escritorio con todas la variables involucradas y diferentes valores de la variable *numero*.

EJEMPLO 4.3.3.4 Calcular el factorial de un número.

Este programa calcula el factorial de un número entero positivo dado, utilizando **for**.
Recuérdese que el factorial de un número es la multiplicación sucesiva desde el 1 al número del que se desea saber el factorial. Por ejemplo, si se desea saber el factorial de 5, el resultado se obtiene de la siguiente manera: *1 * 2 * 3 * 4 * 5 = 120.*

```
#include <stdio.h>
main(){
  int numero, factorial, i;
  printf("\t\t***FACTORIAL DE UN NÚMERO***\n\n ");
  printf("¿De qué número quieres saber el factorial? ");
  scanf("%d", &numero);
//Inicialización del acumulador en 1.
  factorial = 1;
//Inicio del ciclo.

  for (i=1; i<=numero; i++)
    factorial = factorial * i; //Fin del ciclo.

//Impresión de resultados.

  printf("El factorial de %d es %d \n", numero, factorial);
}
```

Nótese que en este ejemplo, el **contador** se llama *i* y el **acumulador** *factorial.* De igual manera, se recomienda hacer una prueba de escritorio.

EJEMPLO 4.3.3.5 *for*'s ANIDADOS

Supongamos que se requiere escribir una lista con 5 elementos en forma vertical, numerada del 1 al 5 y que en cada fila o renglón aparezcan tantos asteriscos como el número de renglón o fila correspondiente. Esto es:

1 *
2 **
3 ***
4 ****
5 *****

Para lograrlo se necesitan 2 **for's**, uno para los números de los renglones y otro dentro del anterior, para la impresión de los asteriscos. Nótese que todas las estructuras se pueden anidar según requerimientos del problema a resolver.

```
     #include <stdio.h>
main()
{
         int i, j;

      //Inicio del ciclo que imprime los números de línea.

               for (i=1; i<=5; i++)
               {
                  printf("\n %d ",i);

                 //Inicio del ciclo para la impresión de
                 //asteriscos.

                  for(j=1; j<=i; j++)
                     printf("*"); //Fin del ciclo de asteriscos.

               } //Fin del ciclo de los números de línea.
      } Fin del programa principal.
```

4.4 EJEMPLOS COMPLEMENTARIOS

EJEMPLO 4.4.1 El algoritmo de Euclides, para calcular el máximo común divisor

Recuérdese que el máximo común divisor (*mcd*) de un par de números es el número más grande que divide a ambos números. En este algoritmo se van probando los divisores del mayor al menor.

Sean *número A* y *número B*, los números de los que se quiere obtener el *mcd*. Si alguno de los dos números divide al otro éste es el *mcd*. Por ejemplo, el *mcd* entre 10 y 5, es 5. Puesto que *10 % 5 = 0*, lo que indica que 5 divide a 10 y evidentemente 5 divide a 5*.*

Si el número menor, por ejemplo *número 2*, no es el *mcd*, entonces el *mcd* se encuentra entre 1 y *residuo a*, donde *residuo a = número A % número B.* ¿Por qué? Se deja al lector que lo analice. Si *residuo a* no es cero, el *mcd* se encuentra entre 1 y *residuo b*, donde *residuo b = número B % residuo a.* Y así sucesivamente, hasta que el resultado del *módulo* sea cero, lo cual indica que se encontró el divisor máximo que es el penúltimo residuo.

Recuérdese que "*%*" es el operador módulo.

Por ejemplo, el *mcd* entre `17` y `5` es `1`. El procedimiento para calcularlo fue:

número A = 17, *número B* = 5

número A % número B = residuo a. Sustituyendo: 17 % 5 = 2.

Entonces *residuo a* = 2.

Como *residuo a* ≠ 0, entonces

número B % residuo a = residuo b. Sustituyendo: 5 % 2 = 1.

Entonces *residuo b* = 1.

Como *residuo b* ≠ 0, entonces

residuo a % residuo b = residuo c. Sustituyendo: 2 % 1 = 0.

Entonces *residuo c* = 0

Como *residuo c = 0,* entonces el *mcd* es 1, esto es, *residuo b*.

Para escribir el código se utiliza una variable temporal llamada *temp* que funciona como "puente" para facilitar el cálculo. Para llegar a una mejor comprensión del mismo, véase el código y realícese una prueba de escritorio, tomando en cuenta el ejemplo que se acaba de describir.

```
#include <stdio.h>
main(){
  int numero1, numero2, temp;
  printf("Dame el primer número = ");
  scanf("%d", &numero1);
  printf("Dame el segundo número = ");
  scanf("%d", &numero2);
  printf("\n El divisor más grande entre %d y %d es ",
  numero1, numero2);
  while ( numero2 != 0 )
  {
    temp = numero2;
    numero2 = numero1 % numero2;
    numero1 = temp;
  }
  printf("%d\n", numero1);
}
```

EJEMPLO 4.4.2 Calcular el máximo común divisor de las parejas de números que el usuario quiera.

Este programa calcula el máximo común divisor (*mcd*) de dos números Primero calcula el *mcd* de los dos números que el usuario ingrese, después le pregunta si quiere volver a calcular otro *mcd*, en caso de que su respuesta sea afirmativa, se repite el procedimiento para calcularlo, desde la solicitud de datos. Y así sucesivamente.

```
#include <stdio.h>
main(){
  int numero1, numero2, temp;
  char c;
//Inicio del ciclo que permite al usuario volver a ingresar
//otro par de números para obtener su mcd.
  do {
    printf("Dame el primer número = ");
    scanf("%d", &numero1);
    printf("Dame el segundo número = ");
    scanf("%d", &numero2);
    printf("\nEl divisor más grande entre %d y %d es ",
    numero1,numero2);
    while ( numero2 != 0 ) {
      temp = numero2;
      numero2 = numero1 % numero2;
      numero1 = temp;
    } //Fin while.
    printf("%d\n", numero1);
    printf("\nQuieres probar otra vez (s/n)? ");
    scanf("\n%c", &c);
    printf("\n");
  } while (c == 'S' || c == 's'); //Fin do-while.
}
```

EJEMPLO 4.4.3 Calcular una aproximación del número π mediante el método de Leibniz.

El método de Leibniz para encontrar una aproximación del número π **(PI)** se basa en el cálculo de la serie:

$$\pi = 4\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

La aproximación al número **PI** depende del número de iteraciones, entre más iteraciones se realicen, la aproximación es mejor.

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

main()
{
  int iteraciones, i;
  double pi;

  printf("Cuantas iteraciones quieres: ");
  scanf("%d", &iteraciones);

  pi = 0.0; //Inicializa acumulador.

  for (i=0; i<iteraciones; i++)
       pi += pow(-1,i) / (2*i + 1);

  pi *= 4;

  printf("pi = %lf\n", pi);
}
```

4.5 EJERCICIOS PROPUESTOS

Elaborar el algoritmo y el código fuente. Compilar y ejecutar para resolver los siguientes ejercicios.

1. Dados dos números enteros positivos identificar el mayor, el menor, o decir si son iguales. Identificar también si son pares o impares.
2. Las próximas reinscripciones en su escuela se llevarán a cabo dependiendo del número de materias que inscribió y certificó. Si certificó un número mayor o igual que el número de materias que inscribió, se deberá reinscribir el 1er. día de reinscripciones. Si la diferencia entre las que inscribió y certificó es de 1 ó 2, se reinscribirá el 2ndo. día. En cualquier otro caso, se reinscribirá el 3er. día. Haga un programa, que dadas las materias que inscribió y certificó, indique el día de reinscripción. Garantice la validez de los datos; esto es: que sean números enteros y positivos, que el número de materias inscritas no sobrepasen de 5 y el número de certificaciones no sobrepase al número de materias de su ruta curricular. Indique las suposiciones que hiciste en caso de necesitarlo.
3. Se encuentra frente a la Biblioteca *Supercómputo S.A. de C.V,* tiene 4 niveles y desea saber qué tema o área de estudio está en un nivel específico. Haga un programa, utilizando la estructura **switch** que dado un nivel, le indique qué tipo de libros se encuentran en dicho nivel. Es necesario que sepa que en el nivel 1 están los libros de sistemas operativos, en el 2 los de microprocesadores, en el 3 los de Lenguajes de Programación y otro software y en el último nivel se encuentran los libros más extraños en computación que te puedes imaginar (si desea llegar a este nivel, sólo recuerde que no hay elevador). Garantice que indique al usuario "opción inválida" cuando escribe algo diferente a 1, 2, 3 ó 4.
4. Haga un programa que pida al usuario el número de sumandos que va a ingresar para realizar una suma, solicite cada sumando y devuelva la suma.
5. Dado un número entero positivo, diga cuántos números impares y pares hay del 1 al número dado. Hágalo con **while**, **do-while** y **for** e identifique las diferencias.

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Capítulo 5
Funciones.

Tabla de contenido:

5. Funciones.

Ya conoce las funciones; hablamos de ellas en los capítulos 3 y 4 de este curso. En este capítulo aprenderá lo necesario para crear sus propias funciones y de esta forma, modularizar el desarrollo del programa. También conocerá acerca de los argumentos de la función **main()**.

El objetivo es descomponer la resolución de un problema en tareas menos complejas, creando módulos.

Un módulo es el recurso de un lenguaje de programación que permite resolver una tarea o subproblema específico y que puede ser llamado desde el programa principal. Suelen ser de notación prefija, con argumentos (en su caso) separados por comas y encerrados entre paréntesis.

Las funciones como ***sqrt, tolower,*** etc., no son sino ejemplos de módulos que resuelven tareas concretas.

```
    /* Ejemplo: Programa que obtiene la hipotenusa de un
triángulo rectángulo a partir de los catetos.  */
        main( ){
              double lado1, lado2, hip, aux;

              /* asignación de valores a los lados */_

              aux = lado1*lado1+lado2*lado2;
              hip= sqrt(aux);
               }
```

Si queremos realizar una misma operación para dos triángulos:

```
main( ){
     double  lado1_TA, lado2_TA, aux_A, hip_A,
  lado1_TB, lado2_TB, aux_B, hip_B;

          /* Asignación de valores a los lados */
     aux_A = lado1_TA*lado1_TA+lado2_TA*lado2_TA;
     hip_A = sqrt(aux_A);

     aux_B = lado1_TB*lado1_TB+lado2_TB*lado2_TB;
     hip_B = sqrt(aux_B);

     }
```

Sería más claro, menos propenso a errores y más reutilizable, si hubiese tenido en alguna biblioteca la función *Hipotenusa*, de forma que el código fuese el siguiente:

```
main( ){
          double  lado1_TA, lado2_TA, hip_A,
  lado1_TB, lado2_TB,  hip_B;

     /* Asignación de valores a los lados */

               hip_A = Hipotenusa(lado1_TA, lado2_TA);
hip_B = Hipotenusa(lado1_TB, lado2_TB);
      }
```

En este ejemplo, *Hipotenusa* es un módulo que resuelve la tarea de calcular la hipotenusa de un triangulo, sabiendo el valor de los lados.

Nota: Podría dar la sensación de que primero debemos escribir el programa con todas las sentencias y luego construir la función *Hipotenusa*. Esto no es así: el programador debe de identificar las funciones antes de escribir una sola línea de código.

5.1 Concepto de función.

A través de este curso hemos utilizado funciones en los programas desde el primero hasta el último. Es que las funciones constituyen una parte básica de la programación en el lenguaje C y son las que definen el concepto de *Programación Estructurada*. Ellas permiten, al programador, diseñar la lógica de un programa por tareas o acciones, en lugar de diseñarlo escribiendo código de principio a fin. Además como una acción puede realizarse más de una vez, las funciones permiten reutilizar ese código.

```
    /* Ejemplo: Programa que lee dos reales, calcula el
       promedio y lo eleva al cuadrado e imprime el resultado
    */

#include <stdio.h>

double Cuadrado(double entrada){
        return entrada * entrada;
                                }

double Media(double a, double b){
        return ((a+b)/2);
                                 }
```

```
main( ){
 double x, y, r;

 printf("\n\t Dame dos numeros reales: ");
 scanf("%lf %lf", &x, &y);

 r = Cuadrado(Media(x,y));

 printf("\n\t Resultado de %lf y %lf es: %lf\n \n",x,y, r);
}
```

Este ejemplo, que aparenta una gran sencillez, es una muestra de lo que un buen programador debe lograr. Al leer el código del programa, en la función **main()**, uno ya tiene una noción de qué es lo que hace el programa, ya que las diferentes tareas están divididas en "módulos" constituidos por las funciones.

Tarea aparte será la programación de cada una de ellas, pero lo más importante es que esta forma de trabajar brinda varias ventajas:

1) Posibilidad de reutilizar el código.
2) Simplificación de la función **main()**.
3) Facilidad en la localización de errores,
4) Si una función tiene un error, bastará con corregirlo una vez para que todos los procesos que utilizan ese fragmento de código se corrijan.

Como programadores, debemos iniciar la construcción de un programa pensado más bien en qué es lo que va a hacer éste y en los datos que necesitamos, en lugar de comenzar pensando cómo lo vamos a hacer. Cada una de las acciones clave del programa será responsabilidad de una función específica. Luego, nuestro trabajo se podrá dividir en varios módulos más pequeños, dedicados a la programación de cada una de las funciones.

5.1.1 Definición.

Una función es un módulo que devuelve un valor cuando es llamado.
Algunas funciones conocidas: ***sin()***, ***abs()***, ***sqrt()***

Para definir nuestras propias funciones:

```
    <tipo> <nombre_fiunción>([<parámetros formales>]) {
           [<sentencias>]
           return <expresión>;
                                                      }
```

- Por ahora, la definición se pondrá después de la inclusión de bibliotecas y antes del ***main***. En general, antes de usar un módulo en cualquier sitio, hay que poner su definición.

- Diremos que <tipo> <nombre_función>(<parámetros formales>) es la cabecera de la función.

- El cuerpo de la función debe contener:
  return <expresión>;
  dónde **<expresión>** ha de ser del mismo tipo que el especificado en la cabecera de la función (también puede ser un tipo compatible). El valor de dicha expresión es el valor que devuelve la función.

  Ejemplo de una función, que devuelve el cuadrado de entrada.

```
double Cuadrado(double entrada){
         return entrada*entrada;
                                }
```

5.1.2 Parámetros formales y actuales.

- Los parámetros formales son aquellos especificados en la cabecera de la función (entrada).
  Al declarar un parámetro formal hay que especificar su tipo de dato.
  Los parámetros formales sólo se conocen dentro del módulo.

- Los parámetros actuales son las expresiones pasadas como argumento en la llamada a una función.

<nombre_función>(<lista de parámetros actuales>);

Ejemplo:

```
        double Cuadrado(double entrada){
          return entrada*entrada;
                                  }

        main( ){
           double valor, resultado;
valor = 4;
resultado  = Cuadrado(valor);   // resultado = 16
printf("El cuadrado de %lf es %lf \n", valor, resultado);
             }
```

Control de flujo:

Cuando se ejecuta la llamada **resultado = Cuadrado(valor);** el control de flujo salta a la definición de la función.

- Se realiza la correspondencia entre parámetros.
  El correspondiente parámetro formal recibe una copia del parámetro actual, es decir, en tiempo de ejecución se realiza la asignación.

  parámetro formal = parámetro actual

  En el ejemplo, entrada = 4

- Empiezan a ejecutarse las sentencias de la función y cuando llega alguna sentencia ***return*** **<expresión>**, la función termina y devuelve <expresión> al módulo llamador.

- A continuación, el control de flujo prosigue por la línea siguiente a la llamada.

Correspondencia entre parámetros actuales y formales.

- Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales.

  Ejemplo de correspondencia incorrecta de parámetros.

```
double Cuadrado(double entrada){
    return entrada*entrada;
                            }

main( ){
 resultado = Cuadrado(5, 8);  // Error en compilación
     . . . . . .
           }
```

- La correspondencia se establece por orden de aparición, uno a uno y de izquierda a derecha.

Véase el ejemplo de la Hipotenusa.

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

double Hipotenusa(double ladoA, double ladoB){
          return sqrt(ladoA*ladoA+ladoB*ladoB);
          }

main( ){
       double lado1, lado2, hip;

       printf("\nIntroduzca el primer lado ");
       scanf("%lf", &lado1);
       printf("\nIntroduzca el segundo lado ");
       scanf("%lf", &lado2);
       hip = Hipotenusa(lado1, lado2);
       printf("\nLa hipotenusa vale %lf ", hip);
            }
```

- Cada parámetro formal y su correspondiente parámetro actual han de ser del mismo tipo (o compatible)

  Problema: que el tipo del parámetro formal sea más pequeño que el actual. El formal se puede quedar con basura.

```
int Cuadrado(int entrada){
               resultado = Cuadrado(400000000000);
// devuelve basura
     . . . . . .
                 return resultado;
         }
```

- El parámetro actual puede ser una expresión.
  Primero se evalúa la expresión y luego se realiza la llamada al módulo.

```
hip = Hipotenusa(ladoA+3,ladoB*5);
```

- Dentro de una función se puede llamar a cualquier otra función que esté definida con anterioridad. El paso de parámetros entre funciones sigue los mismos criterios.

En el ejemplo mostrado a continuación se observa que dentro de la función **sqrt( )** se llama a la función **Cuadrado( )** definida con anterioridad.

```
#include <stdio.h>
#include <math.h>
double Cuadrado(double entrada){
          return entrada*entrada;
          }

double Hipotenusa(double ladoA, double ladoB){
          return sqrt(Cuadrado(ladoA) + Cuadrado(ladoB));
          }
```

5.2 Alcance de las variables locales y globales.

Dentro de una función podemos declarar constantes y variables. Sólo se conocen dentro de la función. Se conocen como datos locales.

```
<tipo> <nombre_función>(<lista de parámetros formales>){
     [<Constantes Locales>]
     [<Variables Locales>]
     [<Sentencias>]

     return <expresión>;
         }
```

Ejemplo. Calcular el factorial de un valor.

```
#include <stdio.h>
int Factorial(int n)   {
     int i;
     int aux =1;

     for (i=2; i<=n; i++)
           aux=aux * i;

     return aux;
   }
```

```
main(){
     int valor, resultado;

     printf("\nIntroduzca el valor ");
     scanf("%d", &valor);

     resultado = Factorial(valor);
     printf("Factorial de %d = %d \n", valor, resultado);
          }
```

Dentro de una función no podemos acceder a datos definidos en otros sitios, a menos que sean variables globales (ver al final el concepto de variable global) y cuyo nombre no aparezca como variable local. (Por ejemplo, en ***main***). Ni viceversa.

```
#include <stdio.h>

int Factorial(int n)
    {
     int i;
     int aux =1;

     valor = 5;     // Error de compilación

     for (i=2; i<=n; i++)
          aux=aux * i;

     return aux;
     }

main(){
     int valor, resultado;
     int i;    // Error de compilación

     printf("\nIntroduzca el valor ");
     scanf("%d", &valor);

     resultado = Factorial(valor);
     printf("Factorial de %d = %d \n", valor, resultado);
          }
```

Las variables locales no inicializadas a un valor concreto tendrán un valor

indeterminado (basura) al inicio de la ejecución de la función.

Ámbito de un dato (variable o constante) "*v*" es el conjunto de todos aquellos módulos que pueden referenciar a "*v*".

- Sólo puede usarse en el propio módulo en el que está definido (ya sea un parámetro formal o un dato local)
- No puede usarse en otros módulos ni en el programa principal.
- En C no se permite definir variables locales a un bloque.

Al estar perfectamente delimitado el ámbito de un dato, los nombres dados a los parámetros formales pueden ser iguales a los actuales.

```
#include <stdio.h>

int Factorial(int valor) {
     int i;
     int aux = 1;
     for (i=2; i<=valor; i++)
          aux = aux * i;
     return aux;
}

int main(){
     int valor = 3, resultado;

     resultado = Factorial(valor);
     printf("Factorial de %d = %d \n", valor, resultado);
     /* Imprime en pantalla lo siguiente: Factorial de 3 = 6
*/
}
```

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

double Hipotenusa(double ladoA, double ladoB){
     return sqrt(ladoA*ladoA+ladoB*ladoB);
          }

main( ){
     double ladoA, ladoB, hip;
```

```
     printf("\n Introduzca el primer lado");
     scanf("%lf", &ladoA);
     printf("\n Introduzca el segundo lado");
     scanf("%lf", &ladoB);

     hip = Hipotenusa(ladoA, ladoB);
     printf("\nLa hipotenusa vale %lf ", valor);
   }
```

Incluso podemos cambiar el valor del parámetro formal, que el actual no se modifica.

```
#include <stdio.h>

int SumaDesde0Hasta(int tope){
     int suma;

     suma=0;
     while (tope>0){
               suma = suma + tope;
                tope--;
              }

     return suma;
}

main( ){
          int tope=5, resultado;

          resultado = SumaDesde0Hasta(tope);
          printf("Suma hasta %d = %d ", tope, resultado);

/* Imprime en la pantalla lo siguiente: Suma hasta 5 = 15 */

            }
```

En la medida de lo posible, como norma de programación, procura darles nombres distintos.

Cada vez que se llama a una función, se crea un entorno de trabajo asociado a él, en una zona de memoria específica: la Pila.

- En principio, cada entorno, sólo puede acceder a sus propios datos.

- En el entorno se almacenan, entre otras cosas:
  - Los parámetros formales.
  - Los datos locales (constantes y variables)
  - La dirección de retorno de la función.
- Cuando un módulo llama a otro, sus respectivos entornos se almacenan apilados uno encima del otro. Hasta que no termine de ejecutarse el último, el control no pasará al anterior.

```
#include <stdio.h>

int Factorial(int valor) {
                    int i;
                    int aux = 1;

                    for(i=2; i<=valor; i++) aux = aux * i;

                    return aux;
                  }

int Combinatorio(int a, int b) {
     return Factorial(a)/(Factorial(b) * Factorial(a-b));
                                  }

main( ) {
          int resul, elementos=5, grupos=2;

     resul = Combinatorio(elementos, grupos);
     printf(" %d sobre %d = %d ",elementos, grupos, resul);
        }
```

main( ) es de hecho una función como otra cualquiera, por lo que también se almacena en la pila. Es la primera función llamada al ejecutarse el programa.

Devuelve un entero al Sistema Operativo:

- Si el programa va bien➔ se debe devolver 0.
  Puede indicarse incluyendo ***return 0;*** al final de ***main*** (antes de **}**)
- Si el programa no va bien ➔ debe devolver un entero distinto de 0.

Ejemplos de funciones mal codificadas:

```
int f(int n) {
          return n+2;
          printf("Nunca se ejecuta \n");
           }
```

La función **f** devuelve **n+2** y termina, nunca ejecuta **printf( )**.

```
int Factorial(int n) {
     int i, aux=1;

     for(i=2; i<=n; i++) {    aux = aux * i;
               return aux;
                          }
                }
```

La función **Factorial** devuelve el valor de **aux** sin terminar el **for**.

```
int EsPar(int n) {
               if (n%2==0) return 1;
              }
```

La función **EsPar** no tiene **return** si la condición es falsa.

```
int EsPrimo(int n) {
                  int i;
                  for (i=2; i<=sqrt(n); i++)
                    if (n%i == 0) return 0;
                         return 1;
               }
```

La función **EsPrimo** si la condición **if** es verdadera el **for** no lo termina.

En lo posible, no introduzca sentencias ***return*** en varios sitios distintos del cuerpo de la función, cuidando de no tener mas de una salida como lo indica uno de los principios de la programación estructurada. La instrucción ***return*** deberá dejarla al final de la función.

Revisemos este ejemplo:

```
main( ) {
          double lado1, lado2, hip, i;

          for (i=1; i<=3; i++) printf("\n***************");
          printf(" Programa básico de Trigonometría ");
          for (i=1; i<=3; i++) printf("\n***************");

          printf("\n Introduzca el primer lado");
          scanf("%lf", &lado1);
          printf("\n Introduzca el segundo lado");
          scanf("%lf", &lado2);

          hip = Hipotenusa(lado1, lado2);
          printf("\nLa Hipotenusa es %lf ", hip);
             }
```

Sería mejor que el código de la función principal quedará de la forma siguiente:

```
main( ) {
          double lado1, lado2, hip, i;

          Presentacion( );

          printf("\n Introduzca el primer lado");
          scanf("%lf", &lado1);
          printf("\n Introduzca el segundo lado");
          scanf("%lf", &lado2);

          hip = Hipotenusa(lado1, lado2);
          printf("\nLa Hipotenusa es %lf ", hip);
              }

/* Presentación es un procedimiento, no devuelve valor. */

void Presentacion( ) {
          for (i=1; i<=3; i++) printf("\n***************");
               printf(" Programa básico de Trigonometría ");
          for (i=1; i<=3; i++) printf("\n***************");
                       }
```

En este ejemplo, *Presentacion( )* es un módulo que resuelve la tarea de realizar la presentación del programa por pantalla.

Observe que *Presentacion( )* no devuelve ningún valor, como por ejemplo las funciones **sqrt** o *Hipotenusa*. Por eso, su llamada constituye una sentencia y no aparece dentro de una expresión.

Al igual que comentamos en las funciones, el programador debe de identificar los procedimientos antes de escribir una línea de código.

Tipos de módulos

Funciones:

- Son módulos que devuelven un valor en la misma llamada.
- No constituyen una sentencia del programa, sino que han de formar parte de las expresiones.
- Ejemplos: **sin(**real**)**, **sqrt(**real**)**, **pow(**real,real**)**,**Hipotenusa(**real**)**

Procedimientos:

- Son módulos que no devuelven ningún valor en la llamada.
- Constituyen una sentencia del programa.
- Ejemplos: **system( )**, **rand( )** (de la biblioteca stdlib), **Presentación( )**

5.3 Paso de parámetros a funciones.

El paso de parámetros y la definición de datos locales siguen las mismas normas que con las funciones.

```
void <nombre_procedimiento>(<lista de parámetros formales>){
     [<Constantes Locales>]
     [<Variables Locales>]
     [<Sentencias>]
      }
```

Observar que no hay sentencia **return**. El procedimiento termina cuando se ejecuta la última sentencia del mismo.

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

double Hipotenusa(double ladoA, double ladoB){
     return sqrt(ladoA*ladoA+ladoB*ladoB);
   }

void Presentación( ){
     printf("Programa básico de Trigonometría");
     printf("\nAutor: MA\n\n");
    }

void MostrarHipotenusa(double valor){
     printf("\nLa hipotenusa vale %lf ", valor);
     }

main( )
{
     double lado1, lado2, hip;

     Presentación (  );

     printf("\n Introduzca el primer lado");
     scanf("%lf", &lado1);

     printf("\n Introduzca el segundo lado");
     scanf("%lf", &lado2);

     hip = Hipotenusa(lado1,lado2);

     MostrarHipotenusa(hip);

}
```

Llamadas entre procedimientos:

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

void Pausa( ) {
               printf("\n\tfin\n\n");
               system("exit");
             }

void ImprimirAsteriscos(int x){
              int i;
              for(i=1; i<=x; i++) printf("*");
                                  }


void MostrarResultados(int coc, int res)
  {
     ImprimirAstericos(33);
     printf("\n\tEl cociente es: %d\n ", coc);
     printf("\n\tEl residuo es: %d\n ", res);
     ImprimirAsteriscos(33);
     Pausa( );
  }

main( ) {
          int dividendo, divisor, cociente, residuo;

          printf("\n\tintroduzca un entero: ");
          scanf("%d", &dividendo);
          printf("\n\tIntroduzca otro entero:  ");
          scanf("%d", &divisor);
          cociente = dividendo / divisor;
          residuo = dividendo % divisor;
          MostrarResultados(cociente, residuo);
             }
```

Veamos qué ocurre en el llamado de una función.

Paso de parámetros.

Existen diferentes métodos para la transmisión o *paso de los parámetros* a subprogramas. Es preciso conocer el método adoptado por cada lenguaje, ya que no siempre son los mismos. Dicho de otra forma, un mismo programa puede producir diferentes resultados bajo diferentes sistemas de paso de parámetros.

Los parámetros pueden ser clasificados como:

Entradas: se proporcionan valores desde el programa que los llama y que se utilizan dentro del subprograma.

Salidas: las salidas producen los resultados del subprograma, este devuelve un valor calculado por dicha función.

Entradas / Salidas: un solo parámetro se utiliza para mandar argumentos a un programa y para devolver resultados.

Los métodos mas empleados para realizar el paso de parámetros son:

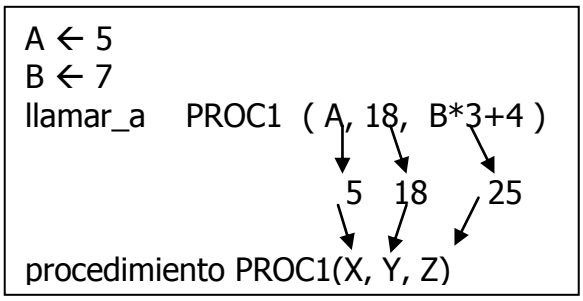
- **paso por valor** (también conocido por parámetro valor)
- **paso por referencia o dirección** (también conocido por parámetro variable)

Paso por valor.

Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Los parámetros locales a la función reciben como valores iniciales los valores de los parámetros actuales y con ello se ejecutan las acciones descritas en el subprograma.

A continuación se muestra el mecanismo de **paso por valor** de un procedimiento con tres parámetros.



Aquí podemos ver que se realiza una copia de los valores de los argumentos en la llamada al procedimiento en las variables X, Y, Z, pero en ningún caso los valores de A y B se verán modificados.

Por ejemplo:

```
int f(int);       //Prototipo de la función f

int Leer( ){
                   int vleido;
                   printf("\n\n\tDame el valor de x: ");
                   scanf("%d",&vleido);
                   return vleido;
                 }

void Escribe(int x, int y){
                   printf("\n\n\tEl valor de x es: %d",x);
                   printf("\n\tEl valor de y es: %d\n\n",y);
                                        }

main( ) {
                 int x, y;
                 x = Leer( );
                 x = x + 1;
                 y = f(x);
                 Escribe(x, y);
               }

int f(m) {
                  int x;
           x = m;
                  x = x * x;
                  m = x;
                  return  m;
                }
```

Si en la aplicación de este programa damos a **x** el valor de 5, la siguiente línea lo incrementa en 1 y llamamos a la función con f(6). En la llamada a la función se hace una copia del valor de la variable **x** del programa principal en la variable **m** de la función.

Una vez dentro de la función se declara otra variable **x** que es distinta a la del programa principal, siendo después asignada al valor de **m**, luego calculamos el cuadrado, lo asignamos a **m** y retornamos el valor. Aquí finaliza la llamada a la función. En el programa principal escribimos los valores de la **x** e **y** dando como resultado x = 6 e y = 36. Es importante entender este mecanismo.

5.4 Recursividad.

Se dice que una función es recursiva cuando se define en función de si misma. No todas la funciones pueden llamarse a si mismas, deben estar diseñadas especialmente para que sean recursivas, de otro modo podrían conducir a bucles infinitos, o a que el programa termine inadecuadamente.

El lenguaje C permite la recursividad. Cuando se llama a una función, se crea un nuevo juego de variables locales, de este modo, si la función hace una llamada a si misma, se guardan sus variables y parámetros en la pila, y la nueva instancia de la función trabajará con su propia copia de las variables locales, cuando esta segunda instancia de la función retorna, recupera las variables y los parámetros de la pila y continua la ejecución en el punto en que había sido llamada.

En lenguaje C, las funciones se pueden llamar a sí mismas. Una función es *recursiva* si una sentencia en el cuerpo de la función se llama a sí misma. A veces se denomina definición circular, la recursión es el proceso de definir algo en términos de sí mismo.

Existen muchos ejemplos de recursión. Una forma recursiva de definir un número entero es como los dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, más o menos un número entero. Por ejemplo, el número 15 es el número 7 más el número 8; 21 es 9 más 12 y 12 es 9 más 3.

Para que un lenguaje de computadora sea recursivo, tiene que ser posible que una función se llame a sí misma. Un ejemplo sencillo es la función factr( ), que calcula el factorial de un entero. El factorial de un número es el producto de todos los enteros entre 1 y ese número: Por ejemplo el factorial de 3 es 1x2x3, ó 6.

A continuación se da una versión no recursiva denominada **fact( )**:

```
fact( n ){          /* no recursiva */
int n;
     int t, respuesta;
     respuesta = 1;
     for (t=1;t<n;t++)
          respuesta=respuesta*t;
     return(respuesta);
}
```

Esta es la versión recursiva, **factr( )**:

```
factr( n )          /* recursiva */
int n;
{
     int respuesta;
     if(n==1) return 1;
     respuesta=factr(n-1)*n;
     return(respuesta);
}
```

La versión no recursiva de **fact( )** debe quedar clara; utiliza un bucle que empieza por 1 y termina en el número n y progresivamente multiplica cada número por el producto actual y lo asigna a *respuesta*.

Cuando la función recursiva **factr( )** se llama con un argumento de 1, la función devuelve un 1; en otro caso devuelve el producto de **factr(n-1)*n**. Para evaluar esta expresión, a **factr( )** se le llama con n-1. Esto sucede hasta que n es igual a 1 y las funciones empiezan a devolver.

Si se quisiese computar el factorial de 2, la primera llamada es **factr( )** originaría una segunda llamada que se haría con un argumento de 1. Esta llamada devolvería un 1, que luego se multiplicaría por 2 (el valor original de n). Entonces la respuesta es 2.

Cuando una función se llama a sí misma, se almacenan en la pila las nuevas variables locales y los parámetros, y el código de la función se ejecuta con estas variables nuevas desde el principio. Una llamada recursiva no hace una copia nueva de la función. Sólo son nuevos los argumentos. A medida que cada llamada recursiva devuelve, las variables locales y los parámetros son sacados de la pila, y la ejecución se reanuda en el punto de llamada a la función dentro de la función.

La mayoría de las rutinas recursivas no ahorran de forma importante ni tamaño de código ni almacenamiento de variables. Las versiones recursivas de la mayoría de las rutinas se ejecutan un poco más despacio debido a las llamadas de función añadidas, pero esto no será perceptible en la mayoría de los casos. Muchas llamadas recursivas a una función podrían originar un error de sincronización en la pila, pero esto es improbable. Debido a que el almacenamiento de los parámetros de las funciones y de las variables locales se hace en la pila, y debido a que cada nueva llamada crea una copia nueva de esas variables, es posible que la pila se meta en alguna otra zona de datos o de programa en la memoria.

Sin embargo, probablemente no habrá que preocuparse por esto a menos que una función recursiva trabaje desordenadamente.

A veces una función recursiva puede ser más clara y fácil de escribir que esa misma función de forma iterativa. Algunas personas parece que les es más fácil el pensar en una forma recursiva que a otras. Si se siente cómodo con la recursión, entonces utilícela. Si no, utilice los métodos iterativos. Hay autores que indican que el criterio para saber que opción utilizar, es el que si se requiere una definición clara y elegante se use recursividad, a menos que se prefiera eficiencia y en ese caso hay que usar iteración.

Al escribir funciones recursivas, hay que tener donde sea una sentencia **if** para obligar a que la función vuelva sin que se ejecute la llamada recursiva. Si no se hace esto y se llama a la función, no volverá nunca. Este es un error muy común cuando se escriben funciones recursivas. Utilice **printf( )** y **getchar( )** libremente durante el desarrollo para que pueda observar lo que está pasando y abordar la ejecución si ve que ha tenido un error.

Por ejemplo:

Función recursiva para calcular el factorial de un número entero. El factorial se simboliza como n!, se lee como "n factorial", y la definición es:

$$n! = n * (n-1) * (n-2) * ... * 1$$

No se puede calcular el factorial de números negativos, y el factorial de cero es 1, de modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos:

```
     /* Función recursiva para cálculo de factoriales */

int factorial(int n) {
   if(n < 0) return 0;
   else if(n > 1) return n*factorial(n-1); /* Recursividad */
   return 1; /* Condición de terminación, n == 1 */
                              }
```

Veamos paso a paso, lo que pasa cuando se ejecuta esta función, por ejemplo: factorial(4):

1ª Instancia
n=4
n > 1
salida ← 4 * factorial(3) (Guarda el valor de n = 4)

2ª Instancia
n > 1
salida ← 3*factorial(2) (Guarda el valor de n = 3)

3ª Instancia
n > 1
salida ← 2*factorial(1) (Guarda el valor de n = 2)

4ª Instancia
n == 1 → retorna 1

3ª Instancia
(recupera n=2 de la pila) retorna 1*2=2

2ª instancia
(recupera n=3 de la pila) retorna 2*3=6

1ª instancia
(recupera n=4 de la pila) retorna 6*4=24
Valor de retorno → 24

La función factorial es un buen ejemplo para demostrar cómo se hace una función recursiva, sin embargo la recursividad no es un buen modo de resolver esta función, que sería más sencilla y rápida con un bucle "for". La recursividad consume muchos recursos de memoria y tiempo de ejecución, y se debe aplicar a funciones que realmente le saquen partido.

Por ejemplo: visualizar las permutaciones de n elementos.

Las permutaciones de un conjunto son las diferentes maneras de colocar sus elementos, usando todos ellos y sin repetir ninguno. Por ejemplo para A, B, C, tenemos: ABC, ACB, BAC, BCA, CAB, CBA.

```
#include <stdio.h>

/* Prototipo de función */

void Permutaciones(char *, int );

                              /* Ver Apéndice C de los
                              apuntas de Introducción a la
                              programación, referente a
                              apuntadores */

main( )
{
     char palabra[ ] = "ABCDE";

   Permutaciones(palabra,0);
   gets();
}
```

```
void Permutaciones(char *cad, int l) {
   char c;    /* variable auxiliar para intercambio */
   int i, j;     /* variables para bucles */
   int n = strlen(cad);

   for(i = 0; i < n-l; i++) {
          if(n-l > 2) Permutaciones(cad, l+1);
          else printf(" %s,  ",cad);
          /* Intercambio de posiciones */
          c = cad[l];
          cad[l] = cad[l+i+1];
          cad[l+i+1] = c;
    if(l+i == n-1) {
      for(j = l; j < n; j++) cad[j] = cad[j+1];
           cad[n] = 0;
         }
   }
}
```

Ejemplo:
Escriba una función recursiva denominada imprime_num( ) que tenga un argumento entero. Imprirá en la pantalla los números del 1 al n, en donde n es un valor del argumento.

Solución:

```
imprime_num( n)
int n;
{
     if(n==1) printf("%d ",n);
     else {
                          imprime_num(n-1);
               printf("%d ",n);
                    }
}
```

5.5 Elaboración e integración de módulos.

Modularización.

Programar no es únicamente ejecutar el compilador, escribir el programa y distribuirlo.

Problema → Análisis de requisitos ← → Diseño ← →
← → Implementación ← → Validación y Verificación

- Análisis de requisitos.
  Especificación de las necesidades de la institución.

- Diseño
  - Elección de la arquitectura (sistema operativo, servidor web, etc.)
  - Elección de la metodología (estructurada, funcional, orientada a objetos, etc.)
  - Diseño de la solución, atendiendo a la metodología usada.
    Si es estructurada, identificación de las tareas, descomposición modular del problema y diseño de los módulos que resuelven dichas tareas.

- Implementación.
  - Elección de las herramientas de programación (módulos, objetos, etc.)
  - Codificación de los componentes (módulos, objetos, etc.)
- Proceso de validación y verificación.
  Será fundamental la realización de baterías de pruebas.

5.5.1 Diseño modular de la solución.

Datos de entrada y salida.

Algoritmo para calcular la media aritmética:
- Datos de entrada: Un conjunto de N valores numéricos.
- Datos de salida: Un real con la media aritmética.

Algoritmo para dibujar un cuadrado:
- Datos de entrada: longitud de lado (l). Coordenadas iniciales (x,y).
- Datos de salida: ninguno.

Algoritmo para calcular la desviación típica.
- Datos de entrada: Un conjunto de N valores numéricos.
- Datos de salida: Un real con la desviación típica.

Las salidas de un algoritmo son entradas a otros algoritmos.

Algoritmos ←→ Módulos

Datos de entrada ←→ parámetros de los módulos

Parámetros pasados por valor ➔ Datos de entrada
Parámetros pasados por referencia ➔ Datos de salida
Datos de e/s

Datos de Entrada ➔ Función ➔ 1 dato de salida

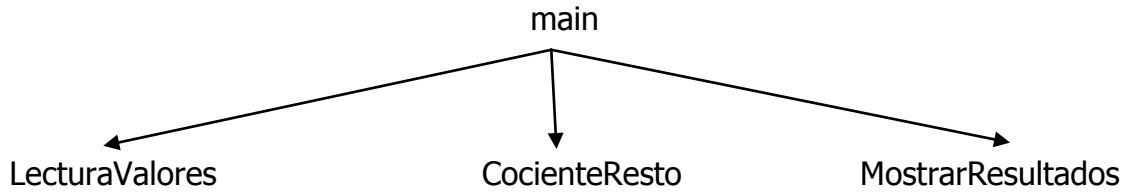
Datos de Entrada ➔ Procedimiento ➔ 0, 2 ó más salidas

Los datos de salida de unos módulos son datos de entrada para otros módulos:

```
int main() {
                    int dividendo, divisor, cociente, resto;
              LecturaValores(dividendo,divisor);

CocienteResto(dividendo.divisor,cociente,resto);
                    MostrarResultados(cociente,resto);
                  }
```

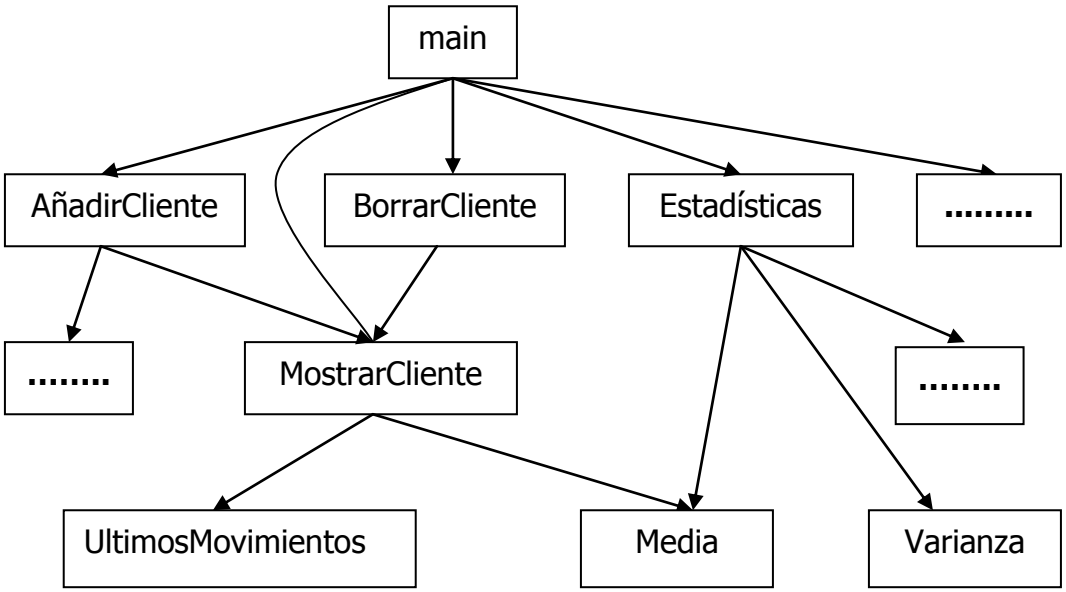
5.5.2 Metodología e integración de los módulos.

La descomposición modular puede ser sencilla como la anterior.



(las flechas indican llamadas de un módulo a otro)

Pero en problemas reales será compleja, con varios niveles:



Pasos a seguir:

- Se realiza el diseño de la descomposición modular de los módulos de alto nivel.

- Se escribe el esqueleto del programa, con las llamadas a los módulos dentro del ***main*** (primer nivel). En esta fase, es muy útil el concepto de prototipo en lenguaje C, ya que permite definir las cabeceras de las funciones y realizar las llamadas, comprobando que éstas son correctas, sin necesidad de definir completamente dichas funciones.

Esto mismo se realiza, descendentemente, en cada nivel, hasta que lleguemos a un nivel suficientemente bajo, que serán módulos que resuelven tareas muy específicas.

Vamos implementando las llamadas a los módulos, aunque no estén terminados (**stubs**).

- En paralelo, o posteriormente, nos centramos en una rama y empezamos a implementar y validar los módulos de los últimos niveles. Así se van integrando en los módulos de niveles superiores.

- Durante este proceso, es posible que haya que revisar la

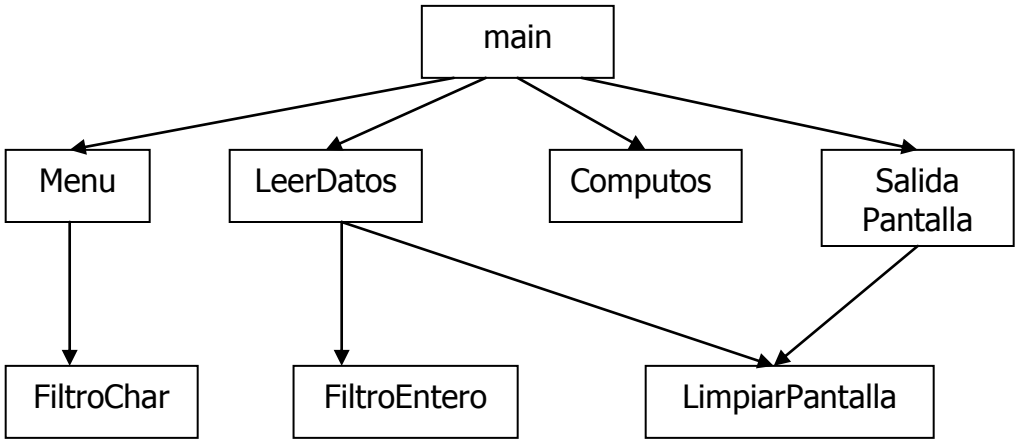
descomposición modular.

Ejemplo:

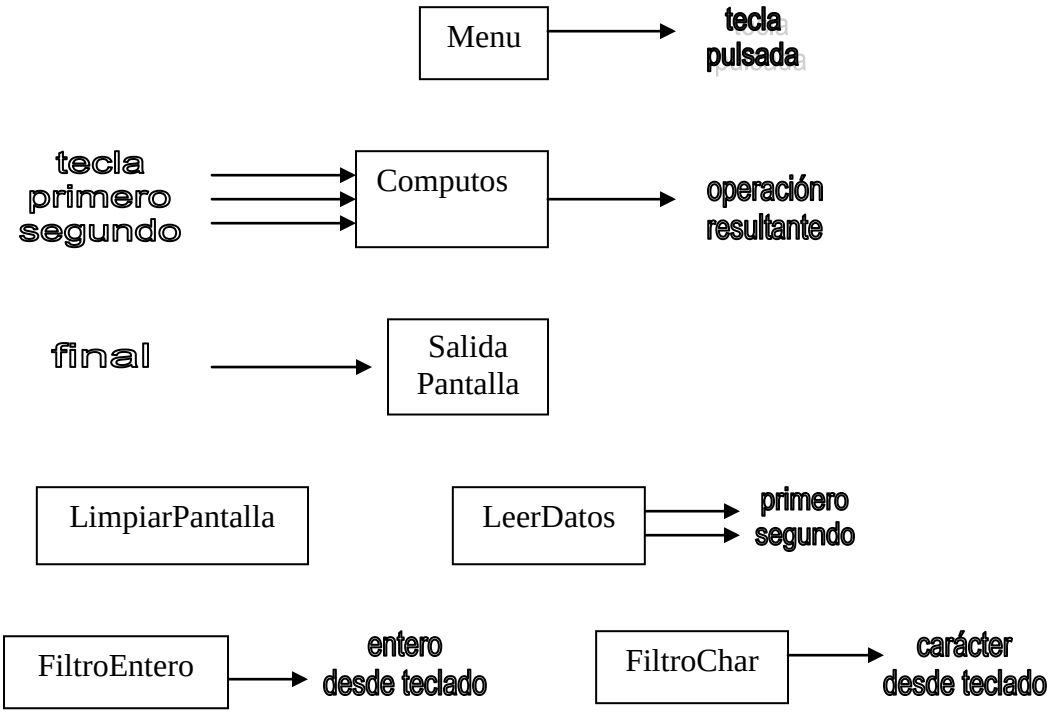
1. Análisis de requisitos.

Ofrecer un menú al usuario para sumar, restar o hallar la media aritmética de dos enteros.

2. Descomposición modular.



Identificación de los datos de entrada y salida



Los prototipos serían:

```
void LimpiarPantalla( );
char FiltroChar( );
int FiltroEntero( );
char Menu( );
void LeerDatos( );
double Computos(char,  int, int );
void SalidaPantalla(double);
```

3. Integración descendente. Nivel 1.

Construimos el esqueleto de ***main*** (todavía no se han definido las funciones)

```
#include <stdio.h>

int primero, segundo;
void LeerDatos(int, int );
char menu( );
double Computos(char , int , int );
void SalidaPantalla(double );

main( ){
          int       dato1, dato2;
          double    final;
          char      opción;

          LeerDatos( );
          dato1=primero;
          dato2=segundo;
          opción = Menu();
          final = Computos(opción, dato1, dato2);
          SalidaPantalla(final);
            }
```

4. Integración descendente. Nivel 2.

Construimos el esqueleto de cada una de las funciones llamadas en el ***main***.

Por ejemplo:

```
void LimpiarPantalla( );
int FiltroEnetro( );

void LeerDatos( ) {
     LimpiarPantalla( );
     primero   = FiltroEntero( );
     segundo  = FiltroEntero( );
       }
```

Hacemos lo mismo con el resto de los módulos: Menu, Computos, SalidaPantalla.

Por ejemplo:

```
char FiltroChar( );

char Menu( ){
     char tecla;

     printf("\nElija una opción\n");
     printf("\nS. Sumar");
     printf("\nR. Restar");
     printf("\nM. Media Aritmética\n");

     tecla = FiltroChar( );
     return tecla;

               }
```

5. Integración ascendente.

Procederíamos a definir los módulos, aunque sea una primera versión (stub)

```
void LimpiarPantalla( ){
                      system("cls");
                      printf("\n\n");
                    }

int FiltroEntero( ){
               int aux;

               printf("Introduzca un entero");
               scanf("%d", &aux);
               return aux;
                }
```

Se pueden implementar mejor los módulos. Pero con esta primera versión, ya es posible empezar las pruebas.
Hacemos lo mismo con el resto de las ramas.

Por ejemplo:

```
     char FiltroChar( ){
                    char caracter;

                    printf("Introduzca un carácter ");
                    scanf(" %c",  &caracter);

                    return caracter;
                      }

     char Menu( ){
                char tecla;

                printf("\nElija una opción\n");
                printf("\nS. Sumar");
                printf("\nR. Restar");
                printf("\nM. Media Aritmética\n");

                tecla = FiltroChar( );

                return tecla;
                   }
```

El programa completo, siguiendo estos pasos, podría quedar como sigue:

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

int primero, segundo;
void LimpiarPantalla( );
char FiltroChar( );
int FiltroEntero( );
char Menu( );
void LeerDatos( );
double Computos(char , int,  int );
void SalidaPantalla(double );

main( ){
     int       dato1, dato2;
     double    final;
     char      opcion;
```

```
     LeerDatos( );
     opcion = Menu( );
     final = Computos(opcion, dato1, dato2);
     SalidaPantalla(final);
        }

void LimpiarPantalla( ){
           system("cls");
     printf("\n\n");
        }

char FiltroChar( ){
     char caracter;

     printf("Introduzca un carácter ");
     scanf(" %c", &caracter);
     return caracter;
        }

int FiltroEntero( ){
     int aux;

     printf("Introduzca un entero ");
     scanf("%d", &aux);
     return aux;
        }


char Menu( ){
     char tecla;

     printf("\nElija una opción\n");
     printf("\nS. Sumar");
     printf("\nR. Restar");
     printf("\nM. Media Aritmética\n");
           tecla = FiltroChar( );
           return tecla;
       }


void LeerDatos( ){
     LimpiarPantalla( );
     primero = FiltroEntero( );
     segundo = FiltroEntero( );
        }
```

```
double Computos(char tecla, int primero, int segundo){
     double resultado;

     switch (tecla){
          case 'S': resultado = primero + segundo;
                    break;
          case 'R': resultado = primero - segundo;
                    break;
          case 'M': resultado = (primero + segundo)/2.0;
                    break;
                   }
     return resultado;
       }

void SalidaPantalla(double final) {
     LimpiarPantalla( );
     printf("\n\nResultado = %lf", final);
        }
```

El diseño del programa principal (main).

Evitar el uso de módulos monolíticos dentro de ***main.*** El programa principal tendrá bastantes líneas de código. Las variables importantes del programa deben estar declaradas como variables de ***main***.

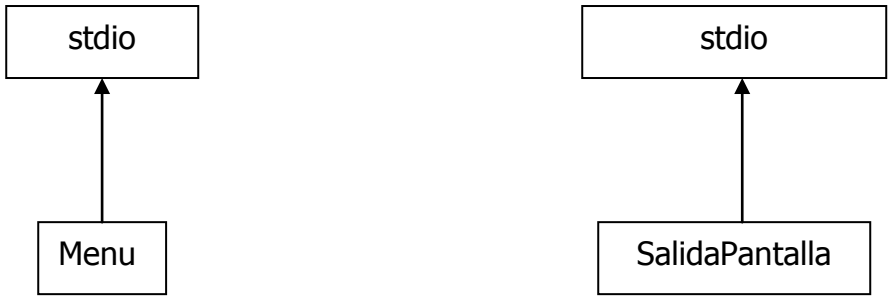
```
void HazloTodo(double primer_cof, double segundo_coef, double
tercer_coef)
        {
     int NumeroRaices;
     double raiz1, raiz2;

     Ec_2_grado(primer_coef, segundo_coef, tecer_coef,raiz1,
     raiz2, NumeroRaices);
     Escribe_Dist(raiz1, raiz2, NumeroRaices);
         }
main( )
       {
     double coef1, coef2, coef3;

     Leer_parabola(coef1,coef2,coef3);
     HazloTodo(coef1, coef2, coef3);
        }
```

Podemos pensar en la función ***main*** como el director de una empresa. El director no es un vago. Se encarga de dirigir a sus empleados.

Diseño de un módulo.

En la fase de definición de prototipos (integración descendente) se deben documentar apropiadamente los módulos.:

- Descripción de la tarea que realiza el módulo
- Descripción de los datos de entrada, incluyendo las restricciones o procedimientos que deben satisfacer para obtener unos datos de salida correctos.
- Descripción de los datos de salida, incluyendo las restricciones o postcondiciones que resultan de su aplicación.
- Indicación de las bibliotecas que necesite. Opcionalmente, se puede incluir un gráfico que lo ilustre.
- A veces, se incluye también en la documentación un diagrama de dependencias (bibliotecas usadas):



Se deben usar comentarios concisos, pero claros y completos, con una presentación visual agradable y esquemática.

De esta forma, construimos fichas que identifican módulos, lo más independientemente posible del lenguaje de programación.

Algunos documentos de los módulos del ejemplo del menú.

```
//-------------------------------------------------------------
/*
     Identificador:    Menu
     Tipo devuelto:  Char
     Cometido: Presentar un menú al usuario para
     sumar, restar o hallar la media de dos enteros.
     Leer la opción del usuario.
     Entradas: Ninguna
     Salidas:   Opción del usuario.
                'S' para sumar
                'R' para restar
                'M' para la media
     Bibliotecas:  stdio
*/
char Menu( );
//-------------------------------------------------------------
```

```
//-------------------------------------------------------------
/*
     Identificador:    Computos
     Tipo devuelto:   double
                    Resultado de la opción realizada.
     Cometido: Dependiendo del valor de tecla:
               sumar, restar o hallar la media
               de dos enteros.
     Entradas:
     tecla:    Opción del usuario
              'S' para suma
                         'R' para restar
                         'M' para la media
     primero:    Primer entero para operar
     segundo:  Segundo entero para operar
     Bibliotecas:  Ninguna
*/
double Computos(char tecla, int primero, int segundo);
//-------------------------------------------------------------
```

```
//-------------------------------------------------------------
/*
     Identificador:     LeerDatos
     Tipo devuelto:   void
     Cometido: Leer dos enteros desde el teclado.
     Entradas:      Ninguna
     Salidas:
          primero:    primer entero
          segundo:  segundo entero
     Bibliotecas:  stdio
*/
void LeerDatos(int &primero, int &segundo);
//-------------------------------------------------------------
```

```
//-------------------------------------------------------------
/*
     Identificador:     SalidaPantalla
     Tipo devuelto:   void
     Cometido: Imprimir en pantalla el resultado de la
               operación seleccionada por el usuario..
     Entradas:
             final:    Valor a mostrar.
     Salidas:       Ninguna
     Bibliotecas:  stdio
*/
void SalidaPantalla(double final);
//-------------------------------------------------------------
```

Los módulos como trabajadores.

Los módulos resuelven tareas específicas. Cómo en la vida real, el trabajador que resuelve una tarea, debe hacerlo de la forma más autónoma posible.

- No dirá al director los detalles de cómo ha resuelto la tarea.
- No pedirá al director más datos ni acciones iniciales de los estrictamente necesarios.
- Lo hará de forma que se pueda reutilizar su esfuerzo desde otras secciones o compañías.
- Debe comprobar que ha realizado correctamente la tarea. Si ha pasado algo, lo comunicará por los cauces establecidos.
- No podrá interferir en el desarrollo de otras tareas.

Con ello, se obtienen módulos autónomos (cajas negras).

Veamos cómo conseguirlo.

Hay que diseñar un módulo pensando en lo que estrictamente necesita.

```
int Factorial (int n) {
               int fac=1;

               for (int i=2 ; i<=n ; i++)
               fac = fact*i;

               return fact;
                        }

main( ){
          printf("\nFactorial de 3 = %d", Factorial(3));
      }
```

En la medida de lo posible, no mezclar Entrada/Cálculo/Salida (E/C/S) en un mismo módulo.

```
int Factorial (int n) {
               int fac=1;

               for (int i=2 ; i<=n ; i++)
               fac = fact*i;

               return fact;
                        }
```

```
void ImprimirFactorial(int n){
     printf("\nEl factorial de %d es %d ",n, Factorial(n));
                 }

main( ){
          ImprimirFactorial(3);
           }
```

En el ejemplo anterior, *ImprimeFactorial* calcula el factorial y lo imprime en pantalla. Pero lo bueno es que se ha conseguido aislar los cálculos del factorial en una función, para que pueda ser usada en otros programas que no requieran escribir el resultado en el periférico de salida por defecto.

Un módulo de cálculo no debe imprimir mensajes. Hágalo, en su caso, en el módulo llamante.

Información de errores.

A veces no es posible que un módulo consiga realizar su tarea correctamente. Por ejemplo, un módulo que lea datos de un fichero e intente acceder a un directorio no exente.

```
double LeerDatosFicheroMAL( ){
          . . . . . .
          if (error)
               printf("\n Se produjo un error ");
          . . . . . .
             }

main( ){
     double valor;
     . . . . . .
     valor = LeerDatosFicheroMAL( );

          // ¿Qué se hace?
         }
```

Se incluirá un parámetro (error) que indique lo sucedido:

- ***int*** si sólo hay que distinguir dos posibles situaciones
- ***char, int*** u otro tipo cuando hay más errores posibles.

```
void LeerDatosFichero(double dato, int error){ . . . . }

main( ){
     int ErrorAcceso;
     double valor;
     . . . . . .
     LeerDatosFichero(valor, ErrorAcceso);

     if (ErrorAcceso)
          printf("\nError de acceso al fichero \n");
     else
          [Operaciones]
        }


void LeerDatosFichero(double dato, char error){ . . . . }

main( ){
     int ErrorAcceso;
     double valor;
     . . . . . .
     LeerDatosFichero(valor, ErrorAcceso);

     switch (ErroAcceso){
          case '0':
             [Operaciones]
                 break;
          case '1':
              printf("\nDisco protegido contra escritura");
                 break;
          case  '2':
              printf("\nFichero no existente");
                 break;
     . . . . . .
     }
     }
```

Si un módulo tiene una variable de control (error), debe asignarle siempre un valor antes de terminar.

Es importante, construir módulos que se puedan reutilizar en otros problemas.

Supongamos que queremos calcular la distancia entre las raíces reales de una ecuación de segundo grado.

Primera modularización.

Ec_2_grado(double coef1, double coef2, double coef3, double distancia, double NumRaices)

dónde NumRaices representa el número de raíces reales. Este módulo será de poca utilidad en otras aplicaciones trigonométricas, ya que no calcula las raíces sino la distancia.

Segunda modularización.

Ec_2_grado(double coef1, double coef2, double coef3, double raiz1, double raiz 2, double NumRaices)

Y en el programa principal calculamos la distancia entre las raíces.

Hay que procurar diseñar un módulo pensando en su posterior reutilización en otros problemas.

La llamada a un módulo no necesita siempre hacer las mismas acciones previas.

Validación de datos devueltos.

Ejemplo. Menú de operaciones.
Hay que comprobar que la operación leída sea correcta.

Primera versión:

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

void SalidaPantalla (double final){
                      printf("\n\nResultado = %lf", final);
                              }
char MenuMAL( ){
     char tecla;

     printf("\nElija una opción\n");
     printf("\nS. Sumar");
     printf("\nR. Restar");
     printf("\nM. Media Aritmética\n");
           scanf("%c", &tecla);
          return tecla;
       }
```

```
main( ){
     int    dato1=3, dato2=5;
     double  resultado;
     char   opcion;

do{
     opcion = MenuMAL( );
     opcion = toupper(opcion);
     } while ((opcion != 'S') && (opcion != 'R') && (opcion
!= 'M'));


switch (opcion){
          case 'S': resultado = dato1 + dato2;
                    break;
          case 'R': resultado = dato1 - dato2;
                    break;
          case 'M': resultado = dato1*1.0/dato2;
                    break;
                     }

     SalidaPantalla(resultado);
       }
```

Segunda versión:

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

void SalidaPantalla (double final){
            printf("\n\nResultado = %lf", final);
                                    }

char Menu( ) {
     char tecla;

     printf("\nElija una opción\n");
     printf("\nS. Sumar");
     printf("\nR. Restar");
     printf("\nM. Media Aritmética\n");
```

```
     do{
          scanf("%c", &tecla);
          tecla = toupper(tecla);
} while ((opcion != 'S')&&(opcion != 'R')&&(opcion != 'M'));

     return tecla;
       }
main( ){
          int    dato1=3, dato2=5;
          double  resultado;
          char   opcion;

               opcion = Menu( );

          switch (opcion){
          case 'S': resultado = dato1 + dato2;
                         break;
          case 'R': resultado = dato1 - dato2;
                        break;
          case 'M': resultado = dato1*1.0/dato2;
                         break;
                       }

          SalidaPantalla(resultado);
                  }
```

Se consiguió un módulo mucho más robusto.

Procurar que los módulos comprueben la validez de los datos devueltos.

Evitar efectos colaterales.

La mayoría de los lenguajes de programación permiten definir variables fuera de los módulos. Automáticamente, su ámbito incluye todos los módulos definidos después. Por eso, se conocen con el nombre de **variables globales**.

El uso de variables globales permite que los módulos se puedan comunicar a través de ellas y no de los parámetros. Esto es pernicioso para la programación estructurada, fomentando la aparición de efectos colaterales.

Ejemplo. Supongamos un programa para la gestión de un aeropuerto. Tendrá dos módulos: *GestiónMaletas* y *ControlVuelos*. El primero controla las cintas transportadoras y como máximo puede gestionar 50 maletas. El segundo controla los vuelos en el área del aeropuerto y como máximo puede gestionar 30. El problema es que ambos van a usar el mismo dato global *Max* para representar dichos máximos.

Primera versión:

```
int Max;
int saturacion;

void GestionMaletas( ){
     Max = 50;      // ← ¡Efecto colateral!
     if (NumMaletasActual <= Max)
          [acción maletas]
     else
          ActivarEmergenciaMaletas( );
      }

void ControlVuelos( ){
     if (NumVuelosActual <= Max)
          [acción vuelos]
     else {
          ActivaEmergenciaVuelos( );
          saturacion = 1;
             }
       }

main( ){
     Max =30;   // ← Máximo número de vuelos

     saturacion = 0;

     while (!saturacion){
     GestionMaletas( ); // → Efecto colateral: Max = 50!
     ControlVuelos( );
                          }
        }
```

Segunda versión:

```
void GestionMaletas(int Max){
     if (NumMaletasActual <= Max)
          [acción maletas]
     else
          ActivarEmergenciaMaletas( );
      }

void ControlVuelos(int Max ){
     int saturacion = 0;

     if (NumVuelosActual <= Max)
          [acción vuelos]
     else {
          ActivaEmergenciaVuelos( );
          saturacion = 1;
             }
     return saturacion;
       }

main( ){
          int saturacion;

          do{
               GestionMaletas(50 );
               saturacion = ControlVuelos(30);
               } while (!saturacion)
                  }
```

Resumiendo: El uso de variables globales, hace que cualquier módulo las pueda usar y/o modificar, lo que provoca efectos colaterales. Esto va encontra de un principio básico en programación: ocultación de información.

Nota. Usualmente, se permite el uso de constantes globales, ya que si no pueden modificarse, no se producen efectos colaterales.

```
     const double Pi = 3.1415927;  // Constante universal

     int FuncionTrigonometrica(int parametro){
          . . . . .// ← Podemos usar la constante global Pi
          }
```

Nos ahorramos un parámetro con respecto a:

```
int FuncionTrigonometrica(double Pi, int parametro){
      . . . . . .
     }
```

En este ejemplo, al trabajar con una constante universal como ¶, podría justificarse el uso de constantes globales. Pero en otros casos, no está claro:

```
int SalarioNeto (int Retencion, int SalarioBruto){
     . . . . . .
}
main( ){
const double ISR = 0.18;      // Constante no universal
          int Sueldo, SalBruto;

          Sueldo = SalarioNeto(ISR, SalBruto);
          . . . . . .
           }

const int ISR = 0.18;

int SalarioNeto (int SalarioBruto){
     . . . . . .
}

main( ){
          int Sueldo, SalBruto;

          Sueldo = SalarioNeto(SalBruto);
          . . . . . .
           }
```

Desventajas en el uso de constantes globales:

- La cabecera del módulo no contiene todas las entradas necesarias.
- No es posible usar la función en otro programa, a no ser que también se defina la constante en el nuevo programa.

Ventajas:

- Las cabeceras de los módulos que usan constantes se simplifican:

Encontrar la solución apropiada a cada problema no es sencillo.

Resumen.

Normas para diseñar un módulo:

- La cabecera debe contener exclusivamente los datos de E/S que sean imprescindibles para el módulo. El resto deben definirse como datos locales.
  ➔ ocultamiento de información.

- Debe ser lo más reutilizable posible:
  - No deben mezclarse E/C/S en un mismo módulo.
  - La cabecera debe diseñarse pensando en la posible aplicación a otros problemas.

- Debe minimizarse la cantidad de información o acciones a realizar antes de su llamada, para su correcto funcionamiento.

- Debe validar los datos devueltos.

- Si pueden producirse errores, debe informárselo al módulo llamador a través de un parámetro por referencia llamado, por ejemplo, *error*.

- No debe realizar efectos colaterales.
  La comunicación entre módulos se realiza obligatoriamente a través de los parámetros.

EJERCICIOS

1. Definir la función *Media* que devuelva la media aritmética de dos reales. Incluir también la función *Cuadrado* anterior. En el ***main***, calcular en una única sentencia el cuadrado de la media aritmética entre dos reales e imprimir el resultado.

2. Crear un procedimiento para que imprima en pantalla todos los divisores de un número entero.

3. Comprobar si un número es par.

4. Comprobar si un número es primo.

5. Calcular el MCD de dos enteros.

6. Calcular el MCM de dos enteros.

7. Leer un valor desde el teclado, obligando a que sea un positivo. Devolver el valor leído.

Bibliografía:

- Hernán Ruiz, Marcelo; Programación C; MP Ediciones; Argentina 2003; ISBN: 987-526-155-6

- Schildt, Herbert; Programación en Lenguaje C; McGraw-Hilll, México 1987; ISBN: 968-422-017-0

- Kernighan, Brian W. y Pike, Rob.; La Práctica de la Programación; Pearson Educación, México 2000; ISBN: 968-444-418-4

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Capítulo 6
Arreglos.

Tabla de contenido:

6.Arreglos.

6.1.Concepto de arreglo.

Los tipos de datos utilizados anteriormente como **int**, **float**, **char**, etc. Son considerados del tipo simple. Los arreglos se consideran como un dato de tipo estructurado.

Un arreglo es un conjunto de datos homogéneos que se encuentran ubicados en forma consecutiva en la memoria, un arreglo sirve para manejar una gran cantidad de datos bajo un mismo nombre o identificador.

Se dice que los datos utilizados en un arreglo son homogéneos por que sólo pueden ser del mismo tipo. Si se define un arreglo que contiene datos del tipo **float** todos los elementos del arreglo podran ser números reales.

Por ejemplo con la sentencia:

float calificacion [3];

Se reserva en memoria el espacio para cuatro variables de tipo real (**float**), las variables se llamarán **"calificacion"** y se accede a cualquiera de ellas a través de un *subíndice*. Los elementos se numeran desde cero hasta el número de elementos menos uno. El manejo de los subíndices para acceder a los elementos del arreglo se realiza generalmente con estructuras repetitivas como se verá más adelante en los programas resueltos. Esta forma para declarar un arreglo se llama *forma explícita*.

El espacio en memoria que ocupa un arreglo es igual al espacio que ocupa en memoria un elemento del tipo de dato (4 *bytes* en el caso de un **float**) multiplicado por la cantidad de elementos definidos en el arreglo. El arreglo "**calificacion**" ocupa en total (4 x 3) 12 *bytes* de memoria.

Es muy útil conocer la cantidad de memoria que ocupa un arreglo, para diseñar programas eficientes. Además, es importante que se utilicen todas las localidades de memoria definidas. Si se define un arreglo con cien localidades y sólo se utilizan seis en cada ejecución del programa, en la memoria se reservan siempre las cien localidades.

Si el programa nunca tiene necesidad de utilizar las cien localidades reservadas se puede decir que el código es ineficiente por el desperdicio de memoria que genera en cada ejecución. Cabe aclarar que durante la ejecución de un programa no se permite que ningún otro recurso de la computadora acceda o utilice esas localidades previamente reservadas.

En C, todos los arreglos constan de posiciones de memoria contiguas.

Nota: También se conoce a los arreglos como un conjunto de datos del mismo tipo donde el nombre del arreglo o vector apunta hacia la primer localidad de memoria. El concepto de apuntador o puntero se explica en el apéndice C de estas notas.

Adicionalmente a los conceptos de *arreglo* y *cantidad de memoria reservada* para un arreglo, se requiere establecer otros que son fundamentales para aprender a utilizar los arreglos en la programación. A continuación se explican algunos de estos conceptos adicionales:

La *memoria base* de un arreglo es la forma de nombrar a la dirección de memoria donde se alojará al primer elemento del arreglo, a partir de ahí según se explicó todos los elementos del mismo arreglo se almacenarán en localidades contiguas.

Los arreglos pueden ser de los siguientes tipos:

- **Unidimensionales**, de una dimensión. Los arreglos unidimensionales cuando contienen sólo números en su interior, se pueden denominar también vectores. Matemáticamente se pueden realizar en un programa las mismas operaciones básicas entre un vector y un escalar o las operaciones más utilizadas entre vectores. El desarrollo de ejemplos de vectores o arreglos unidimensionales, se realiza en el apartado 6.2. Cuando el arreglo unidimensional en lugar de números contiene caracteres, entonces se dice que es una cadena de caracteres. El manejo de caracteres en los arreglos es un caso especial que se detalla en el apartado 6.4.

- **Bidimensionales**, de dos dimensiones. Los arreglos de dos dimensiones que sólo contienen números, se utilizan en problemas de álgebra lineal y generalmente son llamados matrices. Algunas de las operaciones mas comunes entre matrices, se desarrollan en el apartado 6.3. Para efectos de estos apuntes no se realizan ejemplos de matrices bidimensionales con caracteres, sin embargo si es posible realizar programas con estas características, el lenguaje C tiene contemplada esta posibilidad.

- **Multidimensionales**, de tres o más dimensiones. Aunque en estos apuntes no se desarrollen ejemplos ya sea numéricos o con cadenas de texto que tengan tres o más dimensiones, es posible realizarlo con las mismas herramientas del lenguaje C que se utilizan en este capítulo. La razón por la que no se desarrolla este tema, es la poca funcionalidad que esto representa para problemas de programación.

6.2.Arreglos unidimensionales.

Un arreglo unidimensional es un tipo de datos estructurado que está formado de una colección finita y ordenada de datos del mismo tipo. Es la estructura natural para modelar listas o colecciones de elementos iguales.

Forma general para declarar un arreglo unidimensional en lenguaje C:

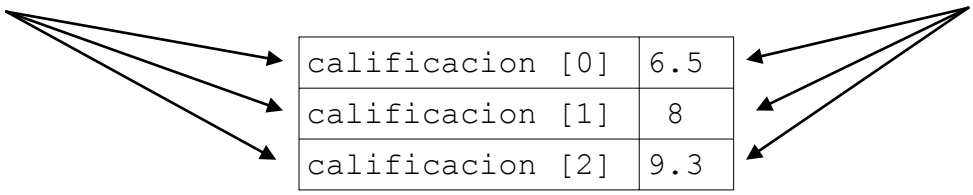
```
tipo_de_dato_de_arreglo   nombre    [cantidad_de_elementos];
```

Cuando se declara un arreglo con una línea como la anterior, el programa se encarga de reservar la *cantidad de memoria* necesaria para que el programa funcione adecuadamente, además el programa cuando se esta ejecutando define cuál será la *memoria base* del arreglo.

A continuación se muestra la forma de acceder a los elementos del arreglo "**calificacion**", esto se realiza de acuerdo a la posición del elemento que se expresa con un número llamado subíndice. Los subíndices deben escribirse entre corchetes [ ] también llamados paréntesis cuadrados.

En el siguiente esquema se observa el subíndice mediante el cual se accede a cualquier elemento del arreglo. Se reemplazó la localidad de memoria donde se encuentran los valores del arreglo. Nótese que un arreglo de tamaño tres, no tiene elemento [3] por que el conteo en los arreglos comienza desde cero.

Subíndice del elemento del arreglo: Contenido de las celdas de memoria:



Según el ejemplo en este arreglo se almacenan las calificaciones parciales de un estudiante, en el elemento **calificacion [0]** se almacenó el valor de 6.5 que corresponde a la primera calificación parcial que obtuvo el estudiante.

De igual manera el 8 almacenado en la ubicación del elemento **calificacion [1]** es la segunda calificación que obtuvo durante el curso. Por ultimo la cantidad 9.3 del elemento **calificacion [2]**, es la calificación que obtuvo el estudiante en su tercera evaluación parcial.

En el lenguaje C no se puede operar a un vector completo o matriz como si fuera una entidad única. La operación que se aplique al arreglo debe efectuarse para cada elemento del arreglo. Los ciclos o bucles que se utilizan generalmente para operar en todo el arreglo son del tipo **for** ó **while**.

A continuación se indican algunos ejemplos de operaciones que se pueden efectuar con elementos de un arreglo, dentro de algún programa:

```
     numeros [0] = 6;     // Esta línea asigna un 6
          // a la variable [0] de un arreglo llamado números.
```

```
     datos[2] = datos[1] – datos[0];    /* En este caso la variable
[2] del arreglo datos es igual a la diferencia entre las variables
[1] y [0] del mismo arreglo datos.
```

Según se observa, estas operaciones aplican de igual forma que cualquier variable de lenguaje C, la única diferencia es que se deberá hacer referencia a esos elementos por medio del subíndice correspondiente.

Solución de problemas con vectores.

A los arreglos unidimensionales, también se les denomina vectores. En programación los arreglos se diseñaron para realizar el manejo y diferentes operaciones entre vectores.

Se realizará un programa para aplicar los conceptos vistos sobre arreglos unidimensionales utilizando un arreglo de cinco elementos, el programa guardará los valores del vector según los datos que capture el usuario.

También se aplicarán los ciclos **for** y **while** en el ejemplo para enfatizar que es indistinto utilizar un ciclo u otro al operar las estructuras:

```
#include<stdio.h>   // ********** unidim.c ************
          // Programa que maneja un arreglo unidimensional
main() {  // Inicia la función main

        // Inicializa la variable y la matriz
  int i;
  float datos [5];

        // Captura de la matriz
  printf("Escriba los valores de la Matriz \n");
   for (i=0; i<5; i++)
        scanf("%f", &datos[i]);

        // Impresión de la matriz
  printf("los valores de la Matriz son: \n");
  i=0;
   while(i<5)   {
              printf("%f \n" , datos[i] );
              i++; }

     }  // Cierra la función main
```

Según se observa en el código, el programa utilizó un ciclo **for** para la captura de datos y un ciclo **while** para la impresión en pantalla, ambos ciclos pudieron ser **while** o **for**, incluso se pudo utilizar cualquier combinación de ellos.

La variable "**i**" del programa se utiliza como subíndice para acceder a los elementos del vector y "**datos**" es el nombre del arreglo que hemos utilizado.

Abajo del ciclo **for** no se utilizan las llaves **{ }** de bloque de instrucciones, porque el ciclo contiene una instrucción o sentencia única y de acuerdo a las especificaciones del lenguaje C, no es necesario utilizarlas en ese caso. Recuerde que las llaves **{ }** se utilizan cuando el ciclo debe repetir dos o más instrucciones.

Existe otra forma de crear e inicializar arreglos en lenguaje C y es la siguiente:

```
float  numeros_vector  [ ]  =  { 1.4 , 2.3 , 8.5 , 6.4 , 5. };
```

La instrucción anterior crea un arreglo que se llama "numeros_vector" y no se indica el tamaño del arreglo debido a que el compilador cuenta la cantidad de elementos del arreglo. Esta forma para declarar un arreglo se denomina la *forma implícita*.

Si se observa el último elemento de arreglo se ve el número cinco seguido de un punto sin fracción, esta es una forma de decirle al compilador que la cantidad se debe interpretar como un tipo de dato **float**. En otras palabras para algunos compiladores de lenguaje C:

$$5. = 5.0$$

Y ambos valores son del tipo **float**.

Las principales operaciones que se realizan a un vector son las siguientes:

- Mutiplicación por un escalar
- Suma de vectores
- Resta de vectores

La multiplicación de matrices bidimensionales se detallará en el tema 6.3. Por ahora se explica que esas operaciones no se realizan elemento a elemento como la suma y resta.

La multiplicación de un vector por un escalar o arreglo unidimensional se realiza de la siguiente forma. Supongamos que a un vector unidimensional lo vamos a multiplicar por un escalar. Indicaremos la operación de multiplicar con un ***** tal como se indica en programación. La operación se realiza de la siguiente manera:

Escalar: Matriz unidimensional: Resultado:

$$5 \; * \; \begin{bmatrix} 4 \\ 2.5 \\ 9 \\ 11 \\ -3 \end{bmatrix} = \begin{bmatrix} 20 \\ 12.5 \\ 45 \\ 55 \\ -15 \end{bmatrix}$$

De acuerdo a lo anterior la multiplicación de una matriz unidimensional por un escalar es igual a multiplicar el escalar por cada uno de los elementos de la matriz. De igual forma si se necesita dividir el vector por un escalar basta con dividir cada elemento del vector por el número (también se dice que esta operación de dividir una matriz entre un escalar equivale a la multiplicación de la matriz por el inverso del escalar).

A continuación se presenta el algoritmo que realiza la mutiplicación de un vector unidimensional por un escalar:

```
#include<stdio.h>   // *********** vecxesc.c ***********
                    /* Programa que multiplica un vector por un
                        escalar. */
main() {  // Inicia la función main

          // Inicializa las variables y la matriz
    int indice;
    float vector [5];
    float numero;

          // Lectura del vector
    printf("Escriba los valores del vector unidimensional \n");
     for (indice=0; indice<5; indice++)
          scanf("%f", &vector[indice]);

          // Impresión del vector original
    printf("los valores capturados del vector son: \n");
     for (indice=0; indice<5; indice++)
          printf("%.2f \n" , vector[indice] );

               // Captura del escalar para multiplicarse al vector
    printf("Escriba un número para multiplicar al vector \n");
    scanf("%f", &numero);

          // Impresión del vector multplicado
    printf("La matriz que resulta es: \n");
     for(indice=0; indice<5; indice++){
          vector[indice] =  vector[indice] * numero;
          printf(" Elemento [ %d ] = %.2f \n", indice,
                    vector[indice]);
                                        }
       }    // Cierra la función main
```

Para este ejemplo sólo se utilizaron ciclos **for**, la variable que accede a los elementos del arreglo es llamada en este ejemplo "indice" cabe aclarar que esta variable siempre debe ser del tipo **int**.

En el último ciclo **for** se encuentra la siguiente sentencia:

```
vector[indice] =  vector[indice] * numero;
```

Esta se encarga de multiplicar el escalar "**numero**" a cada elemento del arreglo que en esta caso fue denominado "**vector**", esta sentencia después de realizar la multiplicación y según su orden de prioridades asigna al elemento "**vector[indice]**" su nuevo valor.

En el caso de las operaciones suma, resta y multiplicación de vectores el algoritmo se resuelve de forma semejante. A continuación se indicará cómo se realizan las operaciones para el caso de la suma y después se generalizará la forma de aplicar el algoritmo:

Matriz1:		Matriz2:		Resultado:
12		5.2		17.2
11.9		1.3		13.2
- 4.5	+	9	=	4.5
0		4		4
16		- 16		0

Según se observa la suma de matrices se realiza elemento por elemento, es decir que el elemento

```
Resultado [ 0 ] = Matriz1 [ 0 ] + Matriz2 [ 0 ];
Resultado [ 1 ] = Matriz1 [ 1 ] + Matriz2 [ 1 ];
      .                .              .
      .                .              .
      .                .              .
Resultado [ 4 ] = Matriz1 [ 4 ] + Matriz2 [ 4 ];
```

Esta es la forma de obtener los valores de la matriz "Resultado", la forma usual para escribirlo dentro de un programa es:

```
Resultado [ i ] = Matriz1 [ i ] + Matriz2 [ i ];
```

Esta instrucción desde luego se encuentra en el interior de una estructura repetitiva **for** o **while** según prefiera o convenga al programador y la variable "**i**" es el contador del ciclo. Es bastante común encontrar que se utiliza la variable "**i**" como parte de una estructura repetitiva especialmente en arreglos unidimensionales.

A continuación se presenta el código de un programa que realiza las operaciones aritméticas básicas de suma y resta entre matrices unidimensionales.

```
#include<stdio.h>   // *********** sumresvu.c ***********
main() {  // Inicia la función main
          /* Programa que suma y resta vectores unidimensionales */
    int i;
    float Arr1 [5], Arr2 [5], Res [5];
               // Lectura del primer vector
    printf("Escriba los valores del primer vector \n");
     for (i=0; i<5; i++)
          scanf("%f", &Arr1 [i]);
               // Lectura del segundo vector
    printf("Escriba los valores del segundo vector \n");
     for (i=0; i<5; i++)
          scanf("%f", &Arr2 [i]);
               // Impresión de la suma de vectores
    printf("La suma de los dos vectores es: \n");
    printf("Arreglo 1 \t Arreglo 2 \t Resultado \n");
     for (i=0; i<5; i++){
         Res [i] = Arr1[i] + Arr2[i];
          // La línea anterior suma elementos
 printf("%.2f \t\t %.2f \t\t %.2f \n", Arr1[i], Arr2[i], Res[i]);
                                    }
    printf("\n");
               // Impresión de la resta de vectores
    printf("La resta del primer arreglo menos el segundo es: \n");
    printf("Arreglo 1 \t Arreglo 2 \t Resultado \n");
     for (i=0; i<5; i++){
         Resultado [i] = Arr1[i] - Arr2[i];
         // La línea anterior resta los elementos
  printf("%.2f \t\t %.2f \t\t %.2f \n", Arr1[i], Arr2[i], Res[i]);
                                    }
       }  // Cierra la función main
```

Según se observa en el programa anterior la única diferencia entre las operaciones de los vectores, es el operador "**+**" o "**-**" dentro de la estructura repetitiva **for**. Por eso se especificó anteriormente que el algoritmo entre ambas operaciones es semejante.

Se llama orden o tamaño de una matriz a las dimensiones del mismo y a su cantidad de renglones o columnas.

Dependiendo de la forma del vector o matriz unidimensional se le conoce como del tipo "columna" o "renglón" véase el siguiente ejemplo:

Vector columna o matriz columna:	Vector renglón o matriz renglón:
$\begin{bmatrix} 102 \\ 65 \\ 30.4 \\ -121 \\ 11 \\ 2.4 \end{bmatrix}$ El orden o tamaño es de 6 renglones por 1 columna.	$\begin{bmatrix} 5 & 3 & -2 & 0 & 1.82 \end{bmatrix}$ El orden o tamaño es de 1 renglón por 5 columnas.

Otra forma de expresar el orden de las matrices anteriores es:
para la matriz columna: 6R X 1C. Y para la matriz renglón: 1R X 5C. Donde las iniciales "C" y "R" significan columna y renglón respectivamente. Cuando se trabaja con matrices unidimensionales se sabe que alguna de sus dimensiones (ya sea renglón o columna) tiene tamaño uno. Este concepto de orden o tamaño cobra mayor relevancia en el siguiente tema.

Las columnas de manera semejante a una hoja de cálculo son las verticales y los renglones son en dirección horizontal. Una convención común al nombrar el tamaño de los arreglos es nombrar primero la cantidad de renglones y después la de columnas. A los renglones eventualmente se les llama filas.

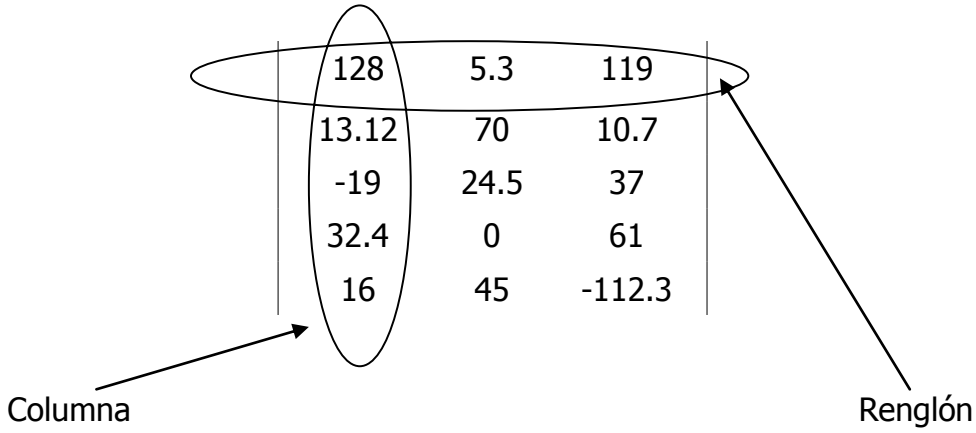
6.3.Arreglos Bidimensionales.

Los arreglos pueden tener más de una dimensión en ese caso se llaman arreglos multidimensionales, el caso más utilizado y sencillo de los arreglos multidimensionales se llama arreglo Bidimensionales.

Este es un tipo de arreglo que requiere de dos subíndices para acceder a cualquier elemento del mismo, de manera análoga a los arreglos unidimensionales también se les conoce a estos arreglos como matrices de dos dimensiones.

Matriz de dos dimensiones

El tamaño de esta matriz es: 5 Renglones X 3 Columnas.



Para efectos de programación supongamos que la matriz se llama “Arreglo”, la forma en que se declara la existencia del arreglo bidimensional es la siguiente:

float Arreglo [5] [3];

De forma semejante a la utilizada para declarar un arreglo unidimensional ahora se verá la forma general de declarar un arreglo bidimensional en lenguaje C:

```
tipo_datos_arreglo  nombre_arreglo  [cantidad_1] [cantidad_2];
```

Cabe aclarar que de igual forma, tal como ocurre con los arreglos unidimensionales, los datos se alojan en celdas contiguas de memoria. Internamente la computadora accede de forma secuencial a los elementos del arreglo. El procedimiento es el siguiente, el programa accede a la dirección base del arreglo y avanza a traves de las diferentes localidades del arreglo, hasta acceder al dato que necesita.

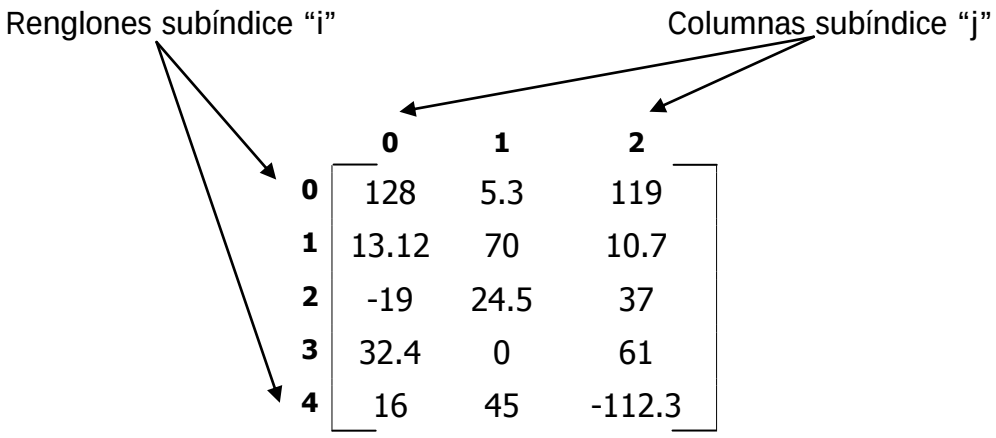
Regresando al arreglo previo podemos suponer que accederemos a los elementos de los renglones con la variable "i" y a los elementos de las columnas con la variable "j", de tal suerte que dentro de un programa accederemos a cualquier elemento de la matriz con una expresión del tipo:

Arreglo [i] [j]

Donde los valores de "i" y "j" son modificados por los ciclos de control del programa; Algunos ejemplos de operaciones que se pueden realizar sobre elementos de un arreglo bidimensional son:

```
Arreglo [ 2 ] [ 1 ] = Arreglo [ 1 ] [ 1 ] + 8;
Arreglo [ i + 1 ] [ j – 2 ] = Arreglo [ i ] [ j ] * 2;
Arreglo [ i ] [ j ] = 0;
```

Recordemos que la matriz "Arreglo" es de tamaño 5R X 3C, y accederemos a sus elementos con los subíndices "i" y "j" para renglón y columna respectivamente. Veamos la matriz "Arreglo" con los subíndices respectivos para identificar mas fácil a sus elementos:



Según se observa en el diagrama superior se puede identificar a cualquier elemento del arreglo por medio de la intersección de los subíndices, por ejemplo:

El valor 128 corresponde al elemento Arreglo [0] [0] y 61 corresponde a Arreglo [3] [2].

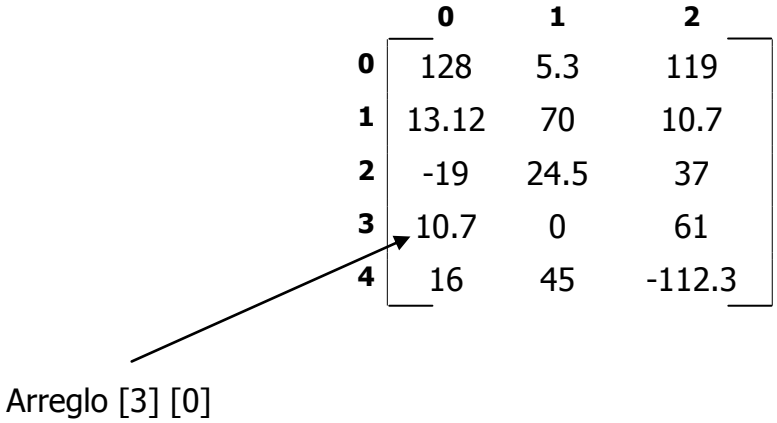
A partir de estos subíndices dentro de un programa se pueden ejecutar algunas expresiones que operen sobre elementos de los arreglos como las siguientes:

Arreglo [3] [0] = Arreglo [3] [1] + Arreglo [1] [2];

Sustituyendo el lenguaje C realizaría la siguiente operación:

Arreglo [3] [0] = 0 + 10.7;
Arreglo [3] [0] = 10.7;

La próxima vez que se imprima toda la matriz se vería de la siguiente forma:



El valor previo de Arreglo [3] [0] que era 32.4 fue sobrescrito por la nueva asignación que indicamos en el programa de tal forma que el nuevo valor de ese elemento del arreglo es 10.7.

Solución de problemas con matrices.

Las operaciones de suma y resta con matrices bidimensionales se efectúa de forma semejante que con las matrices unidimensionales, elemento a elemento como se verá en el siguiente diagrama:

$$\begin{vmatrix} 3 & 5 \\ -11 & 74 \end{vmatrix} + \begin{vmatrix} 16 & 9 \\ 5 & 4 \end{vmatrix} = \begin{vmatrix} 19 & 14 \\ -6 & 78 \end{vmatrix}$$

En este ejemplo se suman dos matrices de orden 2R X 2C y dan como resultado otra matriz del mismo tamaño.

Cuando un arreglo o matriz tiene la misma cantidad de renglones que de columnas, se dice que es una *matriz cuadrada*. En el caso de una matriz cuadrada como ésta, también se le puede denominar matriz de 2 X 2 o bien matriz cuadrada de orden 2.

En el caso de la resta entre matrices se realizaría de la misma forma que se hizo con la suma, sólo hay que cuidar las leyes de signos del Álgebra.

En cuanto a la multiplicación y división de matrices como se mencionó en el apartado de matrices unidimensionales, éstas no se realizan elemento a elemento del arreglo. La división de matrices o la operación que sería equivalente se obtiene a través de una matriz denominada inversa, esa matriz se obtiene a través de una serie de pasos o conversiones llamadas fundamentales que se aplican sistemáticamente en los renglones, existen algunos otros métodos matemáticos para obtener la matriz inversa y es tema de Álgebra Lineal. Ese análisis escapa a los alcances de este curso.

La multiplicación de matrices se realiza sólo en algunos casos y eso está en función del orden de las matrices que se desean multiplicar.

Por ejemplo tomaremos las matrices cuadradas del ejemplo anterior y las multiplicaremos para explicar la operación:

$$\begin{vmatrix} 3 & 5 \\ -11 & 74 \end{vmatrix} * \begin{vmatrix} 16 & 9 \\ 5 & 4 \end{vmatrix} = \begin{vmatrix} 3*16 + 5*5 & 3*9 + 5*4 \\ -11*16 + 74*5 & -11*9 + 74*4 \end{vmatrix}$$

Simplificando las sumas y los productos, tenemos que:

$$\begin{vmatrix} 3 & 5 \\ -11 & 74 \end{vmatrix} * \begin{vmatrix} 16 & 9 \\ 5 & 4 \end{vmatrix} = \begin{vmatrix} 73 & 47 \\ 194 & 197 \end{vmatrix}$$

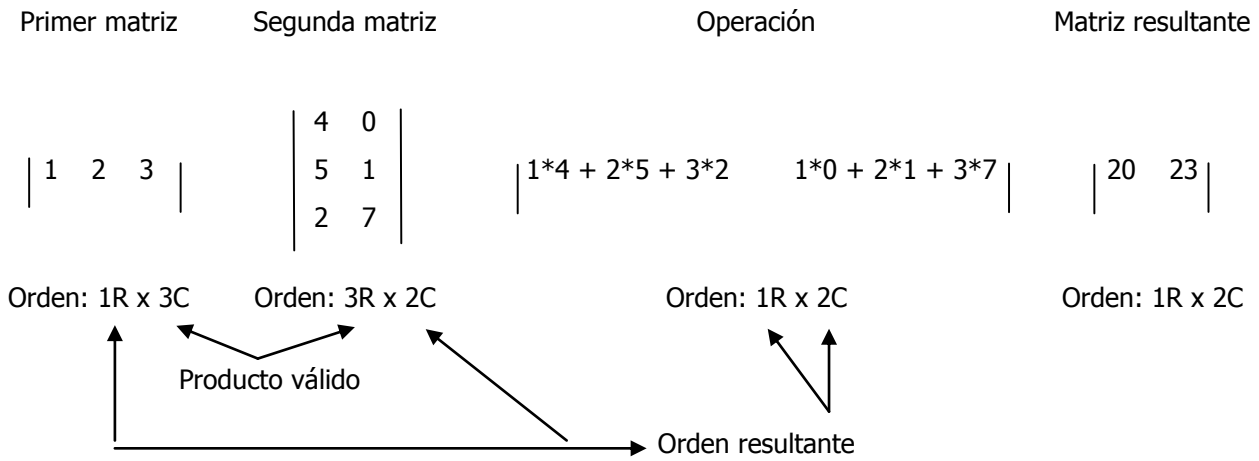
Si se observa con cuidado la multiplicación de matrices se realiza multiplicando cada renglón de la primera matriz por cada columna de la segunda matriz y sumando todas las multiplicaciones.

Gráficamente se vería de la siguiente forma:

$$\begin{vmatrix} 3 & 5 \\ -11 & 74 \end{vmatrix} * \begin{vmatrix} 16 & 9 \\ 5 & 4 \end{vmatrix} = \begin{vmatrix} 3*16 + 5*5 & 3*9 + 5*4 \\ -11*16 + 74*5 & -11*9 + 74*4 \end{vmatrix}$$

De aquí se desprende lo siguiente: **La multiplicación entre dos matrices es posible si la cantidad de columnas de la primera matriz es igual a la de renglones de la segunda matriz.**

En el ejemplo anterior se multiplicaron dos matrices cuadradas de igual orden 2x2. Veamos cómo aplica lo anterior con matrices no cuadradas de distinto orden.



En este caso el producto es posible debido a que la cantidad de columnas de la primer matriz es la misma cantidad de renglones que la segunda matriz. El orden de la matriz resultante será la cantidad de renglones de la primer matriz y la cantidad de columnas de la segunda.

Si tomamos las matrices del ejemplo anterior y las cambiamos de posición es decir la primera en el lugar de la segunda y viceversa, entonces tendríamos una matriz de orden 3 x 2 y la queremos multiplicar por otra de 1 x 3, la operación no se puede llevar a cabo, por las razones vistas anteriormente.

A continuación se presenta el código de un programa que multiplica matrices, este programa reserva espacio en memoria para multiplicar dos matrices con datos de tipo **float** de hasta 15 X 15 elementos y puede multiplicar matrices no cuadradas, siempre y cuando cumplan el requisito expresado antes.

Ejemplo complementario:

```
#include<stdio.h>   // ************ multmat.c ************

      // Multiplica matrices no cuadradas de dos dimensiones
main() {

    int i,j,k;           // Inicializa las variables
    int ren1, col1;      // variables para el tamaño de la matriz 1
    int ren2, col2;      // variables para el tamaño de la matriz 2

    puts("Programa que multiplica matrices no cuadradas");
                   // Indique el tamaño de la matriz 1
    printf("Indique el numero de renglones de la matriz 1 \n");
    scanf("%d" , &ren1);
    printf("Indique el numero de columnas de la matriz 1 \n");
    scanf("%d" , &col1);

              // Indique el tamaño de la matriz 2
    printf("Indique el numero de renglones de la matriz 2 \n");
    scanf("%d" , &ren2);
    printf("Indique el numero de columnas de la matriz 2 \n");
    scanf("%d" , &col2);

    float m1 [ren1] [col1];        // Inicializa la matriz 1
    float m2 [ren2] [col2];        // Inicializa la matriz 2
    float mR [ren1] [col2];        // Inicializa la matriz R

              // Verificación de la multiplicacion de las matrices
    if (col1 != ren2)
      printf("La multiplicacion no se puede realizar por el orden
              de las matrices \n");

      else {   // Inicia el caso "else" donde se multiplica
      printf("La matriz sera de %d Renglones X Columnas %d \n",
              ren1, col2);

               // Lectura de los arreglos
     printf("Escriba los valores de la primera matriz \n");
          for (i=0; i<ren1; i++){       //  i maneja renglones
                for (j=0; j<col1; j++){      // j maneja columnas
                    printf("Ingrese el elemento [%d] [%d] de la
                           Matriz 1\n", i,j);
                    scanf("%f", &m1 [i] [j]); }}
```

```
 printf("Escriba los valores de la segunda matriz \n");
 for (i=0; i<ren2; i++){       //  i maneja los renglones
      for (j=0; j<col2; j++){   // j maneja las columnas
printf("Ingrese el elemento [%d] [%d] de la Matriz 2\n", i,j);
           scanf("%f", &m2 [i] [j]); }}

           // algoritmo de la multiplicación de matrices
 for (i=0; i<ren1; i++){
      for (j=0; j<col2; j++){
                mR [i] [j] = 0;
                for (k=0; k<ren2; k++)
                     mR [i] [j] = mR [i] [j] +  m1 [i] [k] * m2
                                  [k] [j] ; } }

             //Impresión de la matriz final
      printf("La Matriz que resulta de multiplicar las Matrices
              1 y 2: \n");
      for (i=0; i<ren1; i++){
           printf(" \n");
           for (j=0; j<col2; j++)
                printf(" %.2f \t", mR [i] [j] ); }
             } // cierra el caso "else"
 printf("\n");
  }  // cierra la función main
```

Nota: Un arreglo de tipo estático, es aquel en el que la cantidad y tipo de elementos se declara desde el inicio del programa y nunca cambia durante su ejecución. Todos los programas de este capítulo, exceptuando "multmat.c" utilizan el concepto de arreglo estático.

La otra forma de declarar un arreglo es la forma dinámica, esto significa que, la cantidad de elementos del arreglo y el tipo de datos que contendrá se pueden definir durante la ejecución del programa. Esta característica resulta útil para evitar declarar arreglos que contengan en su interior demasiados elementos sin uso. Se observa la declaración del arreglo dinámico en el programa "multmat.c" cuando el usuario del programa define la cantidad de renglones y columnas de los arreglos que desea multiplicar.

La funcionalidad de manejar arreglos de tipos dinámicos, no pertenece a la versión original del lenguaje C, conocida como el ANSI C. esta funcionalidad viene de el estandar C89, el número 89 significa que esas mejoras se incorporaron al lenguaje desde el año 1989.

El lenguaje C sigue siendo bastante popular y difundido entre estudiantes y profesores de cursos de programación por sus importantes actualizaciones. A pesar del surgimiento de nuevos lenguajes que tienen mejores características y están basados en C. Posterior al estandar 89, el lenguaje C se modificó en el año 99, por esa razón este lenguaje continúa siendo vigente en muchos ámbitos de la informática como la investigación y programación de microcontroladores, entre otros.

6.4. Cadenas de texto.

Las cadenas de texto o caracteres son un tipo de arreglos unidimensionales y en su interior contienen unicamente datos del tipo char.

La declaración de arreglos en lenguaje C generalmente es del tipo explícito de esta forma el compilador le reserva espacio en memoria para todos los componentes que lo conforman. Recordemos que cada char ocupa un byte en memoria. Por ejemplo, supóngase que se declara el siguiente arreglo de caracteres:

```
char nombre [10];
```

Con esta línea el programador reserva diez bytes de memoria y el lenguaje C, se encarga de reservar un byte extra, que sirve para el caracter nulo '\0', también conocido como fin de cadena. De tal forma que cuando la instrucción anterior se ejecuta, el programa reserva en total once bytes. El manejo del último byte del arreglo pasa desapercibido por el programador.

Otra forma de declarar un arreglo de caracteres, sin indicar el tamaño es la forma implicita y según se observa, es semejante a la explicada anteriormente en este capítulo, cuando se declararon arreglos numéricos:

```
char nombre  [ ] =  { 'P' , 'a' , 'n' , 'c' , 'h' , 'o' };
```

Se puede suponer que la línea anterior cuando se ejecuta, reserva todos los bytes que conforman la cadena de caracteres “Pancho” más el byte del carácter nulo '\0'. Si el primer carácter de la cadena 'P' se aloja en la localidad de memoria 1E hexadecimal. En las localidades contiguas se almacenarán los caracteres siguientes. Cuando se hace referencia a una dirección de memoria en sistema hexadecimal, antes de escribir el número se le agrega “0X”. En la siguiente tabla se observa la forma en que se acomodarían los caracteres de la cadena en la memoria y su correspondiente dirección hexadecimal.

Caracter	Localidad de memoria en Hexadecimal
'P'	0x001E
'a'	0X001F
'n'	0X0020
'c'	0X0021
'h'	0X0022
'o'	0X0023
'\0'	0X0024

Según se observa en la última localidad se encuentra el caracter nulo '\0', tal como se había explicado. La asignación de caracteres y el orden que ocupan en el arreglo cuando se declara de forma explícita, equivale a las siguientes instrucciones:

```
nombre  [0] =   'P' ;
nombre  [1] =   'a' ;
nombre  [2] =   'n' ;
nombre  [3] =   'c' ;
nombre  [4] =   'h' ;
nombre  [5] =   'o' ;
```

Los caracteres del arreglo "**nombre [ ]**" van del subíndice cero al cinco y es en ese rango de valores del subíndice, en los que el programador puede realizar operaciones con los caracteres de la cadena. Estas operaciones se describirán más adelante en este capítulo y se relacionan directamente con dos bibliotecas del lenguaje C, <string.h> y <ctype.h>. Antes de conocer las funciones para manejo de caracteres que existen en esas bibliotecas, veremos las formas mas usuales para leer desde teclado e imprimir en pantalla una cadena de caracteres.

Entradas y salidas con arreglos de caracteres.

Desde luego los valores de un arreglo de caracteres, de forma semejante los arreglos numéricos, también se pueden leer, asignar o imprimir a través de estructuras repetitivas. Aunque no todo manejo de cadenas de caracteres requiere siempre del uso de estructuras repetitivas. A continuación se presenta el ejemplo de un programa que lee una cadena de texto tecleada por el usuario y manda a imprimir el arreglo y un caracter, sin utilizar estructuras repetitivas.

```
#include<stdio.h>   // ************ califi1.c ************

/* Inicia la función principal */

main(){
   char calificativo[35]; /* Se declara una cadena de texto que
                         puede almacenar hasta 35 caracteres */

   printf(" Escriba un calificativo y presione enter \n");
   scanf("%s", calificativo); /* Guarda la palabra escrita por el
   usuario en la cadena calificativo */

      /* En la instrucción Printf se manda a imprimir el texto
         y además un caracter  y una cadena */

    printf("EL Lenguaje %c me parece %s \n", 'C', calificativo);

}  // Cierra la función main
```

Según se observa en el código del programa con la instrucción:

```
scanf("%s", calificativo);
```

Se recibe del teclado una cadena definida por el tipo de dato *%s* (**String** o cadena de caracteres) y lo que se teclea en el teclado se almacena en la cadena denominada "calificativo", nótese que en este caso el caracter & que se utilizaba en ejericicos previos no es obligatorio, debido a que según se indicó en el inicio del capítulo el nombre de un arreglo o caracter es el apuntador a la primera localidad, así que el compilador identifica perfectamente a que localidad debe enviar los datos leídos desde teclado.

Esta forma de utilizar la instrucción ***scanf***, permite la lectura de palabras, sin embargo no admite espacios en blanco.

Obsérvese que cuando se manda a imprimir con la instrucción:

```
printf("EL Lenguaje %c me parece %s \n", 'C', calificativo);
```

Aparte del texto propio de la funcion **printf** también se imprime un dato del tipo *%c* (**Char** o caracter solo) y una cadena, que corresponde al tipo de dato *%s* (**String** o cadena de caracteres).

Si el usuario teclea "Interesante" el programa imprimirá:

```
EL Lenguaje C me parece Interesante
```

Sin embargo si el usuario teclea "muy bueno" el programa imprimira:

```
EL Lenguaje C me parece muy
```

La instrucción **scanf** terminará hasta que el usuario presione "enter" tal como indica el mismo programa, sin embargo cuando se detecte un espacio en blanco los caracteres serán omitidos en el arreglo. Por eso existe esa diferencia en la ejecución.

A continuación veremos un ejemplo, con una nueva versión del programa anterior, en el que se pueda capturar toda una cadena de caracteres incluyendo los espacios en blanco.

```
#include<stdio.h>   // ************ calific2.c ************

/* Inicia la función principal */

main(){
   char calificativo[35]; /* Se declara una cadena de texto que
     puede almacenar hasta 35 caracteres */

   printf(" Escriba un calificativo \n");
   scanf("%[^\n]", calificativo); /* Guarda la palabra escrita
     en la cadena calificativo incluyendo espacios en blanco*/

      /* En la instrucción Printf se manda a imprimir el texto
         además de un caracter y una cadena */

  printf("EL Lenguaje %c me parece %s \n", 'C', calificativo);

}  // Cierra la función main
```

Si revisamos el código la única diferencia se presenta en la línea:

```
scanf("%[^\n]", calificativo);
```

Esta forma de utilizar la función **scanf** se interpreta en el compilador como leer cualquier tipo de dato, hasta encontrar el caracter **\n**, que según se ha visto desde los primeros capítulos **\n** es la forma de indicar la tecla o caracter "enter".

Ahora se presentará un ejemplo donde la cadena de texto se imprime de dos formas distintas, utilizando estructuras repetitivas:

```
#include<stdio.h>   // ************ texto1.c ************

/* Inicia la función principal */

main(){
   int i;          // Se declara i para utilizar con un ciclo for
   char texto[20]; // Se declara una cadena de texto que puede
                   // almacenar hasta 20 caracteres
   printf(" Escriba su nombre y primer apellido \n");
   scanf("%[^\n]", texto); /* Guarda la cadena escrita por el
       usuario en la cadena texto incluyendo espacios en blanco*/

       // Imprime el texto con sólo un caracter por línea
  for(i=0; texto[i]!='\0'; i++) // Ciclo for es válido siempre que
   printf("%c \n", texto[i]);  // texto[i] no sea el caracter nulo

   // Imprime el texto separando los caracteres con tabuladores
  for(i=0; texto[i]!='\0'; i++) // Ciclo for es válido siempre que
printf("%c \t", texto[i]);   // texto[i] no sea el caracter nulo

}  // Cierra la función main
```

En este ejemplo sencillo, se observa la diferencia para imprimir las cadenas de texto, en el primer ciclo **for** el carácter **\n** que esta dentro de la instrucción **printf**, indica que después de cada caracter se imprima en el siguiente renglón.

En el caso del segundo ciclo **for**, se utiliza el caracter **\t** que indica la impresión de caracteres con un tabulador de distancia.

Para este ejemplo se pudo utilizar cualquier otra estructura repetitiva, no es necesario que siempre se utilice el ciclo **for**. A manera de ejemplo se presenta el mismo programa sólo que ahora utiliza la estructura **while** para la primera impresión de cadenas y utiliza **do-while** para la segunda impresión de caracteres:

```
#include<stdio.h>   // *************** texto2.c ***************

/* Inicia la función principal */

main(){
   int i;           // Se declara i para utilizar con un ciclo for
   char texto[20]; // Se declara una cadena de texto que puede
                   // almacenar hasta 20 caracteres

   printf(" Escriba su nombre y primer apellido \n");
   scanf("%[^\n]", texto); /* Guarda la cadena escrita por el
   usuario en la cadena texto incluyendo espacios en blanco*/

       // Imprime el texto con sólo un caracter por línea
       // Con while y la misma condicion de texto1.c

    i=0;   // Inicializa i en cero para while
       while(texto[i]!='\0'){         // Inicia estructura while
            printf("%c \n", texto[i]);
            i++; }                     // Termina estructura while

      // Imprime el texto separando los caracteres con tabuladores
       // Con do-while y la misma condición de texto1.c

    i=0;   // Inicializa i en cero para do-while
   do {                    // Inicia estructura do-while
            printf("%c \t", texto[i]);
            i++;
       } while(texto[i]!='\0');    // Termina estructura do-while
}  // Cierra la función main
```

En este programa se observa que es posible utilizar cualquier estructura repetitiva de igual forma que en los arreglos numéricos, la elección de la estructura depende del programa y lo que el programador observe mas conveniente.

Bibliotecas <string.h> y <ctype.h>.

Hasta este momento, las cadenas se han manipulado con las funciones **printf** y **scanf** únicamente. Estas funciones están incluidas en la biblioteca estándar de entrada y salida conocida como <*stdio.h*>, ahora se verán dos bibliotecas del lenguaje C, desarrolladas especialmente para manejo de cadenas de caracteres, estas bibliotecas son:

Biblioteca <string.h>

Esta biblioteca toma su nombre de la palabra "**string**" que en inglés significa cadena la extensión "**.h**" viene al igual que todas las bibliotecas de C, de la palabra inglesa "**head**" que significa literalmente "**cabeza***"* y para efectos de programación se entiende como "**encabezado**".

A continuación se presenta una tabla donde se indican algunas de las principales funciones desarrolladas para esta biblioteca.

Nombre de la función:	***Explicación:***
strcpy(cadena1 , cadena2)	Copia el contenido de la cadena2 en la cadena1.
strcat(cadena1 , cadena2)	Une la cadena2 al final de la cadena1, a esta operación se le llama concatenación.
strlen(cadena1)	Devuelve la longitud de cadena1 o expresado de otra forma indica la cantidad de caracteres que conforman a cadena1
strcmp(cadena1 , cadena2)	Esta función devuelve cero si cadena1 es igual a cadena2. Devuelve menor que cero si cadena1 < cadena2. Devuelve mayor que cero si cadena1 > cadena2.

Obsérvese que todos los argumentos de las funciones enlistadas anteriormente utilizan cadenas de caracteres.

Existen más funciones definidas en la biblioteca <string.h>, esas funciones adicionales utilizan el concepto de apuntador o puntero. Por esa razón escapan a los alcances de estos apuntes. Sin embargo, se pueden consultar en la bibliografía indicada al final de este capítulo.

Para compilar un programa que utilice la biblioteca <string.h> bajo el entorno del S.O. Linux se requiere agregar el parametro "-ls" al momento de compilar, de esa forma se liga la biblioteca <string.h> al código ejecutable.

Biblioteca<ctype.h>

Esta biblioteca toma su nombre de la unión de las palabras "character type" la traducción de estas dos palabras, podría ser "*tipos de caracteres"*. De la palabra "*character"* solo se toma la primera letra "*c* " y la palabra "*type"* se ocupa completa. Nuevamente la extensión de la biblioteca indica que son archivos de encabezado.

A continuación se presenta una tabla semejante a la anterior, donde se indican las principales funciones desarrolladas para la biblioteca <ctype.h>.

Nombre de la función:	***Explicación:***
isalnum(caracter1)	Devuelve un valor distinto de cero, si caracter1 es una letra del alfabeto o un dígito. Si el caracter no es alfanumérico la función devuelve cero.
isalpha(caracter1)	Devuelve un valor distinto de cero, si caracter1 es una letra del alfabeto. En caso contrario la función devuelve cero. Los caracteres considerados en el alfabeto son mayúsculas o minúsculas entre las letras a y z. En algunos compiladores no se consideran los caracteres latinos como "ñ" o las vocales acentuadas.
isdigit(caracter1)	Devuelve un valor distinto de cero si caracter1 es un dígito, es decir que esta comprendido entre los números 0 a 9. En caso contrario la función devuelve un cero.
islower(caracter1)	Devuelve un valor distinto de cero si caracter1 es una letra minúscula. En caso contrario devuelve cero.
isupper(caracter1)	Devuelve un valor distinto de cero si caracter1 es una letra mayúscula. En caso contrario devuelve cero.
tolower(caracter1)	Si caracter1 es una letra mayúscula la función devuelve el carácter equivalente en minúscula. En caso contrario la función no devuelve cambio en el carácter.
toupper(caracter1)	Si caracter1 es una letra minúscula la función devuelve el carácter equivalente en mayúscula. En caso contrario la función no devuelve cambio en el carácter.

Para compilar un programa bajo el entorno del S.O. Linux que utilice la biblioteca <ctype.h> de manera semejante a lo explicado para la biblioteca <string.h> se requiere agregar el parametro "-lc" al momento de compilar, de esa forma se liga la biblioteca <ctype.h> al código ejecutable.

En el siguiente aparatado se utilizarán alguna de las funciones, de las dos bibliotecas aquí descritas, para ver su aplicación práctica.

Solución de problemas con cadenas.

A continuación se presenta un ejemplo de código donde se cuenta la cantidad de caracteres que tiene una cadena tecleada por el usuario:

```
#include<stdio.h>   // ************ long.c ************
#include<string.h>

/* Inicia la función principal */

main(){

   char texto[20]; // Se declara una cadena de texto que puede
                   // almacenar hasta 20 caracteres

   printf(" Escriba una cadena de texto \n");
   scanf("%[^\n]", texto); /* Guarda la cadena escrita por el
    usuario en la cadena texto incluyendo espacios en blanco*/

    printf("La cadena tiene %d caracteres \n", strlen(texto));

}  // Cierra la función main
```

El programa anterior, se basa en la función "**strlen**" que proviene de la unión de las palabras inglesas "**string length**" que traducido significa "longitud de cadena". En negritas, se muestran las letras que se utilzan para formar la instrucción.

string **len**gth

La mayoría de las funciones mostradas en las dos tablas previas, toman su nombre de abreviaturas semejantes.

Según se observa en el código para utilizar la función "**strlen**" se requiere indicar al inicio del programa el encabezado <string.h>, que es donde se define la forma de operar para esa función.

Como una función del lenguaje C se debe tener cuidado en el manejo de los parámetros o argumento de la función, en este caso la función como argumento necesita una cadena de caracteres. Esta función esta diseñada para devolver un número de tipo entero, puesto que el conteo de caracteres de una cadena de texto siempre tendrá como resultado una cantidad entera.

Para continuar con la solución de problemas utilizando cadenas de caracteres, se presenta el ejemplo de un programa que permita al usuario, contar la cantidad de caracteres numéricos dentro de una cadena de texto.

```
#include<stdio.h>   // *************** numeros.c ***************
#include<ctype.h>

/* Inicia la función principal */

main(){
   int i;          // Declara la varible i que se utilizara en el ciclo for
   int cuenta=0;    // Declara e inicializa un acumulador llamado cuenta
   char cadena[20]; // Se declara una cadena de texto que puede
                    // almacenar hasta 20 caracteres

   printf("Este programa cuenta la cantidad de cifras numéricas en una cadena \n");

   printf(" Escriba una cadena de texto y presione enter al terminar \n");
   scanf("%[^\n]", cadena); /* Guarda la cadena escrita por el usuario
                           en cadena incluyendo espacios en blanco*/

     // el ciclo for pasa por todas las localidades de la cadena hasta \0
     for(i=0; cadena[i]!='\0';i++)
        if (isdigit(cadena[i])!=0) cuenta++;  // El if incrementa la variable
                                              // cuenta cada que encuentre un número

         // Se imprime el resultado de la cantidad de números en la cadena
   printf("La cadena capturada tiene %d numeros \n", cuenta);

}  // Cierra la función main
```

El programa anterior utiliza una estructura repetitiva semejante a los programas anteriores, que sirve para ingresar en cada localidad de la cadena. Posterior al ciclo "**for**" se utiliza una estructura selectiva simple "**if**", que sirve para verificar cuándo hay una ocurrencia de número en la cadena.

Sólo hay que recordar que la función "**isdigit**" envia cualquier valor distinto de cero al detectar un número y envía exactamente cero cuando el caracter evaluado no es numérico. Notese que el argumento de esta función es un caracter y no toda la cadena, por ello se utiliza el ciclo "**for**" y se evalua en cada caso el caracter **cadena[i]**.

A continuación se realizará un ejemplo donde se realice la concatenación de cadenas, empleando la función **strcat**.

```
#include<stdio.h>   // ************ concat.c *************
#include<string.h>

/* Inicia la función principal */

main(){

   char cad1 [] = { 'A' , 'B', 'C' }; /* cad1 se declara
                                         explícitamente */
   char cad2[20]; /* Se declara una cadena  cad2 de hasta 20
                     caracteres */

   printf("Este programa concatena dos cadenas de texto \n");

   printf(" Escriba la segunda cadena de texto \n");
   scanf("%[^\n]", cad2); /* Guarda la cadena escrita en cad2 */

        // Se imprime el resultado concatenar las dos cadenas
   printf("La concatencación de las cadenas es: %s \n",
          strcat(cad1 , cad2) );

}  // Cierra la función main
```

El ejemplo anterior lee la cadena "**cad2**" y la concatena con "**cad1**" que fue declarada de forma explícita al inicio del programa.

En este caso la función "**strcat**" se encuentra en el encabezado <string.h> y requiere de dos cadenas de caracteres como argumentos.

La operación concatenar en cadenas de texto es una forma de suma o unión de cadenas, donde la segunda cadena enviada como argumento a la función **strcat** se agrega al final de la primer cadena.

Ahora se realizará un programa que utilice ambos encabezados o bibliotecas: <string.h> y <ctype.h>

```
#include<stdio.h>   // *********** cuentcad.c ***********
#include<ctype.h>
#include<string.h>

      // Ejemplo de empleo de los dos encabezados para texto

main()
{              // Inicia main

     char original[100]; // Se declaran la cadena de texto
     int i;    // Variable i se utiliza en ciclo for
     int z=0;  // Se utiliza para contar caracteres alfanuméricos

     printf("Teclear un texto con MAYUSCULAS y minusculas \n");
     printf(" presione ENTER para finalizar \n");

           /* Guarda la cadena escrita en original */
     scanf("%[^\n]", original);

  /* el ciclo for pasa por todas las localidades de la cadena
original hasta \0  */
        for(i=0; original[i]!='\0';i++)
  // Este if verifica si es alfanumérico e incrementa z
            if (isalnum(original[i])!=0) z++;

// Imprime la cantidad de caracteres alfanuméricos de la cadena
         printf("La cadena tiene %d caracteres alfanumericos
                 \n",z);

       // Imprime la cantidad de caracteres de la cadena
         printf("La cadena tiene %d caracteres en total
                 \n",strlen(original));

}              // Termina main
```

Del encabezado <ctype.h> se utiliza la función "**isalnum**" y del encabezado <string.h> se utilza la función "**strlen**" como resultado este programa indica en una cadena cualquiera, cuántos caracteres alfanuméricos tiene (letras y números) y también indica su longitud total. Recuérdese que bajo el S.O. Linux se deben agregar los correspondientes parámetros al compilar.

Ejemplo complementario:

A continuación se presenta un programa que llama a la función “toupper” contenida en <ctype.h>. Esta función sirve para convertir una letra minúscula cualquiera, en su equivalente mayúscula, si el caracter en cuestión es mayúscula lo deja igual.

```
#include<stdio.h>   // *********** minamay.c ***********
#include<ctype.h>

       // Ejemplo de aplicacion ciclos FOR
       // para convertir de minúsculas a MAYÚSCULAS en cadenas
main()
{              // Inicia main

     char letras[100];
     int auxiliar, cont;

     printf("Teclear un texto con MAYUSCULAS y minusculas \n");
     printf(" presione ENTER para finalizar \n");

/* En este ciclo for se captura un arreglo que contenga menos de
  cien caracteres en el arreglo que se llama “letras” hasta que el
usuario presiona enter que es el caracter \n */

 for (cont=0; (letras[cont] = getchar()) != '\n'; cont++);
          auxiliar = cont;

                    // Impresión del texto en MAYUSCULAS

          for (cont=0; cont < auxiliar; ++cont)
          putchar(toupper(letras[cont]));
         printf("\n");

}              // Termina main
```

En este caso es importante observar que la condición del ciclo “*for*”, permite que el usuario capture una cadena de caracteres de cualquier tamaño (sin exceder 100 caracteres) utilizando una estructura repetitiva, hasta detectar el caracter '**\n**' (enter), obsérvese la condición:

(letras[contador] = getchar()) != '\n'

Utiliza la función **getchar()** que sirve para leer caracteres del teclado, esta condición se interpreta al momento de compilar como: Leer caracteres del teclado si son distintos de '**\n**' (enter) y cada carácter leido se almacena o asigna en **letras[contador]**

6.5 Serie de ejercicios propuestos

Nota: La aplicación de los siguientes ejercicios se deja a criterio del Estudiante y/o Profesor, estas son sugerencias de ejercicios que ayuden a fortalecer los conceptos expuestos, en estos apuntes.

1. Realice las siguientes multiplicaciones entre un escalar y matriz unidimensional:

$$3 * \begin{vmatrix} 12.6 \\ 9 \\ 5.2 \\ -4.01 \\ 75 \end{vmatrix} = \begin{vmatrix} \quad \\ \quad \\ \quad \\ \quad \\ \quad \end{vmatrix} \qquad -0.5 * \begin{vmatrix} 1.4 \\ 0 \\ 4 \\ -6 \\ 14 \end{vmatrix} = \begin{vmatrix} \quad \\ \quad \\ \quad \\ \quad \\ \quad \end{vmatrix}$$

$$-6 * \begin{vmatrix} 17 & 4 & 10.5 & 0.8 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad & \quad \end{vmatrix} \qquad 2 * \begin{vmatrix} 0 & 21 & -3.3 & 8.8 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad & \quad \end{vmatrix}$$

2. Realice las siguientes operaciones entre matrices unidimensionales:

$$\begin{vmatrix} -3 \\ 5.2 \\ 9 \\ 2.5 \\ -4 \end{vmatrix} + \begin{vmatrix} 11 \\ 7 \\ 3.5 \\ 8.3 \\ -5 \end{vmatrix} = \begin{vmatrix} \quad \\ \quad \\ \quad \\ \quad \\ \quad \end{vmatrix} \qquad \begin{vmatrix} 12 \\ -2 \\ 3.5 \\ 7 \\ 2 \end{vmatrix} - \begin{vmatrix} 5 \\ -6.3 \\ 3.5 \\ 0.5 \\ 4.8 \end{vmatrix} = \begin{vmatrix} \quad \\ \quad \\ \quad \\ \quad \\ \quad \end{vmatrix}$$

$$\begin{vmatrix} -16 & 7 & 3 & 10.5 & 0.8 \end{vmatrix} - \begin{vmatrix} 1 & 23 & 0 & -3.5 & 2.6 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad & \quad & \quad \end{vmatrix}$$

3. Realice las siguientes operaciones entre matrices bidimensionales:

$$2.5 * \begin{vmatrix} 24 & -14 \\ 12 & 6 \\ 3.2 & 5 \\ 0 & -3 \end{vmatrix} = \begin{vmatrix} \quad & \quad \\ \quad & \quad \\ \quad & \quad \\ \quad & \quad \end{vmatrix} \qquad \begin{vmatrix} 12 & -3 & 1 \\ 0.3 & 4 & 1.5 \\ 7 & 6.5 & 3 \end{vmatrix} - \begin{vmatrix} 3.5 & 9 & -1 \\ 8 & -6 & 0 \\ -14 & 3.5 & 9 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad \\ \quad & \quad & \quad \\ \quad & \quad & \quad \end{vmatrix}$$

Nota aclaratoria para resolver los siguientes ejercicios: todos los programas expuestos en este capítulo tienen su nombre escrito a manera de comentario en la primera línea del código.

4. A partir del código del programa “unidim.c” verifique los resultados del punto 2 de esta serie de ejercicios propuestos.

5. A partir del código del programa “vecxesc.c” verifique los resultados del punto 1 de esta serie de ejercicios propuestos.

6. Escriba un programa en lenguaje C que muestre un menú con las siguientes tres opciones:

a) La primer opción debe permitir que el usuario capture los valores de matrices bidimensionales de hasta cinco renglones y cinco columnas y las multiplique por un escalar e imprima la matriz resultante.

b) La segunda opción debe permitir que el usuario realice resta de matrices bidimensionales de hasta cinco renglones y cinco columnas, también se debe permitir al usuario capturar las matrices, efectuar la operación e imprimir la matriz de resultados.

c) La tercera opción debe terminar la ejecución del programa.

Utilice este programa para verificar sus operaciones del punto 3 de esta serie de ejercicios propuestos.

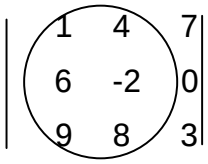
7. Realice manualmente las siguientes multiplicaciones de matrices si son posibles, en caso contrario explique por qué no se puede realizar el producto. A partir del código del programa “multmat.c” verifique los resultados de los productos de matrices.

$$\begin{vmatrix} 2 & 3 & 1 \\ 4 & -5 & 6 \\ 0 & 8 & 4 \end{vmatrix} * \begin{vmatrix} 1 & 0 & -3 \\ 1 & 2 & 4 \\ 3 & 5 & -7 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad \\ \quad & \quad & \quad \\ \quad & \quad & \quad \end{vmatrix}$$

$$\begin{vmatrix} 12 \\ 16 \end{vmatrix} * \begin{vmatrix} 7 & -9 \end{vmatrix} = \begin{vmatrix} \quad & \quad \\ \quad & \quad \end{vmatrix}$$

$$\begin{vmatrix} 3 & 6 & -2 \\ -1 & 2 & 6 \\ 4 & 6 & 3 \end{vmatrix} * \begin{vmatrix} 5 & 7 & 2 \\ -9 & 0 & 1 \end{vmatrix} = \begin{vmatrix} \quad & \quad & \quad \\ \quad & \quad & \quad \\ \quad & \quad & \quad \end{vmatrix}$$

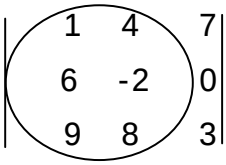
8. Escriba y ejecute un programa en lenguaje C que permita al usuario capturar una matriz cuadrada (cuando la cantidad de renglones es igual a la cantidad de columnas, por ejemplo: 3R x 3C) y que imprima los elementos de la diagonal principal de la matriz, abajo se muestra una matriz de 3 Renglones y 3 Columnas y con un óvalo se destacan cuáles son los elementos de la diagonal principal.



El programa podría imprimir en pantalla lo siguiente:

```
Los elementos de la diagonal principal son:  1  -2  3
```

9. Retomar el código del ejemplo anterior, para que el usuario capture una matriz cuadrada y en este caso se impriman los elementos de la diagonal secundaria. Abajo se muestra la diagonal secundaria de una matriz.



El programa podría imprimir en pantalla lo siguiente:

```
Los elementos de la diagonal secundaria son:  9  -2  7
```

10. Elabore un programa en lenguaje C que permita capturar una matriz cuadrada e imprima la transpuesta de la matriz, el símbolo que indica transposición de la matriz A es A^T. La transpuesta de la matriz A se observa a continuación:

$$\text{matriz } A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix} \qquad \text{Transpuesta } A = \begin{vmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{vmatrix}$$

11. Aplique el código del programa “minamay.c” y reemplace la función “toupper()” dentro del programa, utilice de forma análoga la función “tolower()”, ejecute el programa y observe que ocurre con las cadenas de caracteres, al ejecutar el programa combine mayúsculas y minúsculas al ingresar la cadena de caracteres.

12. Utilizando el código del programa texto1.c, modifíquelo para que utilizando la estructura repetitiva que prefiera, el texto se imprima de forma normal.

13. Modifique el código del ejercicio anterior para que utilizando la estructura repetitiva que prefiera, el texto se imprima al revés, es decir si el usuario teclea “Miguel Javier”. El programa debera imprimir: “reivaJ leugiM”

14. Desarrolle un programa que reciba un texto cualquiera e imprima un caracter en mayúscula y el siguiente en minúscula.
Por ejemplo si el programa recibe la cadena: “Martin Diaz” debera imprimir “MaRtIn dIaZ”

15. Desarrolle un programa que lea una cadena de texto en indique cuántos caracteres, son: mayúsculas, minúsculas, numéricos. Y también que indique el total de caracteres de toda la cadena.

Bibliografía.-

	Libro	Autor	Editorial	Edición	País	Año
1	C Manual de Referencia	Herbert, Schildt	Mc Graw Hill	Cuarta	España	2003
2	Programación en C.	Byron S. Gottfried	Mc Graw Hill	Segunda	España	2003
3	Programación estructurada en lenguaje C.	Leobardo López Román	Alfaomega	Primera	México	2005
4	Introducción al Álgebra Lineal	Howard Anton	Limusa Noriega	Tercera	México	1990

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice "A"
Comandos básicos de MS-DOS y Linux

Tabla de contenido:

1. Preámbulo

Este apéndice de referencia tiene como objetivos resumir los principales comandos de los sistemas operativos MS-DOS y Linux. No se pretende que el estudiante memorice los comandos de MS-DOS o de Linux, ya que finalmente en ambos sistemas operativos existe una ayuda, en donde ésta indica cómo funciona cada comando. En todo caso, lo que sí debe memorizar el estudiante es cómo consultar la ayuda de los comandos para cada sistema operativo: para MS-DOS se escribe **help** y para Linux se escribe **man**. Ejemplos:

```
MS-DOS
C:\> help dir

Linux
$ man ls
```

Este apéndice tampoco pretende ser un tutorial, pues en éste, el estudiante va aprendiendo siguiendo los pasos que se le indican.

Cabe mencionar que actualmente no se utilizan con mucha frecuencia los comandos de MS-DOS y Linux, ya que en la mayoría de los casos, se utiliza alguna herramienta gráfica que contiene opciones o pestañas que al activarse, equivalen a la ejecución de los comandos. Sin embargo, el conocimiento de los comandos es necesario, ya que habrá algunos que únicamente se ejecuten desde una sesión del "símbolo del sistema" para el caso del sistema operativo Windows.

2. Introducción al MS-DOS

MS-DOS son las iniciales de **Microsoft Disk Operating System** (Sistema Operativo de Disco de Microsoft), sistema operativo creado por la empresa Microsoft, que fue la precursora del ambiente gráfico Windows.

MS-DOS es un conjunto de programas que permiten manipular información contenida en discos duros y disquetes, además de que coordina el funcionamiento del microprocesador.

A lo largo del tiempo Microsoft mejoró el **MS-DOS**, desde su primera versión la 1.981, hasta una de las últimas que fue la 6.0. Las versiones que puedan existir, sirven para corregir errores detectados en la versión anterior o para mejorar ciertas partes del programa; si la variación está en la primera cifra, significa que se introdujeron cambios importantes, fruto, por ejemplo, de las prestaciones de los nuevos microprocesadores, en gran parte para aprovechar sus ventajas.

3. Lista de comandos principales de MS-DOS y Linux

MS-DOS	LINUX	Comentarios
help	man	Ayuda y explicación detallada de comandos
dir	ls	Lista los archivos y subdirectorios localizados dentro del subdirectorio actual
md	mkdir	Crea directorios
cd	cd	Cambia de subdirectorio
copy	cp	Copia archivos de un directorio a otro
move	mv	Mueve archivos y subdirectorios
del	rm	Borra archivos
rename	mv	Renombra los archivos o subdirectorios
rd	rmdir	Elimina subdirectorios
type	more	Ve el contenido de un archivo de texto
edit	vi, emacs, pico	Editores de texto
cls	clear	Borra la pantalla
attrib	chmod	Cambia los atributos a un archivo o subdirectorio
\	/	Directorio raíz
ver	uname -a	Despliega la versión: del sistema operativo para el caso de MS-DOS, y del kernel en el caso de Linux
mem	free	Estadísticas de la memoria principal
chkdsk, scandisk	fsck	Revisa y verifica el disco y sus archivos
format	mkfs	Formatea un disquete o un disco duro
----------	mount /dev/fd0 /mnt/floppy	Para montar un disquete
----------	mount /dev/cdrom /mnt/cdrom	Para montar un CD-ROM
a:	cd /mnt/floppy	Para direccionar al disquete
d:	cd /mnt/cdrom	Para direccionar al CD-ROM
----------	umount /mnt/floppy	Para desmontar un disquete
----------	umount /mnt/cdrom	Para desmontar un CD-ROM
Tecla "Impr Pant"	lpr archivo	Para imprimir una pantalla. En MS-DOS se usa la tecla "Impr Pant". En Linux se debe usar el comando lpr

4. MS-DOS

4.1 Comodines: '*', '?'

*

Este signo reemplaza cadenas de caracteres. En el ejemplo, el carácter * reemplaza el nombre de cualquier archivo. Asimismo, dichos archivos se listan con la extensión txt.
Ejemplo: `C:> dir *.txt`

?

Este otro signo reemplaza sólo un caracter. Se pueden especificar más signos de interrogación. Así pues, si se escribe `dir s???*.*` , se listarán los directorios y archivos con un nombre que empiece con letra s y otras 3 letras que pueden ser cualesquiera. A continuación, los caracteres **.*** hacen que se listen archivos con cualquier extensión.

4.2 Comandos

4.2.1 help

Comando que proporciona una ayuda extensa de cada comando. En versiones posteriores a la 5.0, la ayuda incluye ejemplos y notas.

Sintaxis: HELP, HELP [comando], comando/?

Ejemplos:

```
C:\> help

C:\> help cls

C:\> help copy
```

Otra forma de pedir ayuda es:

```
C:\> cls /?

C:\> copy /?
```

4.2.2 dir

Comando que es una abreviación de la palabra inglesa 'directory'.

Este comando permite ver todos los archivos y directorios de la ruta en la que se encuentre el apuntador.

Mediante una serie de parámetros se puede modificar ese listado. Este comando funciona como un filtro.

Sintaxis: `dir [unidad\directorio\archivo]`

Parámetros que se pueden especificar para que se listen los archivos y directorios de una forma concreta:

`/p` Con este parámetro se restringe la secuencia de listado y así detenerla hasta que se oprima una tecla. Al oprimir una tecla se procesará el siguiente bloque de listado y así sucesivamente. Este comando reparte internamente en bloques el número de archivos y directorios para luego desplegarlos.

`/n` Ordena por nombre

`/e` Ordena por extensión

`/s` Ordena por tamaño

`/d` Ordena por fecha

`/g` Ordena agrupando todos los directorios después de los archivos.

`/a:h` Lista los archivos cuyo atributo es h. En otras palabras, lista los archivos que están ocultos.

```
C:\UACM\2006> dir
C:\UACM\2006> dir *.c/p
C:\UACM\2006>dir /s
C:\UACM\2006> dir /a:h
```

4.2.3 md

Comando que se deriva de las palabras inglesas 'make directory' y su función es crear directorios.

Sintaxis: MD [unidad\ruta\]<nombre>
También puede ser mkdir.

Ejemplo:

```
C:\UACM\2006>md programas
C:\UACM\2006\programas>
```

4.2.4 cd

Comando que se deriva de las palabras inglesas 'change directory' y su función es cambiarse de un directorio a otro.

Sintaxis: `cd [unidad:]\[ruta]\[directorio]`
También puede ser `chdir`.

Si se desea retroceder un directorio no hace falta escribir la ruta, basta con escribir cd..

Ejemplos:
```
C:\UACM\2006\apuntes> cd..
C:\UACM\2006>

C:\UACM\2006\tutorados\jpz> cd \
C:\>
```
El apuntador se va al directorio raíz: C:\

El caracter \ se le llama “back slash”, siendo “slash” el carácter /

4.2.5 copy
Comando que se deriva de la palabra inglesa **copy** y su función es permitir la reproducción o copia de archivos. En este comando se pueden usar los comodines.

Sintaxis: `copy <archivo-origen> <archivo-destino>`

Ejemplo: `copy a:\art.txt c:\apunts\historia`
Copia el archivo art.txt de a: hacia el directorio c:\apunts

4.2.6 move
Comando que se deriva de la palabra inglesa **move** y su función es trasladar archivos de un directorio a otro. Internamente, este comando hace una copia del archivo al directorio especificado a mover, y luego borra el archivo de salida. En este comando se pueden usar los comodines.

Sintaxis: `move [/y] <origen> <destino>`
Donde /y es un parámetro que si se escribe, el comando **move** moverá archivos sin preguntar la confirmación de reemplazo sobre otros archivos que se puedan llamar de la misma forma en el directorio destino. En caso de no especificarse, MS-DOS preguntará la confirmación del reemplazo de archivos. A continuación se debe especificar el directorio de origen y el destino.

Ejemplo: `C:> move a:\art.txt c:\apunts\historia`
Mueve el archivo art.txt que está en la unidad a: hacia el directorio c:\apunts\historia

4.2.7 del

Comando que es una abreviación de la palabra inglesa **delete** y su función es eliminar o borrar archivos. En este comando se pueden usar los comodines.

Sintaxis: `del [unidad:]\[ruta]\[directorio]\<archivo>`

Ejemplos:

```
C:\UACM\2006> del apuntes.doc
C:\UACM\2006> del *.xls
C:\UACM\2006> del *.*
```

4.2.8 ren o rename

Comando que es una abreviación de la palabra inglesa **rename** y su función es asignarle un nuevo nombre a un archivo. No pueden existir dos archivos con el mismo nombre dentro de un mismo directorio. Ni MS-DOS ni Windows lo permiten. Sí se permite que existan dos archivos llamados de forma idéntica, pero que se encuentren en directorios distintos.

También se pueden usar en este comando los comodines.

Sintaxis: `rename <nombre-actual> <nombre-nuevo>`

Ejemplos:

```
C:\UACM\2006> ren *.txt   *.doc
C:\UACM\2006\programas> ren *.cpp   *.c
```

4.2.9 rd

Abreviación de **remove directory**: remueve directorio. Ejemplo:

```
C:\UACM\2006> rd programas
```

4.2.10 type

Comando cuya función es visualizar el contenido de archivos de texto en formato ASCII. No se permite el uso de comodines.

Sintaxis: `TYPE [unidad:]\[ruta]\[directorio]\<archivo>`

Por ejemplo TYPE readme.txt visualiza el contenido del archivo readme.txt. Si el archivo es más largo y no cabe en una página, no da tiempo a leerlo. En tal caso se añade el comando **more** precedido del símbolo | (ALT 124, del teclado numérico).

Cuando pasa una página se espera que se oprima una tecla para continuar.

Con este comando no pueden usarse los comodines y se debe señalar el nombre exacto del archivo.

Ejemplo:
`C:\> type autoexec.bat | more`

4.2.11 edit

Comando que se utiliza para editar archivos que contengan texto.

Sintaxis: `edit [unidad:]\[ruta]\[directorio]\<archivo.ext>`

Ejemplo: `c:\>edit autoexec.bat`

Con ésto, se abre el editor junto con el archivo de texto **autoexec.bat**

4.2.12 cls

Comando que es una abreviación de las palabras inglesas **clear screen** (limpiar pantalla) y su función es limpiar la pantalla. Sólo queda el directorio en el que se encuentra el apuntador, situado en la parte superior de la pantalla.

Sintaxis: cls

4.2.13 attrib

Comando que es una abreviación de la palabra inglesa **attribute** y cuya función es mostrar o modificar los atributos de los archivos.

Para visualizar los atributos de los archivos:
Sintaxis: `attrib /s`

Para modificar los atributos de los archivos:
Sintaxis: `attrib <archivo> <+/-><a/h/s/r>`

Atributos del comando attrib
th: atributo de invisibilidad
a: atributo de lectura-escritura
h: oculto (hide)
s: atributo de sistema (system)
r: atributo de solo lectura (read)

Signo más (+): establece atributo

Signo menos (-): quita atributo
t: activa un atributo
m: desactiva un atributo

4.2.14 ver

Comando que es una abreviación de la palabra inglesa **version** y cuya función es mostrar en pantalla la versión que se usa del MS-DOS.

Sintaxis: `ver`

Ejemplo:

```
C:\> ver
```

4.2.15 mem

Comando que es una abreviación de la palabra inglesa **memory** y cuya función es analizar la cantidad de memoria principal ocupada y disponible, para mostrarse en pantalla.

Ejemplo:

```
C:\> mem
```

4.2.16 chkdsk

Comando que es una abreviación de las palabras inglesas **check disk** y cuya función es revisar el disco.

Sintaxis: `chkdsk [unidad:] [archivo]`

Ejemplo:
```
C:\> chkdsk c: /f /v
```

Con lo anterior se comprueba que el disco duro no está defectuoso.

Se puede probar la unidad de disco que se desee. En el ejemplo se estableció la unidad C: , para realizar la prueba, es decir, el disco duro. Si no se escribe la unidad, el MS-DOS entiende que se desea hacer esta operación con la unidad activa.

El MS-DOS preguntará en algún momento "¿Convertir unidades de asignación perdidas en archivos FILEnnnn.CHK ? (S/N)". Si se responde "S" el programa reunirá los datos perdidos (posibles fallos de disco) y los guardará en diferentes archivos de nombre

FILE0000.CHK, FILE0001.CHK …, que se encontrarán esparcidos por el disco duro (y que luego se podrán eliminar). Si se responde a la pregunta con la letra "N", el programa corrige los fallos, eliminando las unidades de asignación perdidas. Generalmente se responde "N".

4.2.17 scandisk

Comando que comprueba la integridad de los datos almacenados basándose en el estado del disco que almacena estos datos.

Sintaxis: `scandisk`

Ejemplo:
```
C:\> scandisk c:
```

4.2.18 format

Comando que da formato lógico a una unidad física. Divide la superficie en sectores y pistas. Se pierden todos los datos almacenados en disco.

Sintaxis: `format unidad [/s] [/q] [/u] [/b] [/v[:etiqueta]] [/f:tamaño] [/t:pistas /n:sectores]`

/s: Transfiere archivos de sistema en disquete con formato (io.sys, msdos.sys y command.com).
/q: Realiza un formateo rápido.
/u: Realiza un formateo incondicional.
/b: Asigna espacio en disco con formato para archivos de sistema: io.sys y msdos.sys
/v [:etiqueta]: Se especifica la etiqueta de volumen.
/f: tamaño: Especifica el tamaño del disquete al que se dará formato (tal como 160, 180, 320, 360, 720, 1.2, 1.44, 2.88).
/t: pistas: Especifica el número de pistas por cara de disquete.
/n: sectores: Especifica el número de sectores por pista.

Formatear un disquete situado en la unidad A (insertar un disquete en la unidad respectiva):

Ejemplo:
```
C:> format a:
```

Responder a la pregunta que se formula oprimiendo la tecla de **return**, y observar cómo la luz de la unidad del disquete se enciende.

5 Linux

5.1 Comandos

5.1.1 man

Es el manual en línea de Linux, el cual contiene una serie de archivos con la documentación sobre cada uno de los comandos de Linux. Ejemplos:

```
$ man ls

$ man rmdir
```

5.1.2 ls

Comando que es una abreviación de la palabra inglesa `list`. Este comando permite ver todos los archivos y directorios de la ruta en la que se encuentre el apuntador.

Mediante una serie de parámetros se puede modificar ese listado. Este comando funciona como un filtro.

Sintaxis: `ls [/directorio/archivo]`

Ejemplos:

```
$ ls
```

Despliega un listado de archivos y directorios

```
$ ls –la
```

Muestra toda la información de cada archivo del directorio actual

Sintaxis: `ls [/directorio/archivo]`

Las opciones más comunes son:

-d lista de directorios
-l Formato de listado largo
-m lista de archivos separados por comas
-x Clasifica los nombres de archivos de manera horizontal
-A Lista todos los archivos
-C Clasifica los archivos de manera vertical
-F Indica directorios, enlaces y ejecutables
-R Lista contenidos de directorios de manera recursiva
-S Clasifica los archivos por tamaños

5.1.3 mkdir

Comando que se deriva de las palabras inglesas **make directory** y su función es crear directorios.

Sintaxis: `mkdir /ruta/<nombre>`

Ejemplo:

```
$ mkdir programas
```

5.1.4 cd

Comando que se deriva de las palabras inglesas `change directory` y su función es cambiarse de un directorio a otro.

Linux, al igual que la mayor parte de los sistemas operativos guarda los archivos en una estructura en árbol, asignándoles un nombre a cada una de las ramas de los directorios. El directorio raíz del cual se cuelgan los directorios, queda especificado con el caracter /, la barra de dividir, distinta a la del MS-DOS que como se sabe, tiene la inclinación contraria: \. Así, para acceder al directorio /home/root se teclea cd /home/root. El directorio actual se representa con el caracter **.** y el anterior (padre) con **..** . Así para acceder al directorio anterior se teclea **cd ..** (notar que existe un espacio en blanco, cd.. no es válido). Ejemplo:

```
$ cd /
```

El apuntador va al directorio raíz.

```
$ cd /home/laboratorio
```

El apuntador cambia al directorio laboratorio.

También, cd se usa para cambiar al directorio de trabajo actual. Si no se especifica directorio, el usuario se regresa a su directorio personal.

Para ir al directorio personal:

```
$ cd
```

Para ir al directorio padre:

```
$ cd ..
```

5.1.5 cp

Comando que se deriva de la palabra inglesa **copy** y su función es permitir la reproducción o copia de archivos. En este comando se pueden usar los comodines.

Sintaxis: `cp <archivo-origen> <archivo-destino>`

Ejemplo:

```
$ cp file1 file2
```

Copia file1 en file2. Para copiar múltiples archivos en una nueva ubicación, se puede usar comodines, como sigue:

```
$ cp file* /temp
```

Se debe usar **cp** con precaución, ya que no avisa si se va a sobrescribir un archivo ya existente. Si se quiere tener el control de sobrescritura, se debe utilizar la opción -i, que avisa si se va a sobrescribir un archivo:

```
$ cp –i file1 file2
```

cp: overwrite “file2”? y

En el ejemplo anterior file2 existía, así que el comando **cp** avisa y pregunta que si se desea sobre-escribir dicho archivo.

Para realizar copias de seguridad de los archivos que se sobrescriben, se debe utilizar la opción -b. Cada archivo copia de seguridad tendrá una ~ (tilde) agregada a su nombre.

La opción -p (parent, padre), junto con la opción -R (recursive, recursivo), no sólo copia los archivos de un directorio a otro, sino que también copia todos los directorios que hay dentro de ese directorio.

```
$ cp –Pr  dir1 dir2

$ cp –r  dir1 dir2

$ cp –R dir1 dir2
```

Todos los ejemplos anteriores copian el directorio dir1 y todos los archivos o directorios que hay dentro de él hacia el directorio dir2.

El mandato **cp** tiene más de 40 opciones diferentes y pueden consultarse en el manual del comando.

5.1.6 mv

En Linux, mover archivos tiene el mismo resultado que copiar los archivos en un nuevo directorio y a continuación borrar los archivos del directorio anterior. A diferencia de **cp**, **mv** no deja copia del archivo original.

Sintaxis: `mv archivo_origen  a_directorio/archivo_destino`.

La inclusión de archivo_destino en la sintaxis se debe a que este comando también se usa para renombrar un archivo, ejemplo: `mv hola.txt adios.txt`

El comando **mv** debe usarse con precaución, ya que no avisa de si se está rescribiendo un archivo o directorio ya existente. Para controlar lo que se está haciendo, es necesario el uso de la opción **-i** (*interactive*, interactivo), que obliga al comando a pedir permiso para realizar su trabajo en el caso de encontrar un archivo o directorio con el nombre que se ha dado como destino.

```
$ mv -i file1 file2

mv: replace "file2"? y
```

Si se acompaña el mandato **mv** con la opción **-b** (*backup*, copia de seguridad), se creará una copia de seguridad del archivo que se va a sobrescribir. La combinación de **-i** y **-b** pide permiso y además crea una copia de seguridad.

```
$ mv file1 file2

mv: replace "file2"? y

$ ls file*
```

file2 file2~

También se puede usar **mv** para mover archivos a una nueva ubicación:

```
$ mv file1 /temp
```

Si, además, se quiere cambiar de nombre en la misma operación, basta añadir el nuevo nombre del archivo:

```
$ mv file1 /temp/file2
```

Con esta técnica también se pueden mover directorios:

```
$ mv dir1 dir3
```

Si el directorio de salida (dir3) no existe, **mv** cambia el nombre del directorio. Si ya existe, todo dir1 se mueve adentro de dir3.

5.1.7 rm

Comando que se utiliza para eliminar archivos. Un archivo borrado en Linux, desaparece. Una vez borrado, su recuperación no es posible.

```
$ rm file1
```

5.1.8 rmdir

Borra el directorio con todo lo que tenga dentro sin preguntar. Este comando sólo elimina directorios vacíos. En realidad lo que se utiliza es el comando rm con la opción -r, que elimina el directorio especificado y todo su contenido, incluidos subdirectorios.

```
$ rmdir directorio1
```

5.1.9 more

Es un programa paginador que se utiliza para desplegar un archivo. Su característica distintiva es que despliega un archivo completo pantalla por pantalla.

Después de desplegar una pantalla de datos, este programa despliega el indicador:
–-More –

para mostrar que aún quedan datos por aparecer. La traducción de **more** es más. Por ejemplo, si se desea desplegar el contenido de un archivo llamado memo, se escribiría lo siguiente:

```
$ more memo
```

Consejo: no adquiera el mal hábito de usar el programa **cat** para desplegar archivos. Para ello use **more**.

Con la opción **–c** (borrar) hace que more despliegue cada pantalla de datos nueva de arriba hacia abajo. Cada renglón se borra antes de ser reemplazado. Sin **–c**, los renglones nuevos surgen desde abajo de la pantalla. Ejemplo:

```
$ more -c memo
```

Se puede usar **+** (signo más) seguido de un número para hacer que more comience a desplegar la información desde ese renglón. Verbigracia, para desplegar el contenido del archivo memo a partir del renglón 41 y que éstos aparezcan de arriba hacia abajo, se

debe escribir:

```
$ more –c +41 memo
```

5.1.10 clear

Comando que significa limpiar pantalla. Sólo queda el directorio en el que se encuentra el apuntador, situado en la parte superior de la pantalla. Ejemplo:

```
$ clear
```

5.1.11 chmod

Abreviación de la palabra change mode (cambiar modo). Comando que se utiliza para cambiar los permisos de un archivo. Ejemplos:

```
$ chmod 644 memo1 memo2 documento
```

El comando anterior asigna permiso de lectura y escritura al dueño del archivo y permiso de lectura al grupo y a todos los demás. Con estos permisos todo mundo puede leer el archivo pero no modificarlo.

```
$ chmod 755 tarea.especial
```

El comando anterior proporciona al dueño del archivo permisos de lectura, escritura y ejecución; permisos de lectura y ejecución para el grupo y para todos los demás. Con estos permisos otras personas ejecutan el archivo pero no lo modifican.

5.1.12 find

Posee un extraordinario conjunto de opciones que le dan una gran funcionalidad. Aquí nos restringiremos al uso más común.

Sintaxis: find directorio_inico_búsqueda-name "archivo"

Ejemplo: Buscar todos los archivos que comiencen por z minúscula desde el directorio raíz (todo el disco duro):

```
$ find / -name "z*"
```

5.1.13 pwd

Para imprimir el directorio de trabajo vigente, se utiliza **pwd** (**print working directory**).

```
$ pwd
```

5.1.14 ping

Este comando se utiliza mucho en internet y sirve para saber si una computadora está enlazada con otra a través de una red ó de internet. Por ejemplo, si un usuario está trabajando con su navegador en internet y accede a la página de Yahoo México y no aparece la información, puede realizar un ping a la dirección de Yahoo México, o a su dirección IP respectiva.

```
$ ping mx.yahoo.com

$ ping 69.147.91.167
```

Si la computadora no se enlaza, se verá un mensaje como el siguiente:
ping: unknown mx.yahoo.mx

¿Qué significa ping? En una nave, un ping (comprobación) es un impulso de sonar que se envía para que se refleje en otra nave. Por analogía, existe un comando que envía una interrogación electrónica para verificar el estado de otro sistema. El nombre de ese comando es ping. Oficialmente, el nombre significa "**Packet Internet Groper**" (Escudriñador de Paquetes de Internet).

5.1.15 Editores de texto

Existen editores de texto ó párrafo tales como: vi, emacs, pico
Se pueden llamar de la siguiente forma:

```
$ vi

$ emacs

$pico
```

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice “B”
Compilar y ligar un programa en Windows

Tabla de contenido:

Compilar y ligar un programa en Windows.

Para ligar y compilar un programa en lenguaje C, existen diferentes maneras de hacerlo y estas se encuentran en función del Sistema Operativo y compilador que se usará. Las explicaciones que se presentan a continuación, aplican para el S.O. **Windows** y con el compilador **Dev-Cpp**, Si la compilación se realizará bajo el S.O. **Linux** y el compilador **gcc**. Ver el apéndice **F** de estos apuntes. Las imágenes de este apéndice fueron tomadas del programa Dev-C++ versión 4.9.9.2, este es un software libre para desarrollo de aplicaciones en lenguaje C y C++.

Este entorno de desarrollo permite realizar programas en lenguaje C y C++; además de que es un programa que trabaja en un ambiente agradable para el usuario y de fácil manejo.

Para abrir tu entorno de desarrollo, tendrás que dar doble clic al icono:

Imagen 1 icono DevCpp

Posteriormente se presentarán dos ventanas de apertura del programa, una ventana de presentación del software, ver imagen 2 y otra con una ventana, la cual contiene comentarios sobre algunos elementos y herramientas del entorno de desarrollo de DevCpp, esta última la podrás cerrar oprimiendo el botón **close** en imagen 3.

Imagen 2 ventana de presentación

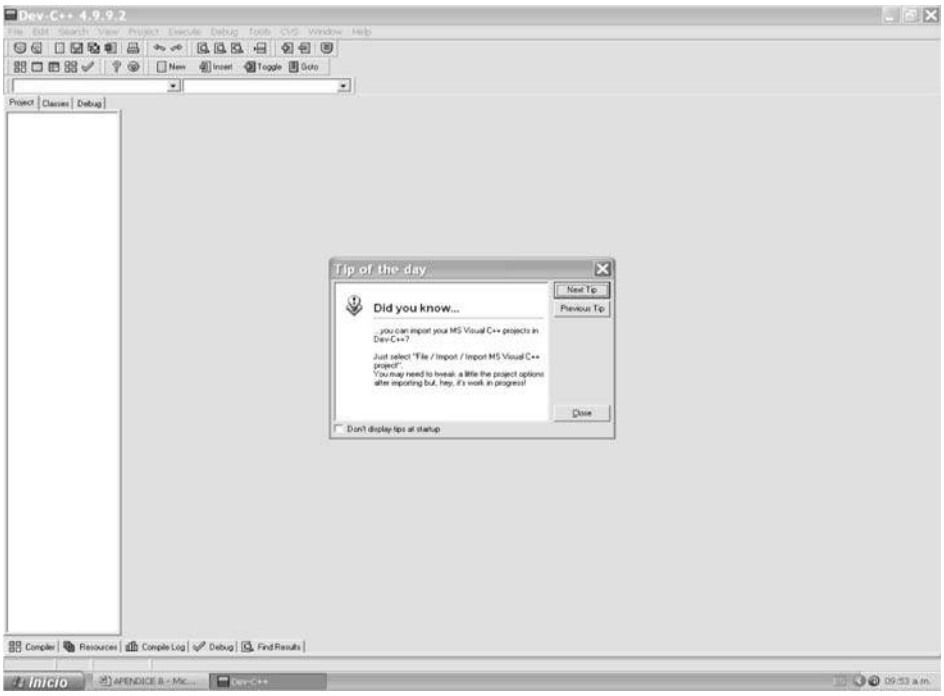


Imagen 3 ventana de comentarios

Una vez cerrada la ventana de comentarios, se tendrá que abrir el programa se muestra en la imagen 4.

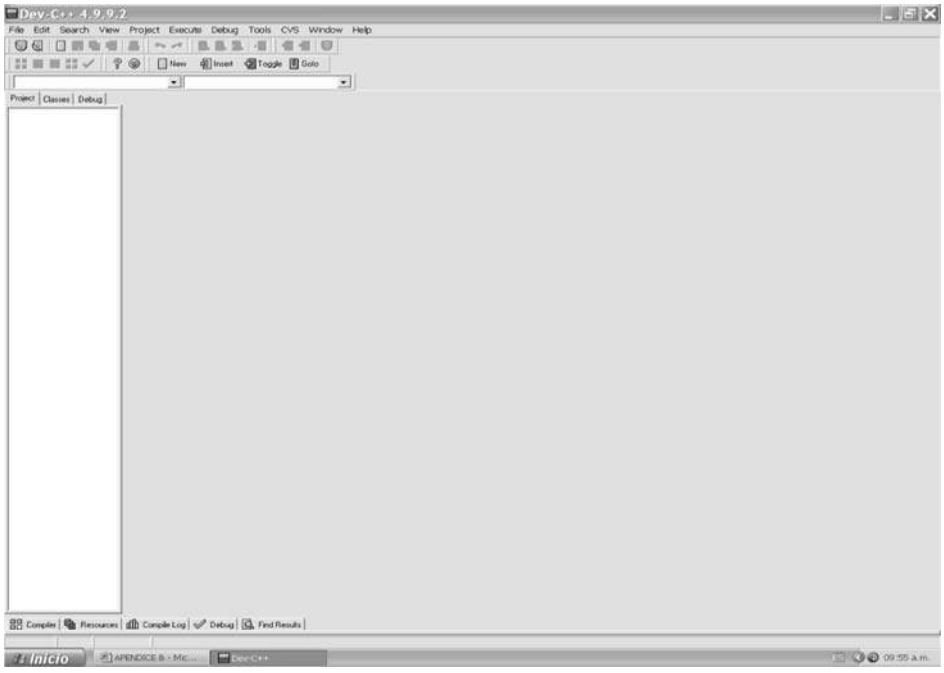


Imagen 4 listo el programa para abrir un archivo nuevo

Para abrir el programa hay tres opciones:

a) La primera es oprimiendo los botones **Ctrl** y **N** simultáneamente ver imagen 5.

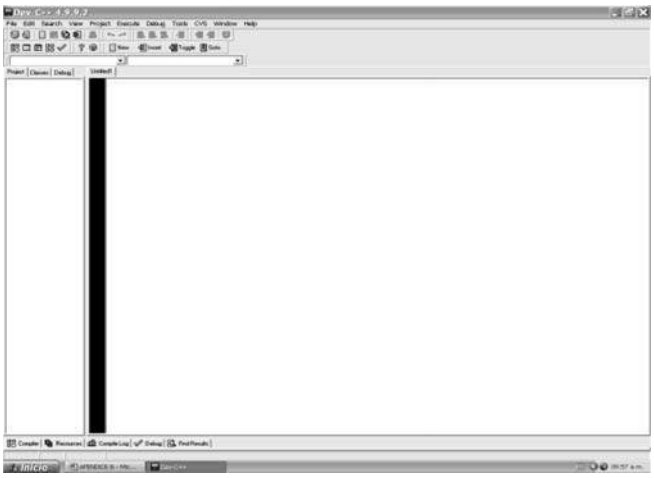


Imagen 5 ventana de inicio de archivo

b) La segunda es oprimiendo el botón **New** ubicado en la barra de herramientas de la ventana del programa, el cual mostrará un menú en donde se escogerá la opción **Source File** mostrada en imagen 6. Para abrir la ventana de inicio de archivo, ver imagen 5.

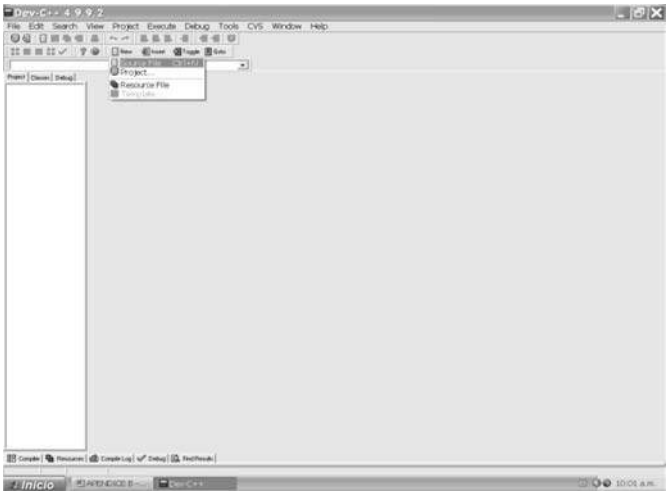


Imagen 6 mismo procedimiento

c) La tercera opción es oprimiendo el botón **File**, el cual desplegará un menú, en donde se escogerá la opción **New**, y éste a su vez desplegará un submenú en donde se seleccionará **Source File** como en imagen 7 y se abrirá la ventana de inicio de archivo ver imagen 5.

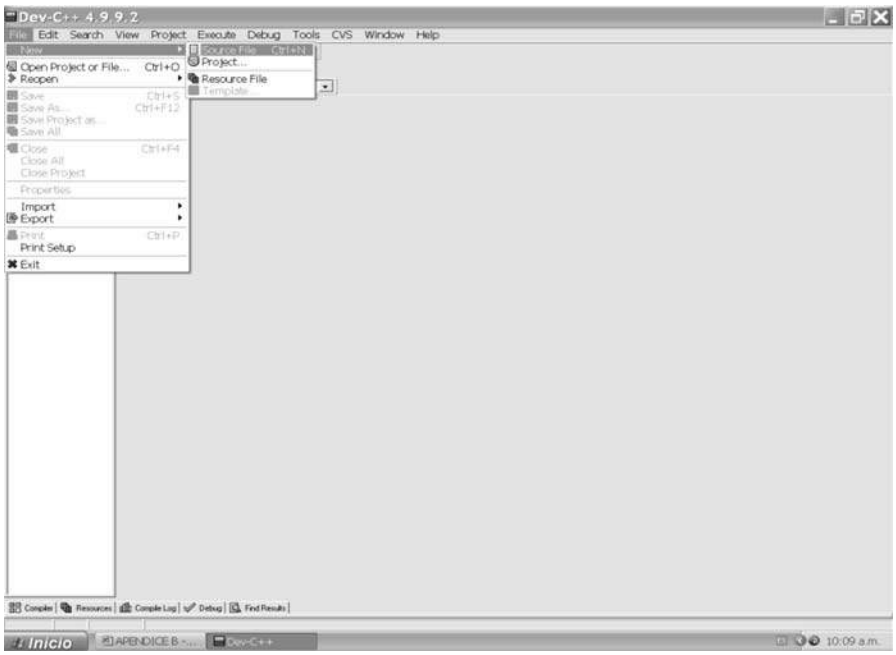


Imagen 7 mismo procedimiento

Una vez abierto el software se podrá comenzar a trabajar en el desarrollo de programas, ver imagen 8.

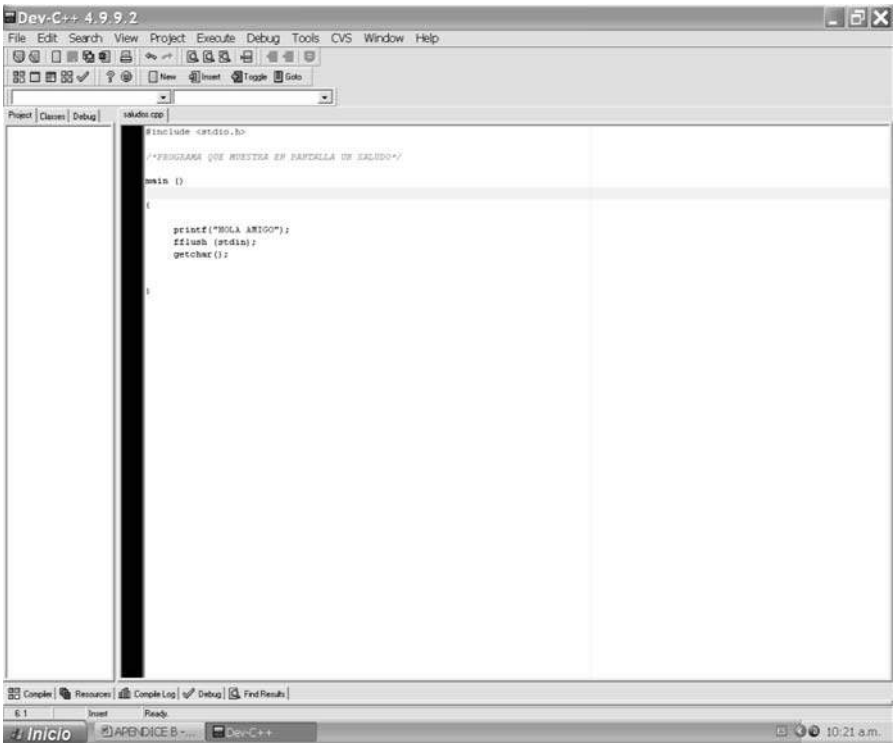


Imagen 8 entorno de desarrollo de programas

Para poder ejecutar tu programa primeramente tendrás que compilar para ver si la sintaxis del programa es correcta y así se pueda correr sin problema.

Para compilar hay tres opciones; pero antes tendrás que guardar tu programa, ya que si no lo haces cuando compiles se presentará la ventana de Guardar como o **Save as**. Por lo tanto para guardar tu archivo oprimirás la opción **File** de la barra de menús y seleccionarás la opción **Save as**, ver imagen 9.

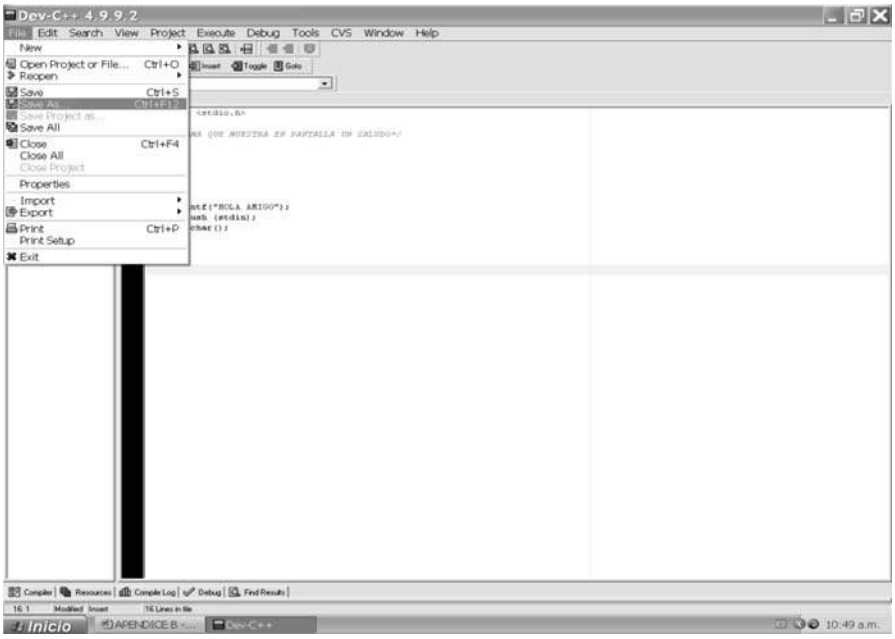


Imagen 9 mismo procedimiento

Después aparecerá la ventana de **Save as** o Guardar como, donde tendrás que poner el nombre de tu programa, posteriormente oprimirás el botón guardar para realizar dicha función como en imagen 10.

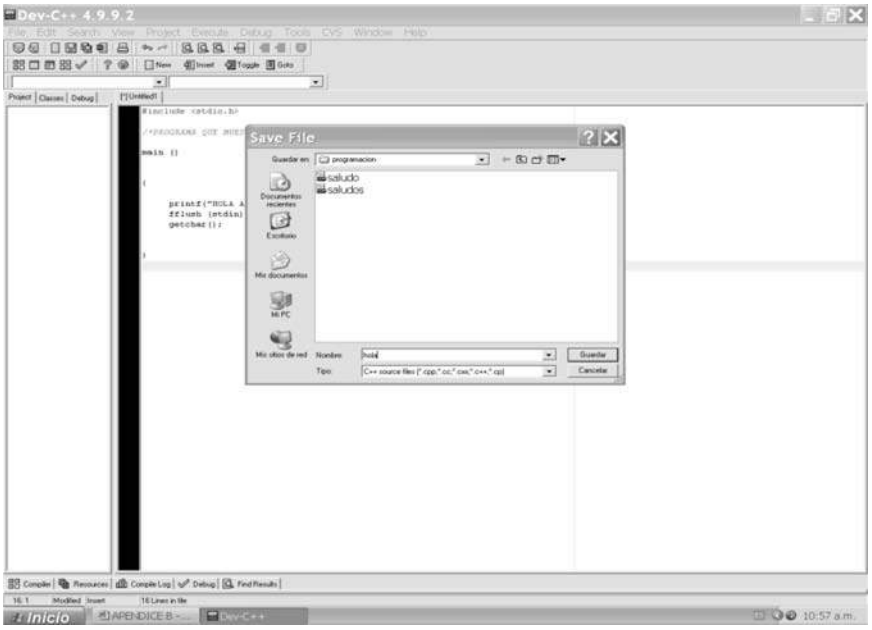


Imagen 10 mismo procedimiento

a) La primera opción para compilar es oprimiendo los botones **Ctrl** y **F9** simultáneamente, ver imagen 11.

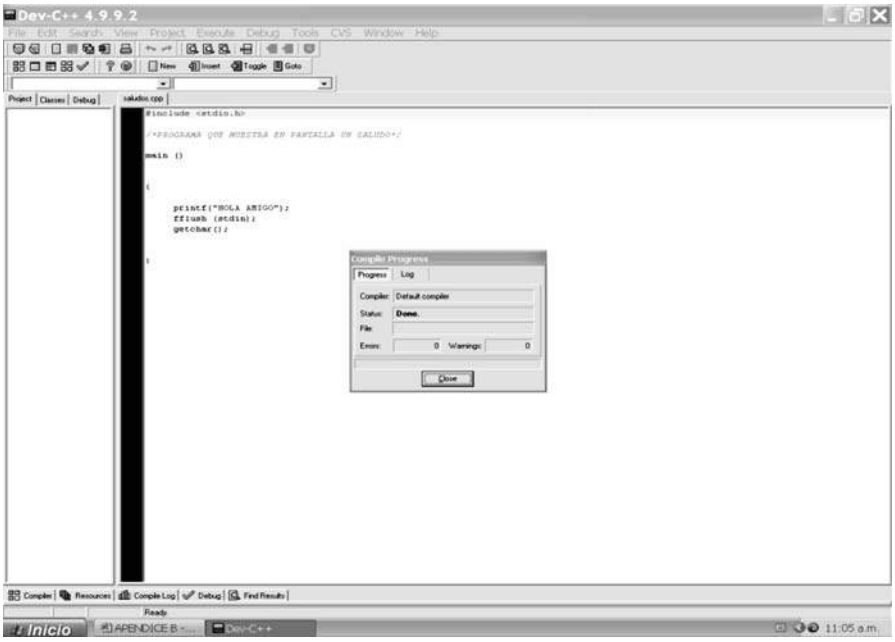


Imagen 11 ventana de compilación

b) La segunda es oprimiendo el botón que muestra cuatro cuadros de diferente color separados, el cual se encuentra en el lado izquierdo de la barra de herramientas, llamado **Compile**, se vera en imagen 12, el cual abrirá la ventana de compilación del programa, como en imagen 11.

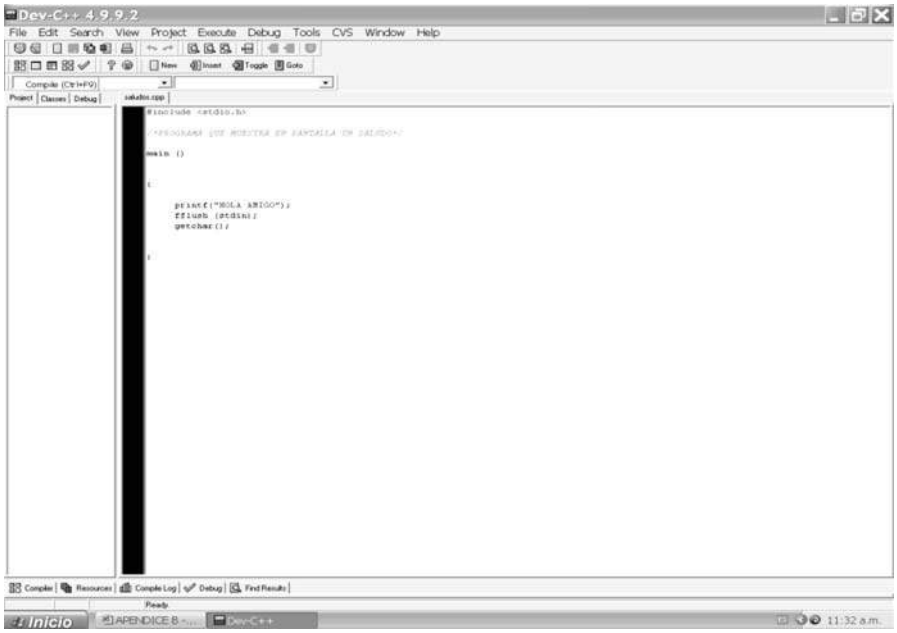


Imagen 12 muestra botón Compile

c) La tercera opción es oprimiendo el botón llamado **Execute** o Ejecutar ubicado en la barra de menús, el cual desplegará uno y seleccionarás la opción Compile según se observa en imagen 13, el cual abrirá la ventana de compilación del programa, tal como se mostro en imagen 11.

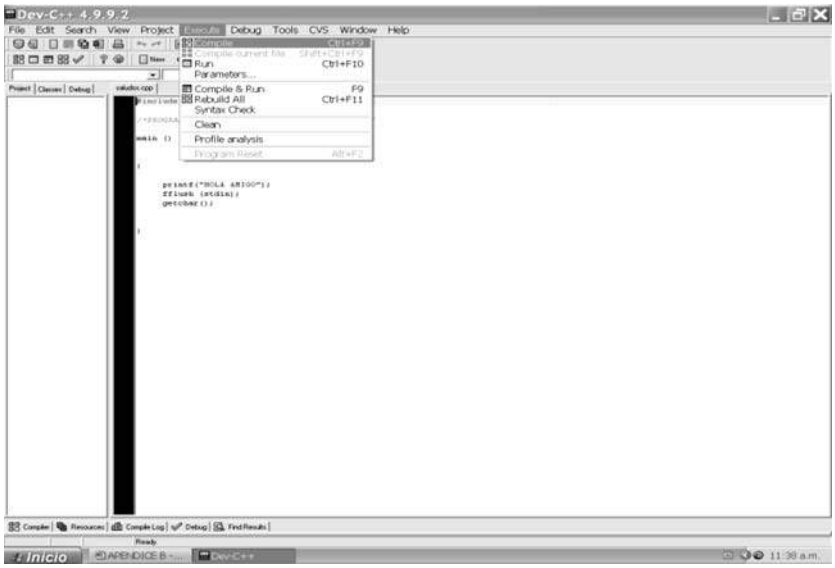


Imagen 13 mismo procedimiento

Posteriormente deberás ejecutar o correr tu programa, para realizar esta acción hay tres opciones y son:

a) La primera opción para ejecutar o correr el programa es oprimiendo los botones **Ctrl** y **F10** simultáneamente y aparecerá la ventana de ejecución del programa, se muestra en imagen 14.

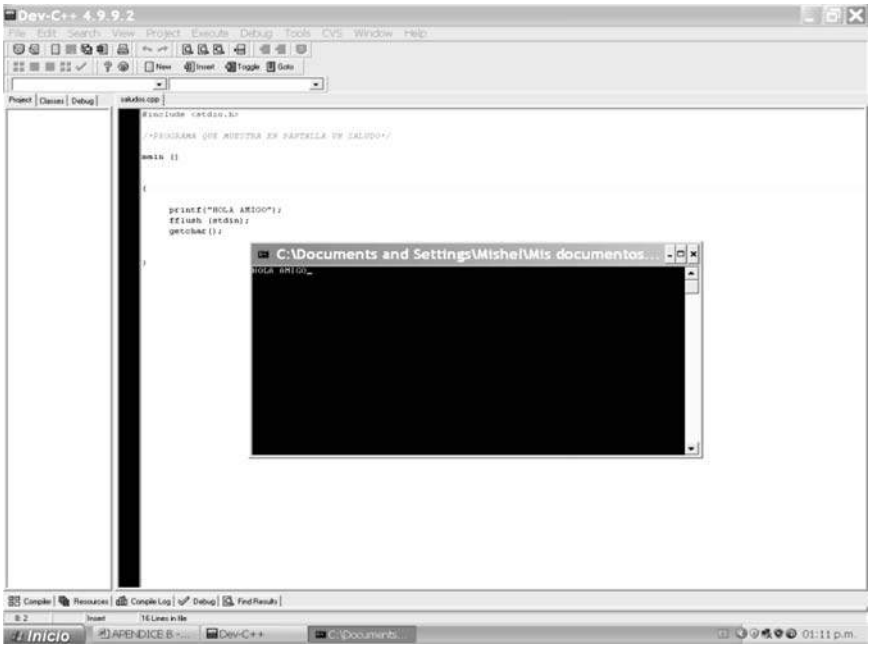


Imagen 14 desplegado de ventana de ejecución de programa

b) La segunda es oprimiendo el botón que se encuentra junto al de compilación en la barra de herramientas. Como se verá en imagen 15 y posteriormente aparecerá la ventana de ejecución de programa ver imagen 14.

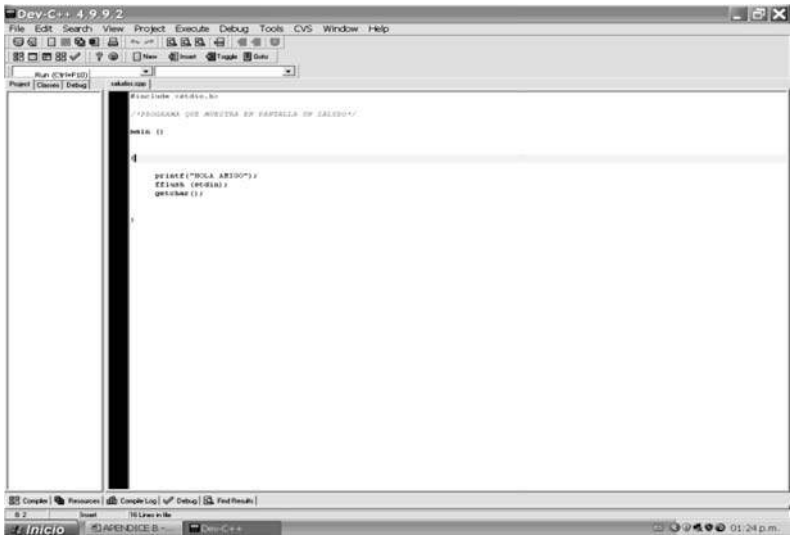


Imagen 15 mismo procedimiento

c) La tercera es oprimiendo el botón **Execute** o Ejecutar de la barra de menús, donde seleccionarás la opción **Run** o Ejecutar como en imagen 16 y después aparecerá la ventana de ejecución de programa ver imagen 14.

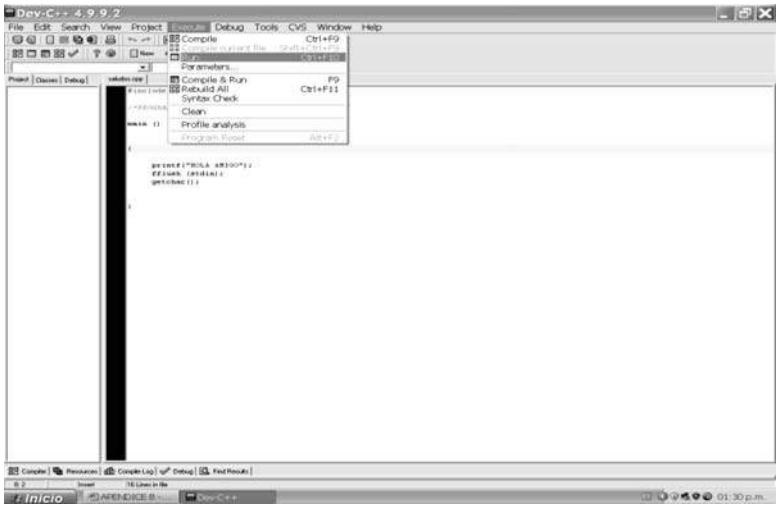


Imagen 16 mismo procedimiento

Para cerrar el programa puedes emplear la opción botones **Close All** ubicada

en el menú **File**, Ver imagen 17 u oprimir el botón Cerrar ubicado en el ángulo superior derecho con la imagen del tache, ver imagen 18.

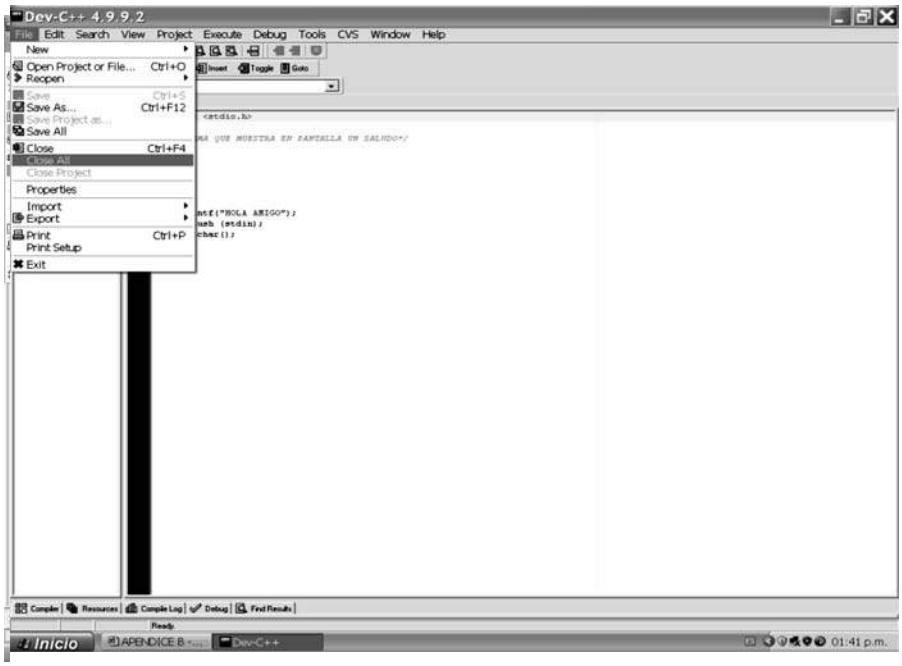


Imagen 17 cerrando el programa mediante la opción **Close All**

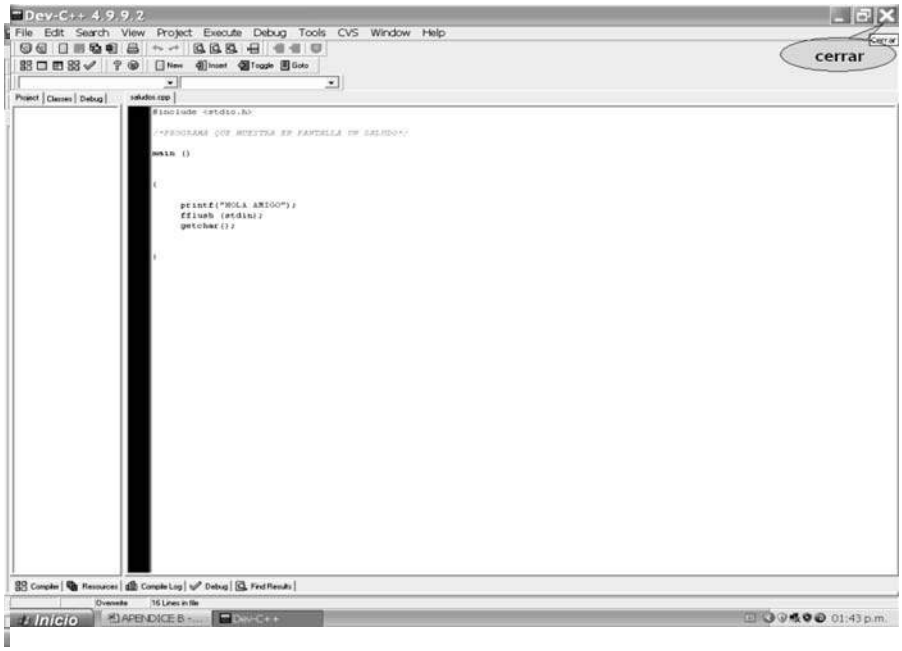


Imagen 18 mismo procedimiento

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice "C"
Paso de parámetros a funciones.

Tabla de contenido:

Introducción.-

Generalmente hay dos formas en las que pueden pasarse los argumentos a las funciones. La primera se denomina *llamada por valor*. Este método copia el valor de cada uno de los argumentos en los parámetros formales de la función. Con este método los cambios que se hacen a los parámetros de la función no tiene efecto en las variables utilizadas para llamar a la función.

Llamada por referencia es la segunda forma en que los argumentos se pueden pasar a las funciones. Con este método, la *dirección* de cada argumento se copia en los parámetros de la función. Esto significa que los cambios hechos en el parámetro afectarán a la variable utilizada para llamar a la función.

Las funciones del lenguaje C utilizan la llamada por valor. Esto significa, por lo general, que no se pueden alterar las variables utilizadas para llamar a la función.

Por ejemplo, en la siguiente función:

```
cuadrado( )
int x;
{
   x = x * x;
   return x;
           }
```

si el argumento de **cuadrado( )** es una constante, por ejemplo *10*, el valor *10* se copia en el parámetro **x**. Cuando la asignación **x = x * x** tenga lugar, lo único que se modifica es la variable local **x**. La constante *10* no queda afectada.

El mismo proceso tiene lugar cuando **cuadrado( )** se llama con una variable. Por ejemplo, si *y* es igual a *100*, la llamada **cuadrado(** *y* **)** copiará el valor de la variable *y* en el parámetro **x**. En ningún caso *y* se modificará así misma.

Recuerde que una copia del valor de la variable o constante utilizada para llamar a una función se pasa a esa función. Lo que ocurra dentro de la función no tendrá efecto en la variable utilizada en la llamada.

Debido a que todos los argumentos en el lenguaje C se pasan por valor, no es posible el cambiar las variables utilizadas en la llamada. Sin embargo, el lenguaje C permite simular una llamada por referencia utilizando apuntadores para pasar la dirección de una variable a una función y para cambiar la variable utilizada en la llamada. Los apuntadores son simplemente las direcciones de las variables. Se puede pasar una dirección a una función igual que cualquier otro valor.

Los operadores de apuntador & y *

En el lenguaje C, un apuntador es la dirección de memoria de una variable. El conocer la dirección de una variable puede representar una gran ayuda en ciertos tipos de rutinas. Sin embargo, en el lenguaje C los apuntadores tienen dos funciones básicas: la primera es que pueden proporcionar una forma muy rápida de referenciar los elementos de un arreglo (*array*), y la otra es que permite a las funciones C el modificar los parámetros que recibe.

El operador **&,** es un operador unario que devuelve la dirección de memoria del operando. Por ejemplo: **m** = **&cont**; coloca en **m** la dirección de memoria de la variable **cont**. Esta es la dirección de la posición interna de la variable en la computadora. No tiene nada que ver con el valor de **cont**.

Suponiendo que la variable **cont** utiliza la posición de memoria 2000 para almacenar su valor, y que este es 100. Después de la asignación **m** = **&cont** , **m** tendrá el valor 2000.

El operador *****, es un operador unario que devuelve el valor de la variable ubicada en la dirección que le siga.

Por ejemplo, si **m** contiene la dirección de memoria de la variable **cont**, entonces **q** = ***m** colocará el valor de **cont** en **q**. Siguiendo con este ejemplo, **q** tendrá el valor de 100, ya que 100 es lo almacenado en la posición 2000, que es la dirección de memoria donde está almacenada **m**.

Los operadores unarios, **&** y ***** tienen una precedencia mayor que cualquier operador aritmético, excepto el de menos unario, respecto al cual la tienen igual.

Las variables que mantendrán direcciones de memoria, o apuntadores, como se llaman en el lenguaje C, deben declararse colocando un ***** delante del nombre de la variable, para indicarle al compilador que esa variable va a contener un apuntador. Por ejemplo, para declarar una variable tipo apuntador llamada **ca** del tipo **char**, podemos escribir :

```
char *ca;
```

Pueden mezclarse indicaciones de apuntador con las que no lo son, en la misma declaración de la sentencia. Por ejemplo:

```
int x, *y, cont;
```

declara que **x** y **cont** sean del tipo entero, y que **y** sea un apuntador a un tipo entero.

En el programa siguiente, los operadores **&** y ***** se utilizan para poner el valor 10 en la variable llamada destino.

```
main( )   /* asignación con * y & */
{
     int destino, origen;
     int *m;
     origen = 10;
     m = &origen;
     destino = *m;
}
```

La función **intercambio( )**, que intercambia los valores de sus dos argumentos enteros, se escribe como:

```
intercambio( x, y )
int *x, *y;
{
   int temp;
   temp=*y;    // toma el valor en la dirección x
   *x = *y;    // pone y en x
   *y = temp;
}
```

Pero esto es solo la mitad. Después de que escriba la función **intercambio( )** hay que utilizar *las direcciones de las variables que se deseen intercambiar como argumentos, cuando se llama a la función*. El programa a continuación muestra la forma correcta de llamar a **intercambio( )**.

```
main( )
{
   int x, y;
   x = 10;
   y = 20;
   intercambio(&x,&y);
}
```

El programa asigna el valor 10 a la variable **x** y el valor 20 a la variable **y**. Luego se llama a **intercambio( )**, y se utiliza el operador unitario **&** para producir las direcciones de las variables **x** e **y**. Por lo tanto, las direcciones de x e y, no sus valores, son pasadas a la función **intercambio( )**.

Dentro de **intercambio( )**, el operador unario ***** se utiliza para indicar el valor de la posición apuntado por la dirección. En la sentencia **temp = *x**, el operador ***** produce el valor apuntado por la dirección en **x** y asigna a **temp** ese valor. En la sentencia ***x = *y**, el operador ***** dirige el valor a la posición de **y** para que se coloque en la posición encontrada en **x**. Luego la posición **y** se asigna al valor de la posición **x**.

Antes de que una función reciba un apuntador hay que definir el tipo de datos al que va a apuntar, pues sino se pueden obtener algunos resultados inesperados. Aunque normalmente los enteros y los caracteres se pueden mezclar libremente en el lenguaje C, no puede hacerse cuando se utilizan como apuntadores.

Llamada de funciones con arreglos.

Se puede utilizar el nombre de un arreglo de caracteres sin ningún índice, si se quiere llamar a una función con una cadena como argumento. Lo mismo es válido para todos los arreglos que se pasan como argumentos a las funciones. Sin embargo, el lenguaje C no copia el arreglo entero en la función. Cuando se llama a una función con el nombre de un arreglo, la dirección del primer elemento en el arreglo se pasa a la función. Esto significa que la declaración de parámetro debe ser un apuntador. Por ejemplo, si se quisiera escribir un programa que introdujera 10 números desde el teclado y los imprimiera, habría que escribir un programa parecido al siguiente:

```
main( )   /* imprime número */
{
   int t[10], i;

   for(i=0;i<10;++i) t[i]=getnum( );
         visualiza(t);
}

visualiza(num)
int *num;
{
   int i;

   for(i=0;i<10;i++) printf("%d ",num[i]);
}
```

En la función visualiza que se muestra, aunque el argumento que se ha declarado como apuntador entero, una vez dentro de la función, se puede accesar al arreglo completo utilizando la indexación normal en una arreglo. La razón de esto es que los arreglos en el lenguaje C realmente son apuntadores a una región de memoria,

y en esencia, la pareja **[ ]** es un operador que encuentra el valor de los datos con ese índice de arreglo especificado entre corchetes. Debido a que el lenguaje C no tiene comprobación de límites de los arreglos, a la función tampoco le preocupa cómo sea de largo el arreglo.

Un elemento de un arreglo utilizado como un argumento se trata igual que cualquier otra variable sencilla. Por ejemplo, el programa anterior se podría haber escrito sin pasar el arreglo completo, como se muestra en el programa siguiente.

```
main( )
{
   int  t[10], i;

   for(i=0;i<10;++i) t[i]=getnum( );
   for(i=0;i<10;i++) vizualiza(t[i]);
}

visuliza(num)
int num;
{
   printf("%d ",num);
}
```

El parámetro de **visualiza( )** es un entero. No tiene importancia el que se llama a **visualiza( )** utilizando un elemento entero de un arreglo, porque sólo se utiliza el valor del arreglo.

Sin embargo, cuando se utiliza el nombre de un arreglo como un argumento de función, se pasa por referencia. Se estará operando y alternado potencialmente el contenido real de los elementos del arreglo utilizandos para llamar a la función. Por ejemplo, el programa siguiente imprime una cadena en mayúsculas.

```
main( )   /* imprime cadena en mayúsculas */
{
    char s[80];
    gets(s);
    imprime_may(s);
}
imprime_may(cadena)
char *cadena;
{
    register int t;
    for(t=0;t<80;++t) {
                          cadena[t]=toupper(cadena[t]);
                        putchar(cadena[t]);
                      }
```

```
}
```

La función **toupper( )**, encontrada en la mayoría de las bibliotecas[9] de el lenguaje C, convertirá una minúscula en una mayúscula. Después de llamar a **imprime_may( )**, el contenido del arreglo **s** en **main( )** se cambiará a mayúsculas.

Si no se tiene la intención de cambiar el arreglo **s** permanentemente, se podría volver a escribir en mismo programa como se muestra a continuación.

```
main( )   /* imprime cadena en mayúsculas */
     {
         char s[80];
         int i;

         gets(s);
         for(t=0;t<80;t++) imprime_may_ca(s[t]);
     }

     imprime_may_ca(cadena)
     char *cadena;
     {
         register int t;

         cadena=toupper(cadena);
               putchar(cadena);
     }
```

En esta versión, los contenidos del arreglo s no se cambian porque sólo sus valores -no las direcciones- se pasan a la función **imprime_may_ca( )**.

Un ejemplo clásico del paso de arreglos a las funciones se encuentra en la función estándar **gets()**. Después de que se llame con un argumento arreglo de un único carácter, **gets()** devolverá una cadena que será la introducida desde el teclado. Aunque **gets()** en su librería[1] estándar es mucho más sofisticada y compleja, la función que se muestra en el siguiente listado le dará una idea de cómo trabaja. Para evitar confusiones con la función estándar, a esta le denominaremos **xgets()**.

[9] Library en inglés, traducible como librería o biblioteca, corresponde aun conjunto de funciones en lenguaje C.

```
xgets(s)   /* una versión muy simple de la función gets( ) de
              la librería estándar */
char *s;
{
   char ch;
   int t;

   for(t=0;t<80;++t) {
           ch=getchar( );
                 switch(ch) {
                     case '\n':
                          s[t]='\0';    /*  fin  de  la  cadena
*/
                    return;
                     case '\b':
                          if(t>0) t--;
                     break;
                     default::
                     s[t]=ch;
                     break;
                                              }
                                          }
    s[80]='0',
}
```

A la función **xgets( )** hay que llamarla con un apuntador a carácter, que puede ser o bien una variable que se declare como un apuntador a carácter, o el nombre de un arreglo de carracteres, que por definición es un apuntador a carácter. Una vez iniciada, **xgets( )** establece un bucle **for** de 0 a 80. Esto evita el que se introduzcan desde el teclado cadenas muy largas. (La mayoría de las funciones **gets( )** de librerías[1] estándar no tienen un límite, en ésta se incluye para ilustrar la comprobación de límites manuales para evitar el error de sobrecarga del arreglo.) Si se teclean más de 80 caracteres, la función acabará y volverá.

Debido a que el lenguaje C no incluye límites de comprobación, hay que asegurarse de que cualquier variable utilizada para llamar a **xgets( )** puede aceptar por lo menos 80 caracteres. A medida que se teclean los caracteres, estos se introducen a la cadena.. Si se pulsa un retroceso, el contador **t** se reduce en 1. Cuando se pulsa un retorno de carro, al final de la cadena se coloca un nulo, lo que señala su terminación. Debido a que el arreglo original utilizado para llamar a **xgets( )** se modifica, a la vuelta contendrá los caracteres que se han tecleado.

No se podrían escribir ni **intercambio( )**, ni **xgets( )** sin utilizar apuntadores para crear una llamada por referencia. Al escribir tal función, recuerde que hay que pasar la *dirección* de la variable que se va a cambiar a la función. Dentro de la función hay que ejecutar todas las operaciones sobre la variable utilizando el operador unario *****.

El operador unario ***** puede interpretarse como "en la dirección". Por ejemplo, en:

```
x = &z;
*x = 10;
y = *x;
```

la primera sentencia de asignación, **x = &z**; se puede leer como "asignar a **x** la dirección de **z**". La segunda sentencia de asignación, ***x = 10**; se puede leer como "en la dirección **x**, poner el valor 10". La última sentencia, **y = *x**; se puede leer como "**y** igual al valor en la dirección **x**".

Recuerde que el unario **&** significa "la dirección de" y el ***** "en la dirección". Si consigue recordar lo que hacen los operadores, tendrá pocos problemas al usarlos.

Devolución de apuntadores.

Aunque las funciones que devuelven apuntadores se manejan exactamente igual que cualquier otro tipo de función, examinaremos unos cuantos conceptos importantes.

Los apuntadores a variables no son enteros ni enteros sin signo. Son apuntadores. La razón de esta distinción estriba en el hecho de que los apuntadores se pueden incrementar y decrementar. Cada vez que un apuntador se incrementa, por ejemplo, el apuntador apuntará al dato siguiente de su tipo. Debido a que cada tipo de dato puede ser de distinta longitud, el lenguaje C tiene que saber a qué tipo de dato está apuntando el apuntador para hacer que señale al dato siguiente. Es fundamental el no tratar de utilizar enteros para devolver direcciones de variables.

Si se quiere escribir una función que devuelva un apuntador a cadena, a la cadena donde se encontró una coincidencia de caracteres, se puede utilizar esto:

```
char *conicide(c,s);
char c, *s;
{
    int cuenta;
    cuenta=0;
          while(c!=s[cuenta] && s[cuenta]1='\0')
          cuenta++;
    return(&s[cuenta]);
}
```

La función **coincide( )** intentará devolver un apuntador al lugar de la cadena en la que se encontró la primera igualdad con el carácter **c**. Si no se encuentra una igualdad, se devolverá un apuntador al terminador nulo.

A continuación se muestra un programa corto que utiliza **coincide( )**. Este programa lee una cadena y luego un carácter. Si el carácter está en la cadena, el programa imprime la cadena desde el punto de coincidencia. En otro caso, imprime el mensaje "no hay coincidencia."

```
main( )
{
   char car, s[80], *p;

   gets(s);
   car=getchar( );
         p=coincide(car,s);
   if(p!=0) printf("%s ",p);   /* hay coincidencia */
   else printf("no hay coincidencia. ");
}
```

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice “D”
Estructuras.

Tabla de contenido:

Concepto de estructura

Una ***estructura*** es una agrupación de variables bajo un nombre y se clasifica como un tipo de datos agregado. Las variables utilizadas en una estructura pueden ser de diferentes tipos. Las variables que componen la estructura son llamados **"miembros"** de la estructura también se les conoce como **"elementos"** o **"campos"**.

A continuación se presenta la forma típica de declarar una estructura y sus miembros en lenguaje C.

```
struct cuenta_banco {
          int no_cuenta;
          char tipo_cuenta;
          char nombre_cliente [80];
          float saldo_cuenta;
          } cuentas_bancarias;
```

El identificador que se observa después de la instrucción **"struct"** y antes de **{** es el **"nombre de la estructura"** o **"etiqueta"** en algunos casos también se denomina **"marca"**.

El identificador que se encuentra entre **}** y **;** se llama **"variables de estructura"**.

Nota: En algunos casos al declarar una estructura, se puede omitir escribir las **"variables de estructura"** o la **"etiqueta"** aunque no ambas. Los casos anteriores quedan fuera del alcance de estos apuntes, solo se mencionan para información del lector.

Según se observa dentro de una estructura se puede utilizar un arreglo como parte de la misma.

La estructura anterior tiene como nombre **"cuenta_banco"** y contiene 4 miembros **"no_cuenta"** que es del tipo **int**, **"tipo_cuenta"** que es un caracter solo, **"nombre_cliente"** un arreglo de caracteres que puede medir hasta ochenta letras y finalmente el miembro **"saldo_cuenta"** es del tipo **float**.

Observe que la declaración de una estructura termina con **;** debido a que se trata de una instrucción o sentencia. Las líneas encerradas entre los simbolos **{ }** son parte de la instrucción de declaración de la estructura y no un bloque de instrucciones.

Manejo de estructura

A continuación se presenta el ejemplo de un programa que maneja una estructura en la cual se cuenta con diferentes elementos como ***float*** y cadenas de caracteres, ahi se observa la forma de definir la estructura y sus miembros. También en este programa se puede observar cuál es la forma de leer desde el teclado el valor de sus elementos así como la forma de imprimirlos.

```
#include <stdio.h>   // ***** estructura.c *****
int main() {

     struct Datos_Estudiantes {
     char nombre[50];
     char matricula[9];
     float Ev_1;
     float Ev_2;
     float Ev_Final;
     float resultado;
     } Estudiante;

     printf("Introducir los siguientes datos: \n ");
     printf(" Nombre completo del estudiante \n");
      gets(Estudiante.nombre); /* Lee el nombre como una cadena
                                  de caracteres */
     printf(" Matricula del estudiante \n");
      gets(Estudiante.matricula); /* Lee la matricula como una
                                     cadena de caracteres */
     printf(" Primera calificacion parcial del estudiante \n");
      scanf("%f", &(Estudiante.Ev_1));
     printf(" Segunda calificacion parcial del estudiante \n");
      scanf("%f", &(Estudiante.Ev_2));
     printf(" Calificacion de final del estudiante \n");
      scanf("%f", &(Estudiante.Ev_Final));
            fflush(stdin);  /* Limpia el buffer de la entrada
                               estandar del sistema */
/* El resultado del estudiante pondera 25% por cada parcial y 50%
                     la evaluación final */
      Estudiante.resultado = Estudiante.Ev_1*0.25 +
      Estudiante.Ev_2*0.25 + Estudiante.Ev_Final*0.5;

     printf("\n Resultados finales del estudiante \n");
     printf(" Nombre: %s \n", Estudiante.nombre);
     printf(" Matricula: %s \n", Estudiante.matricula);
     printf(" Evaluacion del curso: %.2f \n",
              Estudiante.resultado);
          }
```

Este programa lee la cadena de caracteres del nombre del estudiante y su matrícula como otra cadena. Además se leen tres calificaciones dos parciales y una final. El programa calcula el resultado del estudiante considerando una ponderación del 25% por cada evaluación parcial y 50% de la evaluación final.

Se accede a la escritura o lectura de los datos de la estructura a través del identificador de las "variables de estructura" en conjunto con el nombre de alguno de los elementos de la estructura, ambos se operan con **"."**

Por ejemplo obsérvese la siguiente línea de código del programa:

```
gets(Estudiante.matricula);
```

Esta sentencia o instrucción se encarga de leer del teclado una cadena de caracteres que termina con la tecla "Enter" o "Intro". Una vez que la cadena se ha leído se guarda en el arreglo matrícula de la estructura "Estudiante".

Esta notación es semejante a la forma de manipular objetos y se utiliza en la siguiente generación de lenguajes de programación.

Esta siguiente generación de lenguajes toma su nombre de su principal característica que es la Orientación a Objetos, donde básicamente se modelan los elementos de un programa, pensando en objetos que pueden ser de la vida cotidiana y a cada objeto le corresponden ciertas "propiedades" o "atributos". Por ello se utilizan las siglas "POO" para resumir "Programación Orientada a Objetos"

Ejercicio propuesto

Nota aclaratoria para resolver el ejercicio: el programa de este apéndice tiene el nombre escrito a manera de comentario en la primera línea del código.

A partir del código del programa "estructura.c" realice un programa donde aplique una estructura distinta con otros miembros.

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice "E"
Funciones matemáticas.

Tabla de contenido:

FUNCIONES MATEMÁTICAS

El lenguaje C reconoce algunas funciones matemáticas que se encuentran en la biblioteca ***math.h***, por lo que al hacer uso de éstas, es necesario incluir dicha biblioteca en las directivas de preprocesador[10]. Estas funciones son de gran utilidad, ya que reducen la tarea del programador, así como el código del programa, y por lo tanto la ejecución del mismo.
En ANSI C89, todas las funciones operan con el tipo de dato ***double***, el resultado devuelto es también de tipo ***double***.
A continuación se mencionas algunas funciones matemáticas de ANSI C89.

FUNCIÓN	DESCRIPCIÓN
acos(*valor*)	Calcula el *arco coseno* de *valor*
asin(*valor*)	Calcula el *arco seno* de *valor*
atan(*valor*)	Calcula el *arco tangente* de *valor*
cbrt(*valor*)	Calcula la raíz cúbica de *valor* *
ceil(*valor*)	Calcula el entero más pequeño que es mayor o igual a *valor*
cos(*valor*)	Calcula el *coseno* de *valor*
exp(*valor*)	Calcula *e* elevado a la potencia *valor*
exp2(*valor*)	Calcula *2* elevado a la potencia *valor**
fabs(*valor*)	Calcula el *valor absoluto* de *valor*
floor(*valor*)	Calcula el entero más grande que el menor o igual que *valor*
log(*valor*)	Calcula el *logaritmo* natural de *valor*
log10(*valor*)	Calcula el *logaritmo en base 10* de *valor*
pow(*base*, *exponente*)	Calcula *base* a la potencia *exponente*
sin(*valor*)	Calcula el *seno* de *valor*
sqrt(*valor*)	Calcula la *raíz cuadrada* de *valor.*
tan(*valor*)	Calcula la *tangente* de *valor*

* Estas funciones sólo están definidas en ANSI C99.

ANSI C99 ha añadido para todas las funciones C89 y para las de C99 dos variantes: una que funciona con tipo de dato *float* y otra con tipo de dato *long double*. Por ejemplo, la función *acos()*, opera con tipo *double*, ahora se tiene otra que se llama *acosf()*, que opera con tipo *float* y otra que se denomina *acosl()* que opera con tipo *long double*.

BIBLIOGRAFÍA

López Román Leobardo
Programación estructurada en lenguaje C
México: Alfaomega, 2005.

[10] En el capítulo 3 se explica qué son las directivas de preprocesador.

ACADEMIA DE INFORMÁTICA
CURSO: INTRODUCCIÓN A LA PROGRAMACIÓN

Apéndice “F”
Compilar y ligar un programa en Linux

Tabla de contenido:

Compilar y ligar un programa en Linux

Nota: Para ligar y compilar un programa en lenguaje C, existen diferentes maneras de hacerlo y estas se encuentran en función del Sistema Operativo y compilador que se usará. Si la compilación se realizara bajo el S.O. **Windows** y con el compilador **Dev-Cpp**, ver el apéndice **B** de estos apuntes. Las explicaciones que se presentan a continuación, aplican para el S.O. **Linux** y el compilador **gcc**.

Considere un programa llamado `hola.c` escrito en algún editor de texto:

```
#include <stdio.h>
main() {
   printf("\n Hola Mundo\n");
}
```

Si usamos la terminal en Linux, el comando que compila y liga a dicho programa es:

```
$gcc hola.c
```

Si no hay errores, el comando anterior, genera un archivo ejecutable cuyo nombre es:

`a.out`.

También se puede usar:

```
$gcc hola.c –o hola.out
```

Que genera el mismo ejecutable pero con el nombre `hola.out`, ya que al agregarle la opción –o estamos indicando que el ejecutable tenga ese nombre: `hola.out`.

Si usas funciones matemáticas (`pow()`, `sqrt()`, `sin`(), etc.) mas elaboradas que las operaciones aritméticas +, -, *, /, podrás compilar primero y luego ligar, para generar un ejecutable:

```
$gcc –c hipotenusa.c
$gcc –lm hipotenusa.o –o hipotenusa.out
```

Donde con la opción `-c` indicamos que se compile el archivo fuente lo cual producirá un archivo objeto `hipotenusa.o` y con la opción `-lm` indicamos que se ligue ese archivo objeto con las librerías matemáticas, lo cual finalmente producirá un archivo ejecutable llamado `hipotenusa.out`.

Ejemplos de programas:

Compile, ligue y ejecute los siguientes programas:

```
//Programa 1
#include <stdio.h>
#include <math.h>

main() {
  double a=0.0, b=0.0, c=0.0;

  printf("\nIntroduzca los catetos a y b:\n");
  scanf("%lf  %lf", &a, &b);
  c= sqrt(pow(a,2)+ pow(b,2));
  printf("\n La hipotenusa de a=%lf  y b=%lf es: %lf\n",
             a,b,c);
}
```

```
//Programa 2
#include <stdio.h>
main() {
   char x, y[15];

   printf("Introduzca una cadena de texto no mayor a 10
            letras:");
   scanf("%c%s", &x,y);
   printf("\nEl carácter inicial fue: ", x);
   printf("\nEl resto de los caracteres fue: %s\n", y);
}
```

```
//Programa 3
#include <stdio.h>
main() {
   char letra='A';
   char texto[30]= "Que tengan éxito";
   char mensaje[12];

   printf("\n Este carácter inicia el alfabeto: %c\n",
            letra);
   printf("Introduzca un texto no mayor a 10 letras:");
   scanf("%s", mensaje);
   printf("\nEste texto es el incluido: %s\n",  texto);
   printf("\nIntrodujiste este mensaje : %s\n", mensaje);
}
```

Esta obra se terminó de imprimir en el mes de septiembre de 2008
en el taller de impresión de la
Universidad Autónoma de la Ciudad de México
con un tiraje de 1000 ejemplares.